A Formally Verified Interpreter for a Shell-like Programming Language

Claude Marché Nicolas Jeannerod Ralf Treinen

INSTITUT

EN INFORMATIQUE

Vals seminar, July 7, 2017

Nico	as .	lean	ner	od
	-uo -	can		

VALS Seminar

EL OQO

1 / 61

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.

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.

- 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.

- 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.

< □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □

- 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.

- 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.

▲□ ▲ □ ▲ □ ▲ □ ▲ □ ■ □

- 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.

- 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

1. Language

- Elements of Shell
- CoLiS
- Mechanised version
- 2. A sound interpreter
 - Why?
 - Let us see some code
 - Proof
 - An other sound interpreter
- 3. A complete interpreter
 - Which formulation?
 - Heights and sizes
 - Skeletons

< 🗗 🕨

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"
```

5 / 61

< □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □

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 = f \circ o
a=bar f $a ## echoes "foo bar"
                ## echoes "bar"
echo $a
```

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
```

< □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □

Elements of Shell

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	^{1,1} 2>	¹ ,12,*	Return 0	Return 1.125	£t.ii	1. 12. 1. 12.>
Pipe			Normal				
Sequence	Norm	nal	al Exception				
Test	True	False		Exception			
Function call	Success	Failure		Success	Failure	Exception	
Subprocess	Success	Failure		Success	Failure	Success	Failure

Nicolas Jeannerod

VALS Seminar

July 7, 2017

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"

	Nico	las .	lean	nerod
--	------	-------	------	-------

< □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □ > < □

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"

	Nicol	as .	lean	neroc
--	-------	------	------	-------

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"

NICO	20	000	nor	$\sim a$
INTUCO.	Id S		ner	υu

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

1. Language

• Elements of Shell

CoLiS

- Mechanised version
- 2. A sound interpreter
 - Why?
 - Let us see some code
 - Proof
 - An other sound interpreter
- 3. A complete interpreter
 - Which formulation?
 - Heights and sizes
 - Skeletons

< ∃ ►

↓ ∃ ↓ ∃ | = √ Q ∩

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
 - 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
 - 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
 - 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	Xs	\in	SVar
List variables	xı	\in	LVar
Procedures names	с	\in	${\cal F}$

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

Syntax – 2

Terms t ::= true | false | fatal **return** $t \mid exit$ t $| x_s := s | x_l := l$ | t; t | if t then t else t for x_s in / do t | while t do t **process** $t \mid pipe t$ into tcall / | shift

Syntax – 3

String expressions $s ::= nil_s | f_s :: s$ String fragments $f_s ::= \sigma \mid x_s \mid n \mid t$ List expressions $I ::= \mathbf{nil}_I \mid f_I :: I$ List fragments $f_l ::= s \mid \text{split } s \mid x_l$

Nicolas Jeannerod

13 / 61

Semantics – First definitions

Behaviours: terms $b \in \{\text{True}, \text{False}, \text{Fatal}, \text{Return True}\}$ Return False, Exit True, Exit False}

Behaviours: expressions $\beta \in \{\text{True}, \text{Fatal}, \text{None}\}$

▲□▶ ▲□▶ ▲三▶ ▲三▶ 三回日 のの⊙ July 7, 2017

14 / 61

Semantics – First definitions

Behaviours: terms $b \in \{\text{True}, \text{False}, \text{Fatal}, \text{Return True}\}$ Return False, Exit True, Exit False}

Behaviours: expressions $\beta \in \{\text{True}, \text{Fatal}, \text{None}\}$

Environments: strings

Environments: lists

Contexts

 $SEnv \triangleq [SVar \rightarrow String]$

$$LEnv \triangleq [LVar
ightarrow StringList]$$

 $\Gamma \in \mathcal{FS} \times String \times StringList$ \times SEnv \times 1 Env

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

Nico	las .	leanr	ierod

VALS Seminar

▲□▶ ▲□▶ ▲三▶ ▲三▶ 三回日 のの⊙ July 7, 2017

Semantic judgments

Judgments:	terms	$t_{/\Gamma}$	₩	$\sigma \star b_{/\Gamma'}$
Judgments: Judgments:	string fragment string expression	$f_{s/\Gamma} \ s_{/\Gamma}$	⊎ _{sf} ⊎ _s	$\sigma \star \beta_{/\Gamma'}$ $\sigma \star \beta_{/\Gamma'}$
Judgments: Judgments:	list fragment list expression	f _{I/Γ} I _{/Γ}	₩ ⊮ ₩I	$\lambda \star \beta_{/\Gamma'}$ $\lambda \star \beta_{/\Gamma'}$

A few rules – Sequence

$$\frac{\sum_{t_{1/\Gamma} \Downarrow \sigma_{1} \star b_{1/\Gamma_{1}}}{b_{1} \in \{\mathsf{True}, \mathsf{False}\}} \quad t_{2/\Gamma_{1}} \Downarrow \sigma_{2} \star b_{2/\Gamma_{2}}}{(t_{1} ; t_{2})_{/\Gamma} \Downarrow \sigma_{1}\sigma_{2} \star b_{2/\Gamma_{2}}}$$

$$\frac{b_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \quad b_1 \in \{\text{Fatal}, \text{Return _}, \text{Exit _}\}}{(t_1 ; t_2)_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}}$$

Nicolas Jeannerod

VALS Seminar

July 7, 2017

A few rules – Sequence

$$\frac{\sum_{t_{1/\Gamma} \Downarrow \sigma_{1} \star b_{1/\Gamma_{1}}}{b_{1} \in \{\mathsf{True}, \mathsf{False}\}} \quad t_{2/\Gamma_{1}} \Downarrow \sigma_{2} \star b_{2/\Gamma_{2}}}{(t_{1} ; t_{2})_{/\Gamma} \Downarrow \sigma_{1}\sigma_{2} \star b_{2/\Gamma_{2}}}$$

SEQUENCE-EXCEPTION $t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\text{Fatal}, \text{Return}_-, \text{Exit}_-\}$ $(t_1; t_2)_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}$

