

# Formalising an intermediate language for POSIX shell

Nicolas Jeannerod



Séminaire Gallium, Septembre 18, 2017



Shell

# Big picture

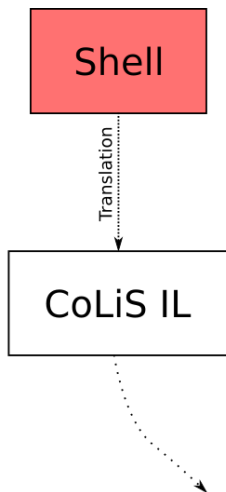
Shell

```
graph TD; Shell[Shell]; CoLiS_IL[CoLiS IL]; Formal_methods[Formal methods]; CoLiS_IL -.-> Formal_methods;
```

CoLiS IL

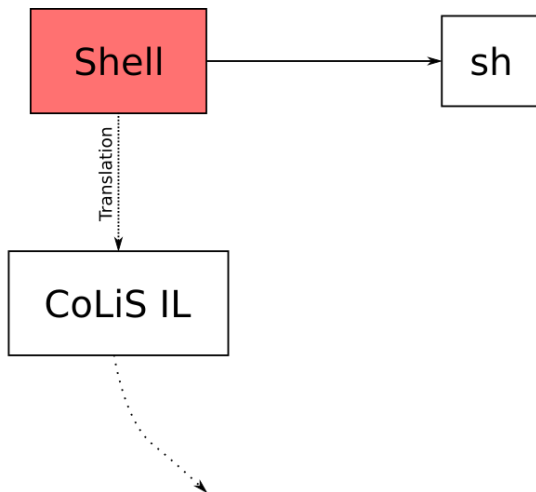
Formal methods

# Big picture



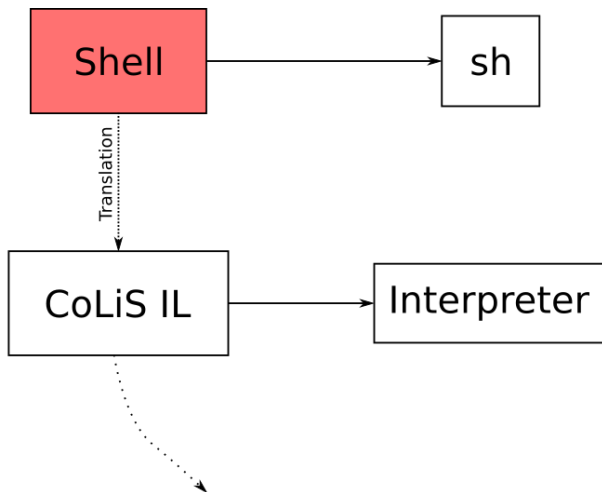
Formal methods

# Big picture



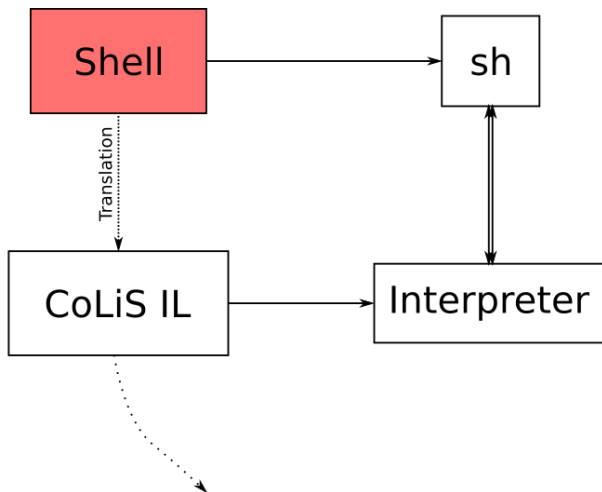
Formal methods

# Big picture



Formal methods

# Big picture



Formal methods

# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- Expansion
- Inconstant semantics
- Control flow

## 2. The CoLiS language

- Requirements
- Definitions

## 3. Formalisation

- Formulation
- Proof



# Execute arbitrary strings

Execute commands from strings:

```
a='echo foo'  
$a          ## prints "foo"
```

or any code with `eval`:

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

# Execute arbitrary strings

Execute commands from strings:

```
a='echo foo'  
$a          ## prints "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           ## prints "bar"
```

I tell ya, everything!

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

# Dynamic

Everything is dynamic:

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

I tell ya, everything!

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

# Dynamic

Everything is dynamic:

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

I tell ya, everything!

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

# Dynamic

Everything is dynamic:

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

I tell ya, everything!

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

# Dynamic

Everything is dynamic:

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

I tell ya, everything!

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

# Dynamic

Everything is dynamic:

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

I tell ya, everything!

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



# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- **Expansion**
- Inconstant semantics
- Control flow

## 2. The CoLiS language

- Requirements
- Definitions

## 3. Formalisation

- Formulation
- Proof

# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)
- Special parameters
- “Formatted” parameters
- Arithmetic
- Globs
- Commands
- Quotes

# All it can contain

- Literals
- Tildes

```
~/Pictures      ~user/Pictures:~/Download
```

- Parameters (*i.e. variables*)
- Special parameters
- “Formatted” parameters
- Arithmetic
- Globs
- Commands
- Quotes

# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)

```
$foo
```

```
$bar
```

- Special parameters
- “Formatted” parameters
- Arithmetic
- Globs
- Commands
- Quotes

# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)
- Special parameters

`$@``$*``$1, $2, ...`

- “Formatted” parameters
- Arithmetic
- Globs
- Commands
- Quotes

# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)
- Special parameters
- “Formatted” parameters

```
`${foo:-bar}`    `${foo-baz}`  
`${foo%.*}`     `${foo###*/}`
```

