Formalising an intermediate language for POSIX shell

Nicolas Jeannerod

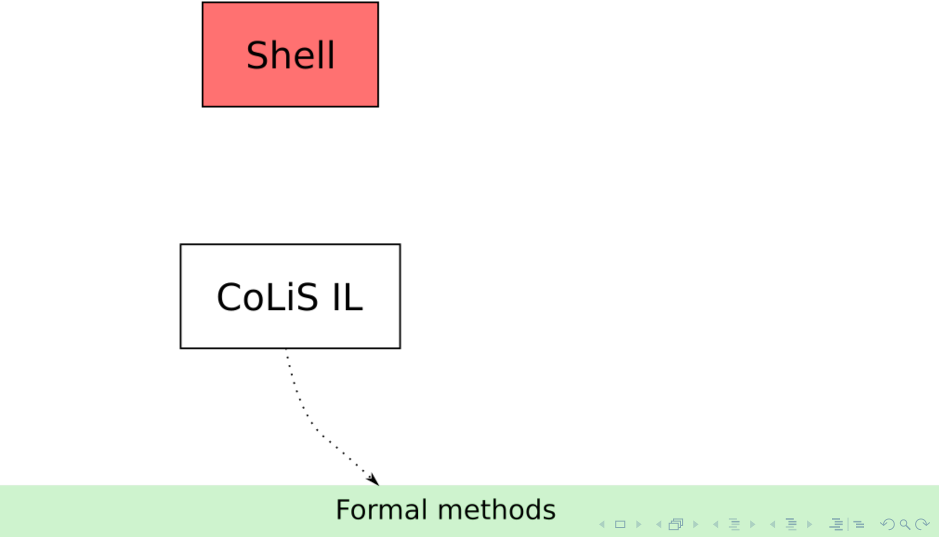
INSTITUT
DE RECHERCHE
EN INFORMATIQUE
FONDAMENTALE