Nicolas Jeannerod

▲□▶ ▲□▶ ▲三▶ ▲三▶ 三回日 のの⊙ July 7, 2017
$\frac{BRANCHING-TRUE}{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}} \quad b_1 = \text{True} \quad t_{2/\Gamma_2} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \sigma_2 \star b_{2/\Gamma_2}}$

 $\begin{array}{c} \text{BRANCHING-FALSE} \\ \underline{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}} \quad b_1 \in \{\text{False}, \text{Fatal}\} \quad \underline{t_{3/\Gamma_3} \Downarrow \sigma_3 \star b_{3/\Gamma_3}} \\ \hline \\ \hline \\ (\text{if } t_1 \text{ then } t_2 \text{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \sigma_3 \star b_{3/\Gamma_3} \end{array}$

BRANCHING-EXCEPTION $t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\text{Return }_, \text{Exit }_\}$ (if t_1 then t_2 else $t_3)_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}$

▲□▶ ▲□▶ ▲三▶ ▲三▶ 三回日 ののの

 $\frac{B\text{RANCHING-TRUE}}{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}} \quad b_1 = \text{True} \quad t_{2/\Gamma_2} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \sigma_2 \star b_{2/\Gamma_2}}$

$$\begin{array}{c|c} \text{BRANCHING-FALSE} \\ \underline{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}} & b_1 \in \{\text{False}, \text{Fatal}\} & \underline{t_{3/\Gamma_3} \Downarrow \sigma_3 \star b_{3/\Gamma_3}} \\ \hline & (\text{if } t_1 \text{ then } t_2 \text{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \sigma_3 \star b_{3/\Gamma_3} \end{array}$$

BRANCHING-EXCEPTION $t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\text{Return }_-, \text{Exit }_-\}$ (if t_1 then t_2 else t_3)_{/ Γ} $\Downarrow \sigma_1 \star b_{1/\Gamma_1}$

▲□▶ ▲□▶ ▲三▶ ▲三▶ 三回日 ののの

$$\frac{\text{BRANCHING-TRUE}}{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}} \quad b_1 = \text{True} \quad t_{2/\Gamma_2} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \sigma_2 \star b_{2/\Gamma_2}}$$

$$\begin{array}{c|c} \text{BRANCHING-FALSE} \\ \underline{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}} & b_1 \in \{\text{False}, \text{Fatal}\} & \underline{t_{3/\Gamma_3} \Downarrow \sigma_3 \star b_{3/\Gamma_3}} \\ \hline & (\text{if } t_1 \text{ then } t_2 \text{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \sigma_3 \star b_{3/\Gamma_3} \end{array}$$

$$\frac{BRANCHING-EXCEPTION}{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \quad b_1 \in \{\text{Return }_, \text{Exit }_\}}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}}$$

CoLiS

How behaviours are handled

	L'us	False	Fatal	Return True	Return False	Inue Inue	Etrit False
Pipe	Normal						
Sequence	Norm	al			Exceptio	n	
Test	True	Fa	lse	Exception			
Function call	Success	Failure		Success	Failure	Exception	
Subprocess	Success	Fai	lure	Success	Failure	Success	Failure

A few rules – Mutual recursion

Terms depend on string expressions:

$$\frac{ASSIGNMENT-STRING}{s_{/\Gamma} \Downarrow_{s} \sigma \star \beta_{/\Gamma'}} \frac{s_{/\Gamma} \Downarrow_{s} \sigma \star \beta_{/\Gamma'}}{x := s_{/\Gamma} \Downarrow_{```} \star \beta_{/\Gamma'[\text{senv}=\Gamma'.\text{senv}[x \leftarrow \sigma]]}}$$

and string fragments depend on terms:

 $\frac{\text{STRING-SUBPROCESS}}{t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}} \frac{t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}}{t_{/\Gamma} \Downarrow_{\text{sf}} \sigma \star \overline{b}_{/\Gamma'}}$

\overline{b} := True if $b \in \{$ True, Return True, Exit True $\}$ | Fatal otherwise

19 / 61

A few rules – Mutual recursion

Terms depend on string expressions:

$$\begin{array}{c} \operatorname{Assignment-String} \\ \frac{s_{/\Gamma} \downarrow_{s} \sigma \star \beta_{/\Gamma'}}{x := s_{/\Gamma} \Downarrow ``` \star \beta_{/\Gamma'[\operatorname{senv}=\Gamma'.\operatorname{senv}[x \leftarrow \sigma]]}} \end{array}$$

and string fragments depend on terms:

 $\begin{array}{l} \text{String-Subprocess} \\ \frac{t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}}{t_{/\Gamma} \Downarrow_{\text{sf}} \sigma \star \overline{b}_{/\Gamma'}} \end{array}$

 $\overline{b} := \text{True} \quad \text{if } b \in \{\text{True}, \text{Return True}, \text{Exit True}\}$ | Fatal otherwise

Nicolas Jeannerod

VALS Seminar

July 7, 2017

Table of Contents

1. Language

- Elements of Shell
- CoLiS
- Mechanised version
- 2. A sound interpreter
 - Why?
 - Let us see some code
 - Proof
 - An other sound interpreter
- 3. A complete interpreter
 - Which formulation?
 - Heights and sizes
 - Skeletons

▲■ ▶ ▲ 臣 ▶ ▲ 臣 ▶ 三目目 のへの

• 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.

21 / 61

- 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.

- 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.

- 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.

▲ Ξ ► Ξ Ξ < < < </p>

- 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.

- 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.

ELE DOG

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 $t_1 \ \ \sigma_1 \ b_1 \ \ \Gamma_1 \ t_2 \ \ \sigma_2 \ \ b_2 \ \ \ \Gamma_2$. eval_term $t_1 \ \Gamma \ \sigma_1$ (BNormal b_1) $\Gamma_1 \rightarrow$ eval_term $t_2 \Gamma_1 \sigma_2 b_2 \Gamma_2 \rightarrow$ eval_term (TSeq $t_1 t_2$) Γ (concat $\sigma_1 \sigma_2$) $b_2 \Gamma_2$

A few rules – Sequence

```
EvalT_Seq_Normal : forall t_1 \ \ \sigma_1 \ b_1 \ \ \Gamma_1 \ t_2 \ \ \sigma_2 \ \ b_2 \ \ \ \Gamma_2.
eval_term t_1 \ \Gamma \ \sigma_1 (BNormal b_1) \Gamma_1 \rightarrow
eval term t_2 \Gamma_1 \sigma_2 b_2 \Gamma_2 \rightarrow
eval_term (TSeq t_1 t_2) \Gamma (concat \sigma_1 \sigma_2) b_2 \Gamma_2
```