- Arithmetic
- Globs
- Commands
- Quotes

# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)
- Special parameters
- “Formatted” parameters
- Arithmetic

```
$( (1 + x + $x) )
```

- Globs
- Commands
- Quotes

# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)
- Special parameters
- “Formatted” parameters
- Arithmetic
- Globs

```
/home/[!a]* *.ml *.ml?
```

- Commands
- Quotes



# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)
- Special parameters
- “Formatted” parameters
- Arithmetic
- Globs
- Commands

```
$(echo foo)
`echo \ `echo foo\ ``
$(which curl)
```

- Quotes

# All it can contain

- Literals
- Tildes
- Parameters (*i.e. variables*)
- Special parameters
- “Formatted” parameters
- Arithmetic
- Globs
- Commands
- Quotes

```
foo='my file'  
rm $foo '$foo' "$foo"
```

## Dirty uses

Abused to represent both strings and lists of strings:

```
path='/home'
path="$path/nicolas"      ## "/home/nicolas"
args='-l -a'
args="$args -h"          ## ["-l"; "-a"; "-h"]
ls $args $path
```

Or lists separated by something else than space:

```
PATH='/usr/local/bin:/usr/bin:/bin'
IFS=:
for dir in $PATH; do
    echo $dir
done
```

## Dirty uses

Abused to represent both strings and lists of strings:

```
path='/home'  
path="$path/nicolas"      ## "/home/nicolas"  
args='-l -a'  
args="$args -h"          ## ["-l"; "-a"; "-h"]  
ls $args $path
```

Or lists separated by something else than space:

```
PATH='/usr/local/bin:/usr/bin:/bin'  
IFS=:  
for dir in $PATH; do  
    echo $dir  
done
```

## Dirty uses

Abused to represent both strings and lists of strings:

```
path='/home'  
path="$path/nicolas"      ## "/home/nicolas"  
args='-l -a'  
args="$args -h"          ## ["-l"; "-a"; "-h"]  
ls $args $path
```

Or lists separated by something else than space:

```
PATH='/usr/local/bin:/usr/bin:/bin'  
IFS=:  
for dir in $PATH; do  
    echo $dir  
done
```

## Dirty uses

Abused to represent both strings and lists of strings:

```
path='/home'  
path="$path/nicolas"      ## "/home/nicolas"  
args='-l -a'  
args="$args -h"          ## ["-l"; "-a"; "-h"]  
ls $args $path
```

Or lists separated by something else than space:

```
PATH='/usr/local/bin:/usr/bin:/bin'  
IFS=:  
for dir in $PATH; do  
    echo $dir  
done
```

## Dirty uses

Abused to represent both strings and lists of strings:

```
path='/home'  
path="$path/nicolas"      ## "/home/nicolas"  
args='-l -a'  
args="$args -h"          ## ["-l"; "-a"; "-h"]  
ls $args $path
```

Or lists separated by something else than space:

```
PATH='/usr/local/bin:/usr/bin:/bin'  
IFS=:  
for dir in $PATH; do  
    echo $dir  
done
```

## Dirty uses

Abused to represent both strings and lists of strings:

```
path='/home'  
path="$path/nicolas"      ## "/home/nicolas"  
args='-l -a'  
args="$args -h"          ## ["-l"; "-a"; "-h"]  
ls $args $path
```

Or lists separated by something else than space:

```
PATH='/usr/local/bin:/usr/bin:/bin'  
IFS=:  
for dir in $PATH; do  
    echo $dir  
done
```



# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- Expansion
- **Inconstant semantics**
- Control flow

## 2. The CoLiS language

- Requirements
- Definitions

## 3. Formalisation

- Formulation
- Proof

# Dynamic changes in the semantics: IFS

```
file='git-sucks'  
rm -r $file      ## deletes "git-sucks"  
IFS=-  
rm -r $file      ## deletes "git" and "sucks"
```

Here is what happens:

- 1 The parsing gives us ["rm"; "-r"; "\$file"];
- 2 We apply *parameter expansion* and get ["rm"; "-r"; "git-sucks"];
- 3 We apply *field splitting*, **but only** where we just applied the parameter expansion: [{"rm"}; [{"-r"}]; [{"git"}; {"sucks"}]];
- 4 We flatten everything: ["rm"; "-r"; "git"; "sucks"];
- 5 We evaluate that so-called simple command.

# Dynamic changes in the semantics: IFS

```
file='git-sucks'  
rm -r $file      ## deletes "git-sucks"  
IFS=-  
rm -r $file      ## deletes "git" and "sucks"
```

Here is what happens:

- 1 The parsing gives us ["rm"; "-r"; "\$file"];
- 2 We apply *parameter expansion* and get ["rm"; "-r"; "git-sucks"];
- 3 We apply *field splitting*, **but only** where we just applied the parameter expansion: ["rm"]; ["-r"]; ["git"; "sucks"];
- 4 We flatten everything: ["rm"; "-r"; "git"; "sucks"];
- 5 We evaluate that so-called simple command.

# Dynamic changes in the semantics: IFS

```
file='git-sucks'  
rm -r $file      ## deletes "git-sucks"  
IFS=-  
rm -r $file      ## deletes "git" and "sucks"
```

Here is what happens:

- 1 The parsing gives us ["rm"; "-r"; "\$file"];
- 2 We apply *parameter expansion* and get ["rm"; "-r"; "git-sucks"];
- 3 We apply *field splitting*, **but only** where we just applied the parameter expansion: [{"rm"}; [{"-r"}]; [{"git"}; {"sucks"}];
- 4 We flatten everything: ["rm"; "-r"; "git"; "sucks"];
- 5 We evaluate that so-called simple command.