```
EvalT_Seq_Error : forall t_1 \ \ \sigma_1 \ b_1 \ \ \Gamma_1 \ t_2.
eval_term t_1 \ \Gamma \ \sigma_1 \ b_1 \ \Gamma_1 \rightarrow
(match b_1 with BNormal _ -> false | _ -> true end) ->
eval_term (TSeq t_1 t_2) \Gamma \sigma_1 b_1 \Gamma_1
```

```
EvalT_If_True : forall t_1 \ \Gamma \ \sigma_1 \ \Gamma_1 \ t_2 \ \sigma_2 \ b_2 \ \Gamma_2 \ t_3.
eval_term t_1 \ \Gamma \ \sigma_1 (BNormal True) \Gamma_1 \rightarrow
eval_term t_2 \ \Gamma_1 \ \sigma_2 \ b_2 \ \Gamma_2 \rightarrow
eval_term (TIf t_1 \ t_2 \ t_3) \Gamma (concat \sigma_1 \ \sigma_2) b_2 \ \Gamma_2
```

```
| EvalT_If_Transmit : forall t_1 \mid \sigma_1 b_1 \mid_1 t_2 t_3.
eval_term t_1 \mid \sigma_1 b_1 \mid_1 ->
(match b_1 with BReturn _ | BExit _ -> true | _ -> false end)
eval_term (TIf t_1 t_2 t_3) \mid \sigma_1 b_1 \mid_1
```

イロト 不得 トイヨト イヨト ヨヨ のくや

```
| EvalT_If_True : forall t_1 \ \Gamma \ \sigma_1 \ \Gamma_1 \ t_2 \ \sigma_2 \ b_2 \ \Gamma_2 \ t_3.
eval_term t_1 \ \Gamma \ \sigma_1 (BNormal True) \Gamma_1 \rightarrow
eval_term t_2 \ \Gamma_1 \ \sigma_2 \ b_2 \ \Gamma_2 \rightarrow
eval_term (TIf t_1 \ t_2 \ t_3) \Gamma (concat \sigma_1 \ \sigma_2) b_2 \ \Gamma_2
```

```
EvalT_If_Transmit : forall t_1 \ \lceil \sigma_1 \ b_1 \ \rceil_1 \ t_2 \ t_3.
eval_term t_1 \ \lceil \sigma_1 \ b_1 \ \lceil_1 \ ->
(match b_1 with BReturn _ | BExit _ -> true | _ -> false end)
eval_term (TIf t_1 \ t_2 \ t_3) \ \lceil \sigma_1 \ b_1 \ \rceil_1
```

```
| EvalT_If_True : forall t_1 \ \Gamma \ \sigma_1 \ \Gamma_1 \ t_2 \ \sigma_2 \ b_2 \ \Gamma_2 \ t_3.
eval_term t_1 \ \Gamma \ \sigma_1 (BNormal True) \Gamma_1 \rightarrow
eval_term t_2 \ \Gamma_1 \ \sigma_2 \ b_2 \ \Gamma_2 \rightarrow
eval_term (TIf t_1 \ t_2 \ t_3) \Gamma (concat \sigma_1 \ \sigma_2) b_2 \ \Gamma_2
```

```
| EvalT_If_False : forall t_1 \ \Gamma \ \sigma_1 \ b_1 \ \Gamma_1 \ t_3 \ \sigma_3 \ b_3 \ \Gamma_3 \ t_2.
eval_term t_1 \ \Gamma \ \sigma_1 \ b_1 \ \Gamma_1 \ ->
(match b_1 with BNormal False | BFatal -> true | _ -> false envel_term t_3 \ \Gamma_1 \ \sigma_3 \ b_3 \ \Gamma_3 \ ->
eval_term (TIf t_1 \ t_2 \ t_3) \Gamma (concat \sigma_1 \ \sigma_3) b_3 \ \Gamma_3
```

```
| EvalT_If_Transmit : forall t_1 \ \Gamma \ \sigma_1 \ b_1 \ \Gamma_1 \ t_2 \ t_3.
eval_term t_1 \ \Gamma \ \sigma_1 \ b_1 \ \Gamma_1 \ ->
(match b_1 with BReturn _ | BExit _ -> true | _ -> false end) -
eval_term (TIf t_1 \ t_2 \ t_3) \ \Gamma \ \sigma_1 \ b_1 \ \Gamma_1
```

A few rules – Mutual recursion

```
EvalT_AsString : forall s \Gamma \sigma \beta \Gamma, \Gamma, x_s.
eval_sexpr s \Gamma \sigma \beta \Gamma' \rightarrow
\Gamma'' = update_senv \Gamma' x_s \sigma \rightarrow
eval_term (TAsString x_s s) \Gamma empty_string
   (if \beta then BNormal True else BFatal) \Gamma'
```

26 / 61

A few rules – Mutual recursion

```
| EvalT_AsString : forall s \Gamma \sigma \beta \Gamma, \Gamma, x_s.
  eval_sexpr s \Gamma \sigma \beta \Gamma' ->
  \Gamma'' = update_senv \Gamma' x_s \sigma \rightarrow
  eval_term (TAsString x_s s) \Gamma empty_string
      (if \beta then BNormal True else BFatal) \Gamma''
```

```
| EvalSF_Process : forall t \Gamma \sigma b \Gamma'.
  eval_term t \Gamma \sigma b \Gamma' ->
  eval_sfrag_opt (SProcess t) \Gamma~\sigma
     (Some (match b with BNormal True | BReturn True | BExit Tr
     {\Gamma with c_fs = \Gamma'.c_fs ; c_input = \Gamma'.c_input}
```

Table of Contents

1. Language

- Elements of Shell
- CoLiS
- Mechanised version

2. A sound interpreterWhy?

- Let us see some code
- Proof
- An other sound interpreter
- 3. A complete interpreter
 - Which formulation?
 - Heights and sizes
 - Skeletons

向下 イヨト イヨト ヨヨ のくら

• For fun;

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

• For fun;

Helps detecting the potential mistakes;

- We can compare the observational behaviour of our interpreter with
- It gives us a way to test an automated translation from Shell to

- 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

- 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.

↓ ∃ ↓ ∃ | = √ Q ∩

Table of Contents

1. Language

- Elements of Shell
- CoLiS
- Mechanised version

2. A sound interpreter

- Why?
- Let us see some code
- Proof
- An other sound interpreter
- 3. A complete interpreter
 - Which formulation?
 - Heights and sizes
 - Skeletons

▲■ ▶ ▲ 臣 ▶ ▲ 臣 ▶ 三目目 のへの

Spirit of the code

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

30 / 61

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) (\Gamma: context) (previous: bool)
                       : (string, bool, context)
with interp_sfrag_aux (f_s: sfrag) (\Gamma: context) (previous: bool)
                       : (string, bool, context)
. . .
```

▲□▶ ▲□▶ ▲□▶ ▲□▶ ▲□■ のQ@

Body – Sequence and branching

Nicolas Jeannerod

```
let rec interp_term (t: term) (Γ: context)
                     (stdout: ref string) : (bool, context)
=
  match t with
  | TSeq t_1 t_2 ->
    let (, \Gamma_1) = interp_term t_1 \Gamma stdout in
    interp_term t_2 \Gamma_1 stdout
```

VALS Seminar

July 7, 2017

31 / 61

Body – Sequence and branching

```
let rec interp_term (t: term) (Γ: context)
                          (stdout: ref string) : (bool, context)
=
  match t with
  | TSeq t_1 t_2 ->
     let (, \Gamma_1) = interp_term t_1 \Gamma stdout in
     interp_term t_2 \Gamma_1 stdout
   | TIf t_1 t_2 t_3 ->
     let (b_1, \Gamma_1) =
       try
          interp_term t_1 \ \Gamma stdout
       with
          EFatal \Gamma_1 \rightarrow (false, \Gamma_1)
       end
     in
     interp_term (if b_1 then t_2 else t_3) \Gamma_1 stdout
                                                   Nicolas Jeannerod
                                  VALS Seminar
                                                            July 7, 2017
                                                                          31 / 61
```

Body – Mutual recursion

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

Body – Mutual recursion

```
let rec interp_term (t: term) (F: context)
                        (stdout: ref string) : (bool, context)
=
  match t with
  | TAsString xs s ->
    let (\sigma, b, \Gamma') = interp_sexpr s \Gamma in
    let \Gamma'' = update_senv \Gamma' xs \sigma in
    if b then (true, \Gamma'') else raise (EFatal \Gamma'')
  . . .
with interp_sfrag_aux (f_s: sfrag) (\Gamma: context) (previous: bool)
                           : (string, bool, context)
=
  match f_{\rm s} with
  | SProcess t ->
    let (\sigma, b, fs, input) = interp_process t \Gamma in
     (\sigma, b, \{\Gamma \text{ with } c_{fs} = fs; c_{input} = input\})
  . . .
                                                Nicolas Jeannerod
                                VALS Seminar
                                                        July 7, 2017 32 / 61
```

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, \Gamma, \sigma, b and \Gamma': 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

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

Tricolas scatticiou

July 7, 2017
Contract

```
let rec interp_term (t: term) (Γ: context)
                      (stdout: ref string) : (bool, context)
  diverges
  returns { (b, \Gamma') -> exists \sigma.
    !stdout = concat (old !stdout) \sigma
    /\ eval_term t \Gamma \sigma (BNormal b) \Gamma, }
```

Contract

```
let rec interp_term (t: term) (Γ: context)
                         (stdout: ref string) : (bool, context)
  diverges
  returns { (b, \Gamma') -> exists \sigma.
     |stdout = concat (old |stdout) \sigma
     /\ eval_term t \Gamma \sigma (BNormal b) \Gamma, }
  raises { EFatal [', -> exists \sigma].
     !stdout = concat (old !stdout) \sigma
     /\ eval_term t \Gamma \sigma BFatal \Gamma' }
  raises { EReturn (b, \Gamma') -> exists \sigma.
     !stdout = concat (old !stdout) \sigma
     /\ eval_term t \Gamma \sigma (BReturn b) \Gamma, }
  raises { EExit (b, \Gamma') -> exists \sigma.
     !stdout = concat (old !stdout) \sigma
     /\ eval_term t \Gamma \sigma (BExit b) \Gamma, }
                                                                IN THE SOC
                                                                        34 / 61
```

Table of Contents

Language

- Elements of Shell
- CoLiS
- Mechanised version

2. A sound interpreter

- Why?
- Let us see some code

Proof

- An other sound interpreter
- 3. A complete interpreter
 - Which formulation?
 - Heights and sizes
 - Skeletons

④ト (E) (E) (E) (O)

Why it is hard

stdout is a reference.

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch

Why it is hard

stdout is a reference:

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

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch

▲ @ ▶ ▲ ∃ ▶

A ∃ ► ∃ = 1 = 1000

Why it is hard

stdout is a reference:

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

- Usual fix: provide a witness as a ghost return value.
- Cannot work here because of exceptions: we would need to catch

A ∃ ► ∃ = 1 = 1000

Why it is hard

• stdout is a reference:

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

- 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.

↓ ∃ ↓ ∃ | = √ Q ∩

Why it is hard

• stdout is a reference:

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

- 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.

1 = 1 = 1 = 1 = 1

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.

Image: A match a ma

↓ ∃ | = \ < ∩ < ∩</p>

But it works!

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

• • • • •

Table of Contents

1. Language

- Elements of Shell
- CoLiS
- Mechanised version

2. A sound interpreter

- Why?
- Let us see some code
- Proof

• An other sound interpreter

- 3. A complete interpreter
 - Which formulation?
 - Heights and sizes
 - Skeletons

An other sound interpreter

```
let rec interp_term (t: term) (Γ: context)
                       (stdout: ref string) : (bool, context)
  diverges
  returns { (b, \Gamma') -> exists \sigma.
    !stdout = concat (old !stdout) \sigma
    /\ eval_term t \Gamma \sigma (BNormal b) \Gamma, }
  . . .
=
  while true do
   ()
  done
```

Table of Contents

- 1. Language
 - Elements of Shell
 - CoLiS
 - Mechanised version
- 2. A sound interpreter
 - Why?
 - Let us see some code
 - Proof
 - An other sound interpreter

3. A complete interpreterWhich formulation?

- Heights and sizes
- Skeletons

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.

Nicolas Jeannerod

VALS Seminar

July 7, 2017 41 / 61

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.

Nicolas Jeannerod

VALS Seminar

July 7, 2017 41 / 61

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.