# Dynamic changes in the semantics: IFS

```
file='git-sucks'  
rm -r $file      ## deletes "git-sucks"  
IFS=-  
rm -r $file      ## deletes "git" and "sucks"
```

Here is what happens:

- 1 The parsing gives us ["rm"; "-r"; "\$file"];
- 2 We apply *parameter expansion* and get ["rm"; "-r"; "git-sucks"];
- 3 We apply *field splitting*, **but only** where we just applied the parameter expansion: ["rm"]; ["-r"]; ["git"; "sucks"];
- 4 We flatten everything: ["rm"; "-r"; "git"; "sucks"];
- 5 We evaluate that so-called simple command.

# Dynamic changes in the semantics: IFS

```
file='git-sucks'  
rm -r $file      ## deletes "git-sucks"  
IFS=-  
rm -r $file      ## deletes "git" and "sucks"
```

Here is what happens:

- 1 The parsing gives us ["rm"; "-r"; "\$file"];
- 2 We apply *parameter expansion* and get ["rm"; "-r"; "git-sucks"];
- 3 We apply *field splitting*, **but only** where we just applied the parameter expansion: ["rm"]; ["-r"]; ["git"; "sucks"];
- 4 We flatten everything: ["rm"; "-r"; "git"; "sucks"];
- 5 We evaluate that so-called simple command.

# Dynamic changes in the semantics: IFS

```
file='git-sucks'  
rm -r $file      ## deletes "git-sucks"  
IFS=-  
rm -r $file      ## deletes "git" and "sucks"
```

Here is what happens:

- 1 The parsing gives us ["rm"; "-r"; "\$file"];
- 2 We apply *parameter expansion* and get ["rm"; "-r"; "git-sucks"];
- 3 We apply *field splitting*, **but only** where we just applied the parameter expansion: ["rm"]; ["-r"]; ["git"; "sucks"];
- 4 We flatten everything: ["rm"; "-r"; "git"; "sucks"];
- 5 We evaluate that so-called simple command.

# Dynamic changes in the semantics: set

With `set`:

- a Every assignment becomes an `export`;
- C `>` no longer overwrite existing files. `>|` still does;
- e The shell shall exit immediately when a command fails, when this failure is not caught;
- f Disables pathname expansion;
- u The shell shall fail when expanding parameters that are unset.

It makes you wonder why most of these options are *disabled* by default.



## Dynamic changes in the semantics: set

With `set`:

- a Every assignment becomes an `export`;
- C `>` no longer overwrite existing files. `>|` still does;

```
echo foo > file
set -C
echo bar > file      ## fails
echo baz >| file     ## succeeds
```

- e The shell shall exit immediately when a command fails, when this failure is not caught;
- f Disables pathname expansion;
- u The shell shall fail when expanding parameters that are unset.

It makes you wonder why most of these options are *disabled* by default.

## Dynamic changes in the semantics: set

With `set`:

- a Every assignment becomes an `export`;
- C > no longer overwrite existing files. >| still does;
- e The shell shall exit immediately when a command fails, when this failure is not caught;

```
set -e
! true ; echo foo    ## prints "foo"
false ; echo foo    ## exists
```

- f Disables pathname expansion;
- u The shell shall fail when expanding parameters that are unset.

It makes you wonder why most of these options are *disabled* by default.

## Dynamic changes in the semantics: set

With `set`:

- a Every assignment becomes an `export`;
- C `>` no longer overwrite existing files. `>|` still does;
- e The shell shall exit immediately when a command fails, when this failure is not caught;
- f Disables pathname expansion;

```
echo *      ## prints the files in $PWD
set -f
echo *      ## prints "*"

```

- u The shell shall fail when expanding parameters that are unset.

It makes you wonder why most of these options are *disabled* by default.

## Dynamic changes in the semantics: set

With `set`:

- a Every assignment becomes an `export`;
- C > no longer overwrite existing files. >| still does;
- e The shell shall exit immediately when a command fails, when this failure is not caught;
- f Disables pathname expansion;
- u The shell shall fail when expanding parameters that are unset.

```
rm -rf "$dir"/      ## deletes everything
set -u
rm -rf "$dir"/      ## fails
```

It makes you wonder why most of these options are *disabled* by default.

## Dynamic changes in the semantics: set

With `set`:

- a Every assignment becomes an `export`;
- C `>` no longer overwrite existing files. `>|` still does;
- e The shell shall exit immediately when a command fails, when this failure is not caught;
- f Disables pathname expansion;
- u The shell shall fail when expanding parameters that are unset.

It makes you wonder why most of these options are *disabled* by default.

# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- Expansion
- Inconstant semantics
- **Control flow**

## 2. The CoLiS language

- Requirements
- Definitions

## 3. Formalisation

- Formulation
- Proof

# Behaviours

Let us play with `exit`:

```
exit | echo 'foo'      ## prints "foo"  
exit || echo 'foo'    ## exits  
exit & echo 'foo'     ## prints "foo"  
exit && echo 'foo'    ## exits  
  
echo 'foo' | exit     ## does nothing  
echo 'foo' || exit   ## prints "foo"  
echo 'foo' & exit    ## prints "foo" and exits  
echo 'foo' && exit   ## prints "foo" and exits
```

# Behaviours

Let us play with `exit`:

```
exit | echo 'foo'      ## prints "foo"  
exit || echo 'foo'    ## exits  
exit & echo 'foo'     ## prints "foo"  
exit && echo 'foo'    ## exits  
  
echo 'foo' | exit     ## does nothing  
echo 'foo' || exit   ## prints "foo"  
echo 'foo' & exit    ## prints "foo" and exits  
echo 'foo' && exit   ## prints "foo" and exits
```