Nicolas Jeannerod

VALS Seminar

July 7, 2017 41 / 61

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.

Nicolas Jeannerod

VALS Seminar

3 N 2 1 2 N 0 0

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.

Nicolas Jeannerod

VALS Seminar

July 7, 2017 41 / 61

3 N 2 1 2 N 0 0

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.

Nicolas Jeannerod

VALS Seminar

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,

Nicolas Jeannerod

VALS Seminar

July 7, 2017

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,

Nicolas Jeannerod

VALS Seminar

July 7, 2017

Which formulation?

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 B A B A B B B A A A

Which formulation?

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,

Nicolas Jeannerod

Which formulation?

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*:

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

then:

 $\sigma_1 = \sigma_2$ and $b_1 = b_2$ and $\Gamma_1 = \Gamma_2$

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.

< □ > < 큔 > < 클 > < 클 > · 클 > 클 = July 7, 2017

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

```
| TSeq t_1 t_2 \rightarrow
let (_, \Gamma_1) = interp_term t_1 \Gamma stdout in
interp_term t_2 \Gamma_1 stdout
```

We know that:

$$\exists \sigma b \Gamma''. \quad (t_1 ; t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \land (\exists \sigma' b' \Gamma'. \quad t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \land \quad t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \land \quad b' \in \{ \text{True}, \text{False} \})$$

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

Nicolas Jeannerod

⇒ ⇒ ≡ = • • • • •

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

```
| TSeq t_1 t_2 \rightarrow
let (_, \Gamma_1) = interp_term t_1 \Gamma stdout in
interp_term t_2 \Gamma_1 stdout
```

We know that:

```
 \begin{array}{l} \exists \sigma b \mathsf{\Gamma}''. \quad (t_1 \ ; \ t_2)_{/\mathsf{\Gamma}} \Downarrow \sigma \star b_{/\mathsf{\Gamma}''} \\ \land (\exists \sigma' b' \mathsf{\Gamma}'. \quad t_{1/\mathsf{\Gamma}} \Downarrow \sigma' \star b'_{/\mathsf{\Gamma}'} \\ \land \quad t_{2/\mathsf{\Gamma}'} \Downarrow \sigma \star b_{/\mathsf{\Gamma}''} \\ \land \quad b' \in \{\mathsf{True}, \mathsf{False}\}) \end{array}
```

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

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

```
| TSeq t_1 t_2 \rightarrow
let (_, \Gamma_1) = interp_term t_1 \Gamma stdout in
interp_term t_2 \Gamma_1 stdout
```

We know that:

$$\exists \sigma b \Gamma''. \quad (t_1 ; t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \land (\exists \sigma' b' \Gamma'. \quad t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \land \quad t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \land \quad b' \in \{ \mathsf{True}, \mathsf{False} \})$$

But we need to say that that Γ' is in fact Γ_1 . Hence the need for the soundness and the functionality, Λ_{\pm} ,

Nicolas Jeannerod

VALS Seminar

July 7, 2017

44 / 61

≣। ≡। ≡ • • • •

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

```
| TSeq t_1 t_2 \rightarrow
let (_, \Gamma_1) = interp_term t_1 \Gamma stdout in
interp_term t_2 \Gamma_1 stdout
```

We know that:

$$\exists \sigma b \Gamma''. \quad (t_1 ; t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \land (\exists \sigma' b' \Gamma'. \quad t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \land \quad t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \land \quad b' \in \{ \mathsf{True}, \mathsf{False} \})$$

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

Hence the need for the soundness and the functionality,

≣। ≡। ≡ • • • •

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

```
| TSeq t_1 t_2 \rightarrow
let (_, \Gamma_1) = interp_term t_1 \Gamma stdout in
interp_term t_2 \Gamma_1 stdout
```

We know that:

$$\exists \sigma b \Gamma''. \quad (t_1 ; t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''} \\ \land (\exists \sigma' b' \Gamma'. \quad t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \\ \land \quad t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''} \\ \land \quad b' \in \{ \text{True}, \text{False} \})$$

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 \sigma b \Gamma'. eval_term t \Gamma \sigma b \Gamma' }
  variant { ... }
  returns { (b, \Gamma') -> exists \sigma.
     !stdout = concat (old !stdout) \sigma
     /\ eval_term t \Gamma \sigma (BNormal b) \Gamma' }
```

Nicolas Jeannerod

. . .

What do we need, then?