# Behaviours

Let us play with `exit`:

```
exit | echo 'foo'      ## prints "foo"
exit || echo 'foo'     ## exits
exit & echo 'foo'      ## prints "foo"
exit && echo 'foo'     ## exits

echo 'foo' | exit      ## does nothing
echo 'foo' || exit     ## prints "foo"
echo 'foo' & exit      ## prints "foo" and exits
echo 'foo' && exit      ## prints "foo" and exits
```

# Behaviours

Let us play with `exit`:

```
exit | echo 'foo'      ## prints "foo"
exit || echo 'foo'    ## exits
exit & echo 'foo'     ## prints "foo"
exit && echo 'foo'    ## exits

echo 'foo' | exit     ## does nothing
echo 'foo' || exit   ## prints "foo"
echo 'foo' & exit    ## prints "foo" and exits
echo 'foo' && exit   ## prints "foo" and exits
```

# The incredible story of set -e

*When this option is on, when any command fails, the shell immediately shall exit, as if by executing the exit special built-in utility with no arguments, with the following exceptions: [...]*

Snippet 1:

```
false; echo 'foo'
```

Snippet 2:

```
{ false; echo 'foo'; } && echo 'bar'
```

Snippet 3:

```
{ false; echo 'foo'; } | echo 'bar'
```

# The incredible story of set -e

*When this option is on, when any command fails, the shell immediately shall exit, as if by executing the exit special built-in utility with no arguments, with the following exceptions: [...]*

## Snippet 1:

```
false; echo 'foo'
```

## Snippet 2:

```
{ false; echo 'foo'; } && echo 'bar'
```

## Snippet 3:

```
{ false; echo 'foo'; } | echo 'bar'
```

# The incredible story of set -e

*When this option is on, when any command fails, the shell immediately shall exit, as if by executing the exit special built-in utility with no arguments, with the following exceptions: [...]*

## Snippet 1 (**exits**):

```
false; echo 'foo'
```

## Snippet 2:

```
{ false; echo 'foo'; } && echo 'bar'
```

## Snippet 3:

```
{ false; echo 'foo'; } | echo 'bar'
```

# The incredible story of set -e

*When this option is on, when any command fails, the shell immediately shall exit, as if by executing the exit special built-in utility with no arguments, with the following exceptions: [...]*

## Snippet 1 (**exits**):

```
false; echo 'foo'
```

## Snippet 2:

```
{ false; echo 'foo'; } && echo 'bar'
```

## Snippet 3:

```
{ false; echo 'foo'; } | echo 'bar'
```

# The incredible story of set -e

*When this option is on, when any command fails, the shell immediately shall exit, as if by executing the exit special built-in utility with no arguments, with the following exceptions: [...]*

Snippet 1 (**exits**):

```
false; echo 'foo'
```

Snippet 2 (**prints "foo bar"**):

```
{ false; echo 'foo'; } && echo 'bar'
```

Snippet 3:

```
{ false; echo 'foo'; } | echo 'bar'
```

# The incredible story of set -e

*When this option is on, when any command fails, the shell immediately shall exit, as if by executing the exit special built-in utility with no arguments, with the following exceptions: [...]*

Snippet 1 (**exits**):

```
false; echo 'foo'
```

Snippet 2 (**prints "foo bar"**):

```
{ false; echo 'foo'; } && echo 'bar'
```

Snippet 3:

```
{ false; echo 'foo'; } | echo 'bar'
```



# The incredible story of set -e

*When this option is on, when any command fails, the shell immediately shall exit, as if by executing the exit special built-in utility with no arguments, with the following exceptions: [...]*

Snippet 1 (**exits**):

```
false; echo 'foo'
```

Snippet 2 (**prints "foo bar"**):

```
{ false; echo 'foo'; } && echo 'bar'
```

Snippet 3 (**prints "bar"**):

```
{ false; echo 'foo'; } | echo 'bar'
```

# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- Expansion
- Inconstant semantics
- Control flow

## 2. The CoLiS language

- Requirements
- Definitions

## 3. Formalisation

- Formulation
- Proof

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.



# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Requirements

- Intermediate language for a subset of shell;
- Not a replacement of shell;
- 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:
  - We must be convinced that it shares the same semantics as the shell,
  - Target of an automated translation from shell.

# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- Expansion
- Inconstant semantics
- Control flow

## 2. The CoLiS language

- Requirements
- **Definitions**

## 3. Formalisation

- Formulation
- Proof

# Syntax

Programs	$p ::= vdecl^* pdecl^* \mathbf{program} \ t$
Variables decl.	$vdecl ::= \mathbf{varstring} \ x_S \mid \mathbf{varlist} \ x_I$
Procedures decl.	$pdecl ::= \mathbf{proc} \ c \ \mathbf{is} \ t$
Terms	$t ::=$ <ul style="list-style-type: none"> <li><math>\mathbf{true} \mid \mathbf{false} \mid \mathbf{fatal}</math></li> <li><math>\mid \mathbf{return} \ t \mid \mathbf{exit} \ t</math></li> <li><math>\mid x_S := s \mid x_I := /</math></li> <li><math>\mid t ; t \mid \mathbf{if} \ t \ \mathbf{then} \ t \ \mathbf{else} \ t</math></li> <li><math>\mid \mathbf{for} \ x_S \ \mathbf{in} \ / \ \mathbf{do} \ t \mid \mathbf{while} \ t \ \mathbf{do} \ t</math></li> <li><math>\mid \mathbf{process} \ t \mid \mathbf{pipe} \ t \ \mathbf{into} \ t</math></li> <li><math>\mid \mathbf{call} \ / \mid \mathbf{shift}</math></li> </ul>

# Syntax

Programs	$p ::= vdecl^* pdecl^* \mathbf{program} \ t$
Variables decl.	$vdecl ::= \mathbf{varstring} \ x_S \mid \mathbf{varlist} \ x_I$
Procedures decl.	$pdecl ::= \mathbf{proc} \ c \ \mathbf{is} \ t$
Terms	$t ::=$ <ul style="list-style-type: none"> <li><math>\mathbf{true} \mid \mathbf{false} \mid \mathbf{fatal}</math></li> <li><math>\mid \mathbf{return} \ t \mid \mathbf{exit} \ t</math></li> <li><math>\mid x_S := s \mid x_I := /</math></li> <li><math>\mid t ; t \mid \mathbf{if} \ t \ \mathbf{then} \ t \ \mathbf{else} \ t</math></li> <li><math>\mid \mathbf{for} \ x_S \ \mathbf{in} \ / \ \mathbf{do} \ t \mid \mathbf{while} \ t \ \mathbf{do} \ t</math></li> <li><math>\mid \mathbf{process} \ t \mid \mathbf{pipe} \ t \ \mathbf{into} \ t</math></li> <li><math>\mid \mathbf{call} \ / \mid \mathbf{shift}</math></li> </ul>

# Syntax

Programs	$p ::= vdecl^* pdecl^* \mathbf{program} \ t$
Variables decl.	$vdecl ::= \mathbf{varstring} \ x_S \mid \mathbf{varlist} \ x_I$
Procedures decl.	$pdecl ::= \mathbf{proc} \ c \ \mathbf{is} \ t$
Terms	$  \begin{aligned}  t ::= & \mathbf{true} \mid \mathbf{false} \mid \mathbf{fatal} \\  & \mid \mathbf{return} \ t \mid \mathbf{exit} \ t \\  & \mid x_S := s \mid x_I := l \\  & \mid t ; t \mid \mathbf{if} \ t \ \mathbf{then} \ t \ \mathbf{else} \ t \\  & \mid \mathbf{for} \ x_S \ \mathbf{in} \ l \ \mathbf{do} \ t \mid \mathbf{while} \ t \ \mathbf{do} \ t \\  & \mid \mathbf{process} \ t \mid \mathbf{pipe} \ t \ \mathbf{into} \ t \\  & \mid \mathbf{call} \ l \mid \mathbf{shift}  \end{aligned}  $

# Syntax

Programs	$p ::= vdecl^* pdecl^* \mathbf{program} \ t$
Variables decl.	$vdecl ::= \mathbf{varstring} \ x_S \mid \mathbf{varlist} \ x_I$
Procedures decl.	$pdecl ::= \mathbf{proc} \ c \ \mathbf{is} \ t$
Terms	$  \begin{aligned}  t ::= & \mathbf{true} \mid \mathbf{false} \mid \mathbf{fatal} \\  & \mid \mathbf{return} \ t \mid \mathbf{exit} \ t \\  & \mid x_S := s \mid x_I := l \\  & \mid t ; t \mid \mathbf{if} \ t \ \mathbf{then} \ t \ \mathbf{else} \ t \\  & \mid \mathbf{for} \ x_S \ \mathbf{in} \ l \ \mathbf{do} \ t \mid \mathbf{while} \ t \ \mathbf{do} \ t \\  & \mid \mathbf{process} \ t \mid \mathbf{pipe} \ t \ \mathbf{into} \ t \\  & \mid \mathbf{call} \ l \mid \mathbf{shift}  \end{aligned}  $



# Syntax

Terms  $t ::=$  **true** | **false** | **fatal**  
 | **return**  $t$  | **exit**  $t$   
 |  $x_S := s$  |  $x_I := l$   
 |  $t; t$  | **if**  $t$  **then**  $t$  **else**  $t$   
 | **for**  $x_S$  **in**  $l$  **do**  $t$  | **while**  $t$  **do**  $t$   
 | **process**  $t$  | **pipe**  $t$  **into**  $t$   
 | **call**  $l$  | **shift**

String expressions  $s ::=$  **nil** <sub>$S$</sub>  |  $f_S :: s$

String fragments  $f_S ::=$   $\sigma$  |  $x_S$  |  $n$  |  $t$

List expressions  $l ::=$  **nil** <sub>$I$</sub>  |  $f_I :: l$

List fragments  $f_I ::=$   $s$  | **split**  $s$  |  $x_I$

# Syntax

Terms  $t ::=$  **true** | **false** | **fatal**  
 | **return**  $t$  | **exit**  $t$   
 |  $x_S := s$  |  $x_I := l$   
 |  $t; t$  | **if**  $t$  **then**  $t$  **else**  $t$   
 | **for**  $x_S$  **in**  $l$  **do**  $t$  | **while**  $t$  **do**  $t$   
 | **process**  $t$  | **pipe**  $t$  **into**  $t$   
 | **call**  $l$  | **shift**

String expressions  $s ::=$  **nil** <sub>$S$</sub>  |  $f_S :: s$

String fragments  $f_S ::=$   $\sigma$  |  $x_S$  |  $n$  |  $t$

List expressions  $l ::=$  **nil** <sub>$I$</sub>  |  $f_I :: l$

List fragments  $f_I ::=$   $s$  | **split**  $s$  |  $x_I$

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).



# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic judgements

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

A *context*  $\Gamma$  contains:

- flags?
- a file system,
- the standard input,
- the arguments line,
- environments for string and list variables,
- an environment for procedures.

A *behaviour*  $b$  can be

True, False, Fatal, Return (True|False) or Exit (True|False).