```
let rec interp_term (t: term) (Γ: context)
                         (stdout: ref string) : (bool, context)
  requires { exists \sigma b \Gamma'. eval_term t \Gamma \sigma b \Gamma' }
  variant { ... }
  returns { (b, \Gamma') -> exists \sigma.
     !stdout = concat (old !stdout) \sigma
     /\ eval_term t \Gamma \sigma (BNormal b) \Gamma' }
   . . .
```

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

Nicolas Jeannerod

Table of Contents

- Elements of Shell
- CoLiS
- Mechanised version
- 2. A sound interpreter
 - Why?
 - Let us see some code
 - Proof
 - An other sound interpreter

3. A complete interpreter

- Which formulation?
- Heights and sizes
- Skeletons

Let us find a variant

Terms are structurally decreasing? Wrong.

(while t_1 do t_2)_{$/\Gamma_2$} $\Downarrow \sigma_3 \star b_{3/\Gamma_3}$ (while t_1 do t_2)_{/ Γ} $\Downarrow \sigma_1 \sigma_2 \sigma_3 \star b_{3/\Gamma_3}$

Let us find a variant

Terms are structurally decreasing? Wrong.

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \quad b_1 = \mathsf{True}}{(\mathsf{while } t_1 \mathsf{ do } t_2)_{/\Gamma_2} \And \sigma_3 \star b_{3/\Gamma_3}} \frac{t_{2/\Gamma} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{(\mathsf{while } t_1 \mathsf{ do } t_2)_{/\Gamma_2} \Downarrow \sigma_3 \star b_{3/\Gamma_3}}$$

Let us find a variant

Terms are structurally decreasing? Wrong.

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

$$\frac{(\text{while } t_1 \text{ do } t_2)_{/\Gamma_2} \Downarrow \sigma_1 \sigma_2 \sigma_3 \star b_{3/\Gamma_3}}{(\text{while } t_1 \text{ do } t_2)_{/\Gamma} \Downarrow \sigma_1 \sigma_2 \sigma_3 \star b_{3/\Gamma_3}}$$

Proofs are structurally decreasing?
Let us find a variant

Terms are structurally decreasing? **Wrong.**

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

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

Let us find a variant

Terms are structurally decreasing? **Wrong.**

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

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?

Heights and sizes

Why it does not work -1

Superposition provers are bad with arithmetic.

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

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).

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).

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.

A ∃ ► ∃ = < < < </p>

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.

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

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.

1 = 1 = 1 = 1 = 1

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.

▲□▶ ▲□▶ ▲三▶ ▲三▶ 三回日 ののの

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.

50 / 61

▲□▶ ▲□▶ ▲三▶ ▲三▶ 三回日 ののの

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.

(日) (周) (日) (日) (日) (日) (000)

Table of Contents

- - Elements of Shell
 - CoLiS
 - Mechanised version
- 2. A sound interpreter
 - Why?
 - Let us see some code
 - Proof
 - An other sound interpreter

3. A complete interpreter

- Which formulation?
- Heights and sizes
- Skeletons

Nicolas Jeannerod

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.

Tricolas scatticiou

A ∃ ► ∃ = 1 = 1000

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.

	Nicol	as .	lean	neroc
--	-------	------	------	-------

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 =
```

```
eval_term t_1 \ \Gamma \ \sigma_1 (BNormal b_1) \Gamma_1 \ sk1 \rightarrow
eval term t_2 \Gamma_1 \sigma_2 b_2 \Gamma_2 \mathbf{sk2} \rightarrow
eval_term (TSeq t_1 t_2) \Gamma (concat \sigma_1 \sigma_2) b_2 \Gamma_2 (S2 sk1 sk2)
```

```
EvalT_Seq_Error : forall t_1 \ \ \sigma_1 \ b_1 \ \ \Gamma_1 \ t_2 \ \ sk.
eval_term t_1 \ \Gamma \ \sigma_1 \ b_1 \ \Gamma_1 \ sk \rightarrow
(match b_1 with BNormal _ -> false | _ -> true end) ->
eval_term (TSeq t_1 t_2) \Gamma \sigma_1 b_1 \Gamma_1 (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, \Gamma') -> exists \sigma.
    !stdout = concat (old !stdout) \sigma
    /\ eval term t \Gamma \sigma (BNormal b) \Gamma' 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
```

<□> <同> <同> < 回> < 回> < 回> < 回> < 回> < 回< の< ○

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

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 t_1 t_2 ->
let ghost sk1 = skeleton12 sk in
let (\_, \Gamma_1) = interp_term t_1 \Gamma stdout sk1 in
let ghost (_, sk2) = skeleton2 sk in
interp_term t_2 \Gamma_1 stdout sk2
```

Put them everywhere – In the code