# Semantic rules – Branching

BRANCHING-TRUE

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 = \text{True} \quad t_2/\Gamma_1 \Downarrow \sigma_2 * b_2/\Gamma_2}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1\sigma_2 * b_2/\Gamma_2}$$

BRANCHING-FALSE

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 \in \{\text{False}, \text{Fatal}\} \quad t_3/\Gamma_1 \Downarrow \sigma_3 * b_3/\Gamma_3}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1\sigma_3 * b_3/\Gamma_3}$$

BRANCHING-EXCEPTION

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 \in \{\text{Return } \_, \text{Exit } \_ \}}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1}$$

# Semantic rules – Branching

BRANCHING-TRUE

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 = \text{True} \quad t_2/\Gamma_1 \Downarrow \sigma_2 * b_2/\Gamma_2}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1\sigma_2 * b_2/\Gamma_2}$$

BRANCHING-FALSE

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 \in \{\text{False}, \text{Fatal}\} \quad t_3/\Gamma_1 \Downarrow \sigma_3 * b_3/\Gamma_3}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1\sigma_3 * b_3/\Gamma_3}$$

BRANCHING-EXCEPTION

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 \in \{\text{Return } \_, \text{Exit } \_ \}}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1}$$

# Semantic rules – Branching

BRANCHING-TRUE

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 = \text{True} \quad t_2/\Gamma_1 \Downarrow \sigma_2 * b_2/\Gamma_2}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1\sigma_2 * b_2/\Gamma_2}$$

BRANCHING-FALSE

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 \in \{\text{False}, \text{Fatal}\} \quad t_3/\Gamma_1 \Downarrow \sigma_3 * b_3/\Gamma_3}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1\sigma_3 * b_3/\Gamma_3}$$

BRANCHING-EXCEPTION

$$\frac{t_1/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1 \quad b_1 \in \{\text{Return } \_, \text{Exit } \_ \}}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)/\Gamma \Downarrow \sigma_1 * b_1/\Gamma_1}$$

# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- Expansion
- Inconstant semantics
- Control flow

## 2. The CoLiS language

- Requirements
- Definitions

## 3. Formalisation

- **Formulation**
- Proof



# Formalisation

- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,
  - Helps detecting potential mistakes in the semantics,
  - More easily readable than the semantics,
  - Allows us to validate the translation by testing.

# Formalisation

- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,

```
type term = TTrue | TFalse | TFatal
          | TReturn term | TExit term
          | TSeq term term | TIf term term term
          | ...
```

- The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,
  - Helps detecting potential mistakes in the semantics,
  - More easily readable than the semantics,
  - Allows us to validate the translation by testing.

# Formalisation

- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;

```

inductive eval_term term context
      string behaviour context =
| 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

```

- Interpreter proven sound and complete:
  - Written in a “natural way”,
  - Helps detecting potential mistakes in the semantics,
  - More easily readable than the semantics,
  - Allows us to validate the translation by testing.

# Formalisation

- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,
  - Helps detecting potential mistakes in the semantics,
  - More easily readable than the semantics,
  - Allows us to validate the translation by testing.

# Formalisation

- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,

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

let rec interp_term (t: term) ( $\Gamma$ : context)
  (stdout: ref string) : (bool, context)
```

- Helps detecting potential mistakes in the semantics,
- More easily readable than the semantics,
- Allows us to validate the translation by testing.

# Formalisation

- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,

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

let rec interp_term (t: term) ( $\Gamma$ : context)
  (stdout: ref string) : (bool, context)
```

- Helps detecting potential mistakes in the semantics,
  - More easily readable than the semantics,
  - Allows us to validate the translation by testing.

# Formalisation

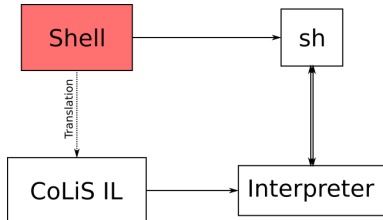
- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,
  - Helps detecting potential mistakes in the semantics,
  - More easily readable than the semantics,

```
| 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
```

- Allows us to validate the translation by testing.

# Formalisation

- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,
  - Helps detecting potential mistakes in the semantics,
  - More easily readable than the semantics,
  - Allows us to validate the translation by testing.





## Soundness of the interpreter

We write  $t/\Gamma \mapsto \sigma \star b/\Gamma'$  for: “on the input consisting of  $t$ ,  $\Gamma$  and a reference, the interpreter writes  $\sigma$  at the end of that reference and terminates:

- normally and outputs  $(b, \Gamma')$ ;
- with an exception corresponding to the behaviour  $b$  that carries  $\Gamma'$ .”

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  $t$ ,  $\Gamma$  and a reference, the interpreter writes  $\sigma$  at the end of that reference and terminates:

- normally and outputs  $(b, \Gamma')$ ;
- with an exception corresponding to the behaviour  $b$  that carries  $\Gamma'$ .”

### Theorem (Soundness of the interpreter)

For all  $t$ ,  $\Gamma$ ,  $\sigma$ ,  $b$  and  $\Gamma'$ : if

$$t/\Gamma \mapsto \sigma \star b/\Gamma'$$

then

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

# Completeness of the interpreter

We write  $t/\Gamma \mapsto \sigma \star b/\Gamma'$  for: “on the input consisting of  $t$ ,  $\Gamma$  and a reference, the interpreter writes  $\sigma$  at the end of that reference and terminates:

- normally and outputs  $(b, \Gamma')$ ;
- with an exception corresponding to the behaviour  $b$  that carries  $\Gamma'$ .”