```
| TSeq t_1 t_2 ->
  let ghost sk1 = skeleton12 sk in
  let (\_, \Gamma_1) = interp_term t_1 \Gamma stdout sk1 in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term t_2 \Gamma_1 stdout sk2
| TIf t_1 t_2 t_3 ->
  let (b_1, \Gamma_1) =
    try
       let ghost sk1 = skeleton12 sk in
       interp_term t_1 \ \Gamma stdout sk1
    with
       EFatal \Gamma' \rightarrow (false, \Gamma')
    end
  in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term (if b_1 then t_2 else t_3) \Gamma_1 stdout sk2
                                                Nicolas Jeannerod
                                 VALS Seminar
                                                          July 7, 2017
                                                                       57 / 61
```

And it's all green!

Interview Statu Time Source code Task Edge 0 Upproved goals V Grain integrate/, minuppeter.minup Nonce code Task Edge Nonce code Nonce code <th><u>File View Jools H</u>elp</th> <th>4</th> <th></th> <th></th> <th></th>	<u>File View Jools H</u> elp	4			
O Unproved goals W Corr intergreem P 29 29 29 M goals All goals All goals P 20 Destination P 20 29 29 P 20 Destination P 20 Desceptional postcondition P 20 Desceptional postcondition	Context	Theories/Goals	Status Time	Source code Task Edited proof	Prover Output Counter-example
Algoals *** setting algoal up 237.99 rategies > 1 postcondian 3.02 *** setting algoal up 237.99 *** setting algoal up 3.02 *** setting algoal up setting algoal up **** setting algoal up setting algoal up ***** setting algoal up setting algoal up ****************	 Unproved goals 		297.99	A37 let rec intern term (*:	file: interpreter//interpreter.mlw
aranges > if 1, bostcondion > 3.02 * Compute > if 2, postcondion > 1.01 * Compute > if a costcondion > 1.01 * Inne > if a costcondion > 1.01 * Inne > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.05 > if a costcondion > 1.01 * Source does > 0.07 > if a costcondion > 1.01 * Source does > 0.07 > if a costcondion > 1.01 * Source does > 0.07 > if a costcondion > 1.01 * Source does > 0.07 > if a costcondion > 0.07 * Source	All goals	▼ 🍞 split_goal_wp	297.99	438 requires { exists s b	g'. eval_term t g s b g' sk }
alwayse > @ 2. postcondition 2.14 > @ 3. exceptional postcondition 2.14 > @ 3. exceptional postcondition 2.14 > @ 1. postcondition 2.14 > @ 1. precondition 2.24 > @ 1. precondition 0.24 > @ 1. precondition 0.27 > @ 1. precondition 0.27 > @ 1. precondition 0.27<	Strategies	I. postcondition	3.02	439 variant { size sk } 440 returns { (b, g')	-> exists s, !stdout = concat (old !stdout) s /\
 Compute I acceptional postcondition <lii acceptional="" li="" postcondition<=""> <</lii>	strategies	2. postcondition	1.91	441 raises { EFatal g'	-> exists s. !stdout = concat (old !stdout) s /\
internal > # + precondition 4.14 > = 5. valuation 0.26 > = 5. valuation 0.26 > = 7. exceptional postcondition 0.13 Vers > # exceptional postcondition 0.13 A # Ergo (1.30) > # exceptional postcondition 0.14 VCV4 (1.5. preendation 0.13 Cod (8.6) > # exceptional postcondition 0.14 Eprover (1.5.1.001) > # 12 precendation 0.04 > # 13. precondition 0.13 Z (4.5.0) > # 2.2 valuation decrease 0.07 > # 13. precondition 0.12 > # 14. exceptional postcondition 0.137 > # 14. exceptional postcondition 0.137 > # 13. precondition 0.137 > # 14. exceptional postcondition 0.147 > # 13. precondition 0.147 > # 14. exceptional postcondition 0.147 > # 14.4 # # # # # # # # # # # # # # # # # # #	2 Compute	3. exceptional postcondition	2.96	442 raises (EReturn (b, 443 raises { EExit (b, g	<pre>(') -> exists s. !stdout = concat (old istdout) s /\ '') -> exists s. !stdout = concat (old !stdout) s /\ </pre>
• Split • S variant decrease • 0.26 • G precondition • 0.3.3.3 • G precondition • 0.3.3 • G precondition • 0.3.4 • G precondition • 1.4 • G precondition	1 Inline	♦] 4. precondition	4.14	444 = 445 match t with	
• Spit:		5. variant decrease	0.26	446 (* Literals *)	
Overs > [] 7 acceptional postcondition 150 ARtErgo (1.30) > [] 6 acceptional postcondition 2.244 > 9 acceptional postcondition 2.244 > 0 acceptional postcondition 2.244 > 10 acceptional postcondition 2.244 > 10 acceptional postcondition 124 > 11 percondition 0.044 > 12 variant decrease 0.07 > 23 (45.01 > [] 12 variant decrease 0.07 > 23 (45.01 > [] 13 precondition 0.13 > [] 14 acceptional postcondition 0.211 > [] 15 exceptional postcondition 0.211 > [] 14 acceptional postcondition 0.211 > [] 15 exceptional postcondition 0.211 > [] 15 exceptional postcondition 0.211 > [] 16 acceptional postcondition 0.211 > [] 16 acceptional postcondition 0.211 > [] 15 exceptional postcondition 0.211 > [] 16 acceptional postc	⊘} ⊢ Split	6. precondition	3.25	447 448 TTrue -> (true, g	0
Aktrog (1.30) > 8 exceptional postcondition 2.54 > 9 0:exceptional postcondition 1.13 > 0:exceptional postcondition 1.13 Cod (8.6) > 11 precondition Spass (3.9) > 1.2 44 Spass (3.9) > 1.3 0 Za (4.5.0) > 1.4 exceptional postcondition 1.23 obs 1.3 precondition 0.237 1.4 exceptional postcondition 1.24 Spass (3.9) 2.3 1.4 exceptional postcondition 1.24 1.4 exceptional postcondition 1.47 Edit 1.5 exceptional postcondition 1.47 1.4 exceptional postcondition 1.47 image: 1.3 precondition 4.00 1.47 image:	rovers	7. exceptional postcondition	2 1.50	449 TFalse -> (false, g 450 TFatal -> raise (EF) atal g)
VCVC (11.5-prerelease) > 9 exceptional postcondition 9 129 Coq (8.6) > 10 exceptional postcondition 9 161 Eprover (1.9.1001) > 21 aprecondition 0 64 Spass (3.9) > 13 aprecondition 9 177 Sta (4.50) > 14 exceptional postcondition 9 177 Ob > 15 exceptional postcondition 9 177 Ob > 15 exceptional postcondition 9 177 Ob > 15 exceptional postcondition 9 177 Ob > 16 exceptional postcondition 9 177 Ob > 18 exceptional postcondition 9 149 Ob > 18 exceptional postcondition 9 149 Ob > 18 exceptional postcondition 9 149 Ob > 19 exceptional postcondition 9 149 Ob > 21 expectional postcondition 1149 Ob <td>Alt-Ergo (1.30)</td> <td>8. exceptional postcondition</td> <td>2.54</td> <td>451 (* Poturo Exit *)</td> <td></td>	Alt-Ergo (1.30)	8. exceptional postcondition	2.54	451 (* Poturo Exit *)	
Vicolas > 10. exceptional postcondition > 161 • 10. exceptional postcondition • 161 • 11. precondition • 064 • 21.4.1 • 0.64 • 21.4.1 • 0.64 • 21.4.1 • 0.64 • 21.4.1 • 0.64 • 21.4.1 • 0.64 • 21.4.1 • 0.64 • 21.4.1 • 0.64 • 11. precondition • 0.07 • 11. precondition • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1 • 0.07 • 11.4.1.1 • 0.07 • 11.	CVC4 (1 5-prerelease)	9. exceptional postcondition	2 1.29	453	