## Theorem (Completeness of the interpreter)

For all  $t$ ,  $\Gamma$ ,  $\sigma$ ,  $b$  and  $\Gamma'$ : if

$$t/\Gamma \Downarrow \sigma \star b/\Gamma'$$

then

$$t/\Gamma \mapsto \sigma \star b/\Gamma'$$

# Soundness of the interpreter in Why3

```
let rec interp_term (t: term) ( $\Gamma$ : 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  $\Gamma'$  -> 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)  $\Gamma'$  }
```

```
  ...
```

# Soundness of the interpreter in Why3

```
let rec interp_term (t: term) ( $\Gamma$ : 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  $\Gamma'$  -> 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)  $\Gamma'$  }
```

```
  ...
```

# Completeness of the interpreter in Why3

```
lemma functionality: forall t  $\Gamma$   $\sigma_1$   $\sigma_2$   $b_1$   $b_2$   $\Gamma_1$   $\Gamma_2$ .
  eval_term t  $\Gamma$   $\sigma_1$   $b_1$   $\Gamma_1$  ->
  eval_term t  $\Gamma$   $\sigma_2$   $b_2$   $\Gamma_2$  ->
   $\sigma_1 = \sigma_2$  /\  $b_1 = b_2$  /\  $\Gamma_1 = \Gamma_2$ 
```

```
let rec interp_term (t: term) ( $\Gamma$ : 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)  $\Gamma'$  }
```

# Completeness of the interpreter in Why3

```
lemma functionality: forall t  $\Gamma$   $\sigma_1$   $\sigma_2$   $b_1$   $b_2$   $\Gamma_1$   $\Gamma_2$ .
  eval_term t  $\Gamma$   $\sigma_1$   $b_1$   $\Gamma_1$  ->
  eval_term t  $\Gamma$   $\sigma_2$   $b_2$   $\Gamma_2$  ->
   $\sigma_1 = \sigma_2 \wedge b_1 = b_2 \wedge \Gamma_1 = \Gamma_2$ 
```

```
let rec interp_term (t: term) ( $\Gamma$ : 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)  $\Gamma'$  }
```

# Completeness of the interpreter in Why3

```
lemma functionality: forall t  $\Gamma$   $\sigma_1$   $\sigma_2$   $b_1$   $b_2$   $\Gamma_1$   $\Gamma_2$ .
  eval_term t  $\Gamma$   $\sigma_1$   $b_1$   $\Gamma_1$  ->
  eval_term t  $\Gamma$   $\sigma_2$   $b_2$   $\Gamma_2$  ->
   $\sigma_1 = \sigma_2 \wedge b_1 = b_2 \wedge \Gamma_1 = \Gamma_2$ 
```

```
let rec interp_term (t: term) ( $\Gamma$ : 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)  $\Gamma'$  }
```



# Completeness of the interpreter in Why3

```
lemma functionality: forall t  $\Gamma$   $\sigma_1$   $\sigma_2$   $b_1$   $b_2$   $\Gamma_1$   $\Gamma_2$ .
  eval_term t  $\Gamma$   $\sigma_1$   $b_1$   $\Gamma_1$  ->
  eval_term t  $\Gamma$   $\sigma_2$   $b_2$   $\Gamma_2$  ->
   $\sigma_1 = \sigma_2$  /\  $b_1 = b_2$  /\  $\Gamma_1 = \Gamma_2$ 
```

```
let rec interp_term (t: term) ( $\Gamma$ : 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)  $\Gamma'$  }
```

# Table of Contents

## 1. Gallery of horrors in shell

- Dynamic!
- Expansion
- Inconstant semantics
- Control flow

## 2. The CoLiS language

- Requirements
- Definitions

## 3. Formalisation

- Formulation
- **Proof**

# Why it is hard

- `stdout` is a reference.

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

- We cannot provide a witness as a return value here, because of exceptions,
  - We (c|sh)ould change it to something more structured.
  - We decided to use superposition provers.
- 
- We need a variant:
    - The term?
    - The derivation tree of the hypothesis?
    - The height of the derivation tree?
    - The size of the derivation tree?
    - What then?

# Why it is hard

- `stdout` is a reference:

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

- We cannot provide a witness as a return value here, because of exceptions,
  - We (c|sh)ould change it to something more structured.
  - We decided to use superposition provers.
- We need a variant:
    - The term?
    - The derivation tree of the hypothesis?
    - The height of the derivation tree?
    - The size of the derivation tree?
    - What then?

# Why it is hard

- `stdout` is a reference:

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

- We cannot provide a witness as a return value here, because of exceptions,
  - We (c|sh)ould change it to something more structured.
  - We decided to use superposition provers.
- We need a variant:
  - The term?
  - The derivation tree of the hypothesis?
  - The height of the derivation tree?
  - The size of the derivation tree?
  - What then?

# Why it is hard

- `stdout` is a reference:

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

- We cannot provide a witness as a return value here, because of exceptions,
  - We (c|sh)ould change it to something more structured.
  - We decided to use superposition provers.
- We need a variant:
    - The term?
    - The derivation tree of the hypothesis?
    - The height of the derivation tree?
    - The size of the derivation tree?
    - What then?

# Why it is hard

- `stdout` is a reference:

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

- We cannot provide a witness as a return value here, because of exceptions,
  - We (c|sh)ould change it to something more structured.
  - We decided to use superposition provers.
- 
- We need a variant:
    - The term?
    - The derivation tree of the hypothesis?
    - The height of the derivation tree?
    - The size of the derivation tree?
    - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term?
  - The derivation tree of the hypothesis?
  - The height of the derivation tree?
  - The size of the derivation tree?
  - What then?



# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term?
  - The derivation tree of the hypothesis?
  - The height of the derivation tree?
  - The size of the derivation tree?
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**

$$\frac{
 \begin{array}{l}
 t_1/\Gamma \Downarrow \sigma_1 \star b_1/\Gamma_1 \quad b_1 = \text{True} \\
 t_2/\Gamma \Downarrow \sigma_2 \star b_2/\Gamma_2 \quad b_2 \in \{\text{True}, \text{False}\} \\
 (\mathbf{while} \ t_1 \ \mathbf{do} \ t_2)/\Gamma_2 \Downarrow \sigma_3 \star b_3/\Gamma_3
 \end{array}
 }{
 (\mathbf{while} \ t_1 \ \mathbf{do} \ t_2)/\Gamma \Downarrow \sigma_1\sigma_2\sigma_3 \star b_3/\Gamma_3
 }$$

- The derivation tree of the hypothesis?
- The height of the derivation tree?
- The size of the derivation tree?
- What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis?
  - The height of the derivation tree?
  - The size of the derivation tree?
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis?
    - True, but we cannot manipulate them in Why3.
  - The height of the derivation tree?
  - The size of the derivation tree?
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis? **True, but no.**
  - The height of the derivation tree?
  - The size of the derivation tree?
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis? **True, but no.**
  - The height of the derivation tree? **Err... no.**
    - Superposition provers are bad with arithmetic, and we need the maximum function and inequalities.
    - Given the height of a derivation tree, we cannot deduce the heights of the premises (only an upper bound).
  - The size of the derivation tree?
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis? **True, but no.**
  - The height of the derivation tree? **Err... no.**
    - Superposition provers are bad with arithmetic, and we need the maximum function and inequalities.
    - Given the height of a derivation tree, we cannot deduce the heights of the premises (only an upper bound).
  - The size of the derivation tree?
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis? **True, but no.**
  - The height of the derivation tree? **Err... no.**
  - The size of the derivation tree?
  - What then?



# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis? **True, but no.**
  - The height of the derivation tree? **Err... no.**
  - The size of the derivation tree? **Err... no.**
    - Superposition provers are bad with arithmetic, and we need addition and substraction.
    - Given the size of a derivation tree, we cannot deduce the size of the premises.
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis? **True, but no.**
  - The height of the derivation tree? **Err... no.**
  - The size of the derivation tree? **Err... no.**
    - Superposition provers are bad with arithmetic, and we need addition and subtraction.
    - Given the size of a derivation tree, we cannot deduce the size of the premises.
  - What then?

# Why it is hard

- `stdout` is a reference.
- We need a variant:
  - The term? **No.**
  - The derivation tree of the hypothesis? **True, but no.**
  - The height of the derivation tree? **Err... no.**
  - The size of the derivation tree? **Err... no.**
  - What then?

# Skeletons

We add a skeleton type:

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

It represents the “shape” of the proof.

# Skeletons

We add a skeleton type:

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

It represents the “shape” of the proof.

# Skeletons

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  $\Gamma$   $\sigma_1$  b1  $\Gamma_1$  t2  $\sigma_2$  b2  $\Gamma_2$  sk1 sk2.
  eval_term t1  $\Gamma$   $\sigma_1$  (BNormal b1)  $\Gamma_1$  sk1 ->
  eval_term t2  $\Gamma_1$   $\sigma_2$  b2  $\Gamma_2$  sk2 ->
  eval_term (TSeq t1 t2)  $\Gamma$  (concat  $\sigma_1$   $\sigma_2$ ) b2  $\Gamma_2$  (S2 sk1 sk2)

| EvalT_Seq_Error : forall t1  $\Gamma$   $\sigma_1$  b1  $\Gamma_1$  t2 sk.
  eval_term t1  $\Gamma$   $\sigma_1$  b1  $\Gamma_1$  sk ->
  (match b1 with BNormal _ -> false | _ -> true end) ->
  eval_term (TSeq t1 t2)  $\Gamma$   $\sigma_1$  b1  $\Gamma_1$  (S1 sk)

```

## Put them everywhere – In the contract

```
let rec interp_term (t: term) ( $\Gamma$ : 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 }
  ensures { match sk with S1 sk1 | S2 sk1 _ -> result = sk1 |
    _ -> absurd }
= 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 premises 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 |
    _ -> absurd }
= 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 premises and we name the first one `sk1`.”

## Put them everywhere – In the code

```
| TSeq t1 t2 ->
  let ghost sk1 = skeleton12 sk in
  let (_,  $\Gamma_1$ ) = interp_term t1  $\Gamma$  stdout sk1 in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term t2  $\Gamma_1$  stdout sk2
```

```
| TIf t1 t2 t3 ->
  let (b1,  $\Gamma_1$ ) =
    try
      let ghost sk1 = skeleton12 sk in
      interp_term t1  $\Gamma$  stdout sk1
    with
      EFatal  $\Gamma'$  -> (false,  $\Gamma'$ )
  end
  in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term (if b1 then t2 else t3)  $\Gamma_1$  stdout sk2
```

## Put them everywhere – In the code

```

| TSeq t1 t2 ->
  let ghost sk1 = skeleton12 sk in
  let (_,  $\Gamma_1$ ) = interp_term t1  $\Gamma$  stdout sk1 in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term t2  $\Gamma_1$  stdout sk2

```

```

| TIf t1 t2 t3 ->
  let (b1,  $\Gamma_1$ ) =
    try
      let ghost sk1 = skeleton12 sk in
      interp_term t1  $\Gamma$  stdout sk1
    with
      EFatal  $\Gamma'$  -> (false,  $\Gamma'$ )
  end
  in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term (if b1 then t2 else t3)  $\Gamma_1$  stdout sk2

```

And it's all green!



	Soundness	Completeness
Proof obligations	117	233
Time (seconds)	190	510

# Other things about skeletons

- Generalisable, if we want more than the shape;
- Help in writing recursion in case of mutually recursive types;
- 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;
- 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;
- 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;
- 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;
- 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*