Cog (8.6) iii preconduon iiii iii iiii iiiiiiiiii	CVC4 (1.5-prefelease)	10. exceptional postcondition	2 1.61	454 TReturn t -> 455 let ghost sk' = ske	letonl sk in
Eprover (1.9.1.001) 	Coq (8.6)	v 🗾 11 precondition	0.64	456 raise (EReturn (int	erp_term t g stdout sk'))
Space (3.9) > 12 variant decrease 0.07 Z3 (4.5.0) > 3.0 precondition 0.157 23 (4.5.0) > 3.0 precondition 0.157 > 13. seceptional postcondition 0.147 > 15. seceptional postcondition 0.147 > 16 > 13. seceptional postcondition 0.147 > 13. seceptional postcondition 0.147 > 14. seceptional postcondition 0.147 > 15. seceptional postcondition 0.147 > 16. 10. seceptional postcondition 0.147 > 13. precondition 0.26 > 10. precondition 0.26 > 10. precondition 0.26 > 10. precondition 0.26 > 13. precondition 0.26 > 14. seceptional postcondition 0.147 > 16. 21. postcondition 0.26 > 17. seceptional postcondition 0.26 > 19. precondition 0.26 > 10. precondition 0.26 > 10. precondition 0.21 > 10. precondition 0.22 > 10. precondition 0.22 > 10. precondition 0.22 > 10. precondition	Eprover (1.9.1-001)	× 73 (4.4.1)	0.64	457 458 TExit t ->	
splas 13.97 > [] 13. precondition > 3.27 23 (43.0) > [] 14. exceptional postcondition > 1.57 0s > [] 15. exceptional postcondition > 1.17 16 [] 16. exceptional postcondition > 1.14 17. Strings 1, g) != inter_preser s g sk' in it for inter graver g sk' in it for inter grave	(2.0)	b 12 variant decrease	0.07	459 let ghost sk' = ske 460 raise (EExit (inter	(letonl sk in p term t g stdout sk'))
23 (4.5.0) 	2base (2:2)	b 13 precondition	3 27	461	
obs 	Z3 (4.5.0)	b / 14 exceptional portrondition	1 57	463	
Edit > 2.5 Seeptional postcondition -1.11 >> 1.6 exceptional postcondition -1.14 >> 1.6 exceptional postcondition -1.14 >> 1.6 exceptional postcondition -1.14 >> 1.7 exceptional postcondition -1.14 >> 1.8 precondition -1.14 >> 1.8 precondition -1.14 >> 1.9 precondition -1.14 >> 1.0 precondition -1.14 >> 1.0 precondition -1.14 >> 1.0 precondition -1.14 >> 1.0 precondition -1.01 >> 1.0 precondition -1.01 >> 1.0 precondition -1.14 >> 1.0 precondition -1.14 >> 1.0 precondition -1.14 >> 1.0 precondition -1.14 >> 2.0 precondition -1.14 >> 2.2 precondition -1.14 >> 2.2 precondition -1.14 >> 2.4 variant decrease -0.22 >> 2.4 variant decrease <td< td=""><td>ols</td><td>h 7 15 exceptional postcondition</td><td>2.37</td><td>464 TAsString xs s -> 465 let ghost sk' = ske</td><td>leton1 sk in</td></td<>	ols	h 7 15 exceptional postcondition	2.37	464 TAsString xs s -> 465 let ghost sk' = ske	leton1 sk in
	/ Edit	b 7 16 exceptional postcondition	2.11	466 let (sigma, b, g')	= interp_sexpr s g sk' in
We play > 1.1 <td< td=""><td>-</td><td>b 7 17 exceptional postcondition</td><td>1.40</td><td>468 if b then</td><td>nv g xs signa in</td></td<>	-	b 7 17 exceptional postcondition	1.40	468 if b then	nv g xs signa in
	Replay	V 17. exceptional postcondition	1.47	(true, g'') 470 else	
→ →	C Remove	P _ 18. precondition	4.00	: 471 raise (EFatal g'') (* It is really g''! *)
Image: Claim of the state	-	P _ 19. variant decrease	0.26	472 473 TAsList x_ll ->	
off monotring > 21. postcondition 4.91 176 It et g'' update_lenv g' x_l^lambda in the the g'' g'' list is really g'' (t) Wating: 0 > 22. exceptional postcondition 1.16 477 if b then the g''' g'' list is really g'' (t) Scheduled: 0 > 23. precondition 4.01 477 if b then then then g''' g'' list is really g'' (t) Sinterrupt > 25. precondition 3.22 483 (f Sequence t) > 26. postcondition 3.22 483 (f Sequence t) > 27. exceptional postcondition 1.01 485 It et g'' to the skeletan12 sk in the tet gt (t) to the skeletan12 sk in the tet gt (t) to the skeletan12 sk in the skeletan12 sk in the tet gt (t) to the skeletan12 sk in	(III) Clean	P _ 20. precondition	3.20	474 let ghost sk' = ske 475 let (lambda, b, g')	leton1 sk in = interp lexpr l a sk' in
Watting: 0 > 22: exceptional postcondution 1.16 11: (1: 100 rg·) Scheduled: 0 > 23: precondution 401 175 41: (1: 100 rg·) Rinning: 0 > 23: precondution 401 475 41: (1: 100 rg·) Pinterupt > 24: variant decrease 0.22 430 416 416 Pinterupt > 25: precondution 0.32 434 (* Sequence) 416 444 (* Sequence) Pinterupt > 26: precondution 5.54 444 (* Sequence) 10: 10: 10: 10: 10: 10: 10: 10: 10: 10:	roof monitoring	P 21. postcondition	4.91	476 let g'' = update_le	nv gʻx_l lambda in
Burning:0 > 23. precondition 410 479 else Winning:0 > 24. variant decrease 0.22 400 raise ([fatal g'') (* it is really g'' *) > 25. precondition 0.22 400 (* Sequence *) > 25. precondition 0.22 400 (* sequence *) > 25. precondition 0.22 400 (* sequence *) > 25. precondition 0.10 484 Tequet 12 -> 27. exceptional postcondition 1.01 485 et ghost 1, si = skeleton12 sk in 485 et ghost 1, si = skeleton2 sk in et ghost 1, si = skeleton2 sk in et ghost 1, si = skeleton2 sk in 10 2.8 2.8 et ghost 1, si = skeleton2 sk in et ghost 1, si = skeleton2 sk in 10 2.8 et ghost 1, si = skeleton2 sk in et ghost 1, si = skeleton2 sk in	Waiting: 0 Scheduled: 0	22. exceptional postcondition	1.16	477 11 D Then 478 (true, g'')	
Interrupt ▶ 24. variant decrease ⊘ 0.22 482 (*) 6 6 6 7 6 6 7 6 7 6 7 6 7 6 7 7 7	Running: 0	23. precondition	4.01	479 else 480 raise (EFatal g'') (* it is really g''! *)
b 15 15 15 15 16 101 16 16 101 16	🙁 Interrupt	24. variant decrease	0.22	481	
▷ ☐ 26. postcondition ○ 5.54 #44 Tseq tit 2 · · · ▶ ☐ 27. exceptional postcondition ○ 1.01 #86 tet ghost (, c) = interp.term tig gatout ski in		25. precondition	3.22	402 (+ sequence *) 483	
▶ ☐ 77. exceptional postcondition Q 1.01 ##6 let f, all interp.tera 1/2 attack ▶ ☐ 72. exceptional postcondition Q 2.07 ##7 let diabat () = interp.tera 1/2 attack Nicolas Jeannerod VALS Seminar VALS Seminar July 7, 2017 58 /		26. postcondition	5.54	484 TSeq t1 t2 ->	leton12 sk in
► ► ► ► ► ► ► ► ► ► ► ► ► ► ► ► ►		27. exceptional postcondition	1.01	486 let (_, g1) = inter	p_term tl g stdout skl in
Nicolas Jeannerod VALS Seminar July 7, 2017 58 /		28. precondition	2.87	487 let ghost (_, sk2)	= skeletonz sk in +dau+ ek2
	Nicolas	Jeannerod	VALS	Seminar	July 7, 2017 58 /

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.

↓ ∃ ↓ ∃ | = √ Q ∩

59 / 61

Image: A match a ma

And it's all green!

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

< 4 → <

A ∃ ► ∃ = 1 = 1000

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.

JE SOO

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