Séminaire Gallium, Septembre 18, 2017

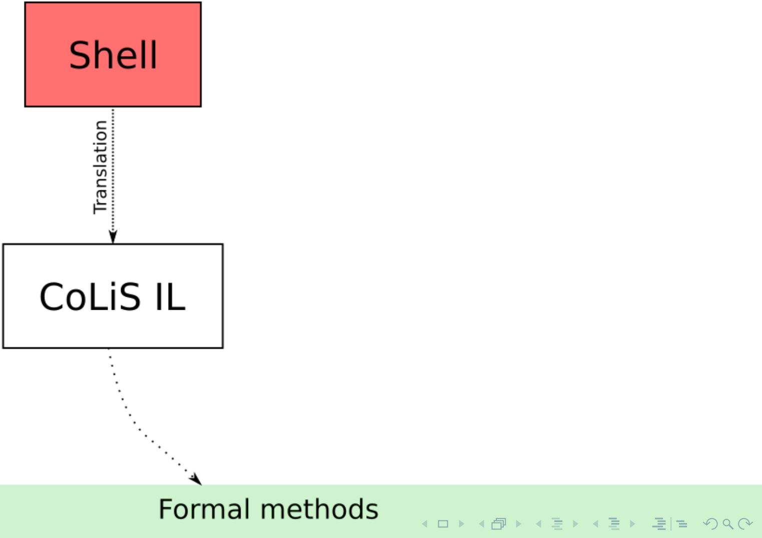
Big picture

Formal methods

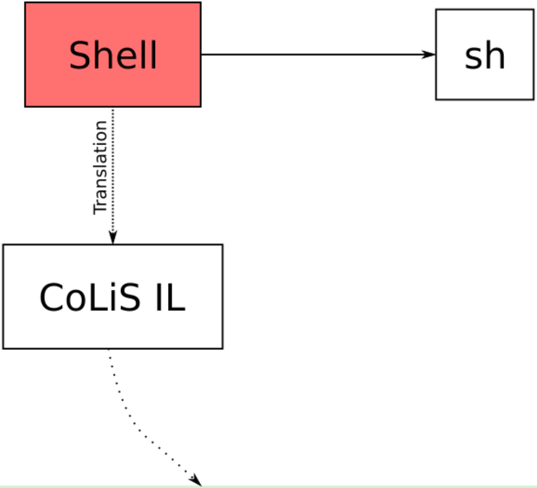
Big picture



Big picture

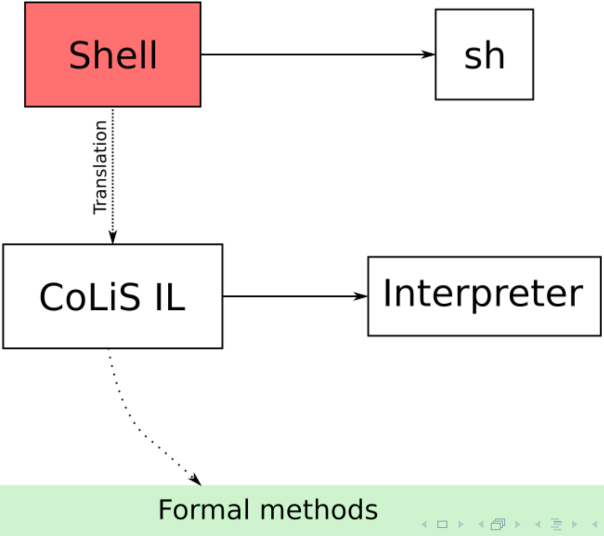


Big picture

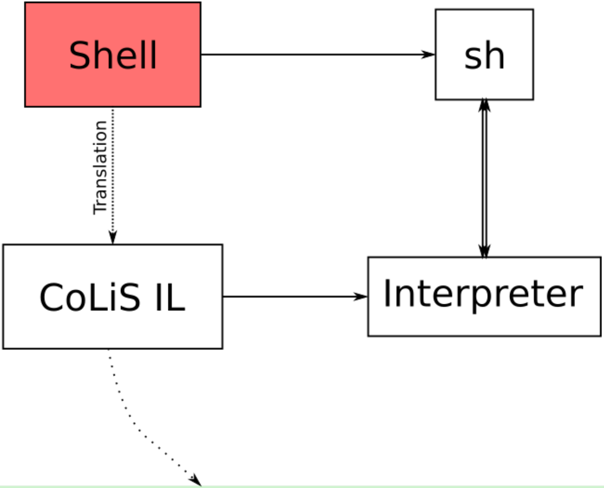


Formal methods

Big picture



Big picture



Formal methods

Table of Contents

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

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

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

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

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

Syntax

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

Syntax

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

Syntax

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

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

Syntax

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

Syntax

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

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

Syntax

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

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

Semantic judgements

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

A *context* Γ 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* Γ 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* Γ 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* Γ 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* Γ 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* Γ 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* Γ 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* Γ 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* Γ 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* Γ 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

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 = \mathsf{True} \qquad t_{2/\Gamma_1} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star b_{2/\Gamma_2}}$$

Branching-False

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\mathsf{False}, \mathsf{Fatal}\} \qquad t_{3/\Gamma_1} \Downarrow \sigma_3 \star b_{3/\Gamma_3}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_3 \star b_{3/\Gamma_3}}$$

Branching-Exception

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

Semantic rules – Branching

$$\textsc{Branching-True}\quad \frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 = \mathsf{True} \qquad t_{2/\Gamma_1} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star b_{2/\Gamma_2}}$$

$$\textsc{Branching-False}\quad \frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\mathsf{False}, \mathsf{Fatal}\} \qquad t_{3/\Gamma_1} \Downarrow \sigma_3 \star b_{3/\Gamma_3}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_3 \star b_{3/\Gamma_3}}$$

$$\textsc{Branching-Exception}\quad \frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\mathsf{Return}\ \_, \mathsf{Exit}\ \_\}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}}$$

Semantic rules – Branching

Branching-True

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 = \mathsf{True} \qquad t_{2/\Gamma_1} \Downarrow \sigma_2 \star b_{2/\Gamma_2}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_2 \star b_{2/\Gamma_2}}$$

Branching-False

$$\frac{t_{1/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1} \qquad b_1 \in \{\mathsf{False}, \mathsf{Fatal}\} \qquad t_{3/\Gamma_1} \Downarrow \sigma_3 \star b_{3/\Gamma_3}}{(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)_{/\Gamma} \Downarrow \sigma_1\sigma_3 \star b_{3/\Gamma_3}}$$

Branching-Exception

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

Table of Contents

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) (Γ: 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) (Γ: 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

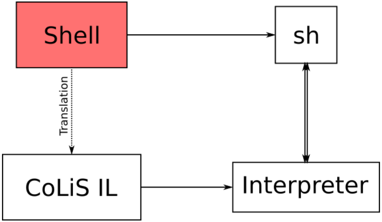
- Formalised in the proof environment Why3:
  - The syntax becomes an algebraic data type,
  - The semantics become an inductive predicate;
- Interpreter proven sound and complete:
  - Written in a “natural way”,
  - Helps detecting potential mistakes in the semantics,
  - More easily readable than the 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, Γ *and a reference, the interpreter writes* σ *at the end of that reference and terminates:*

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

Theorem (Soundness of the interpreter)

For all t, Γ, σ, b *and* Γ'*: if*

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

then

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

Soundness of the interpreter

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

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

Theorem (Soundness of the interpreter)

For all t, Γ, σ, b *and* Γ'*: if*

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

then

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

Completeness 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 (Completeness of the interpreter)

For all t, Γ, σ, b *and* Γ'*: if*

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

then

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

Soundness of the interpreter in Why3

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

  diverges

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

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

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

  ...
```

Soundness of the interpreter in Why3

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

  diverges

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

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

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

  ...
```

Completeness of the interpreter in Why3

```
lemma functionality: forall t Γ σ1 σ2 b1 b2 Γ1 Γ2.
  eval_term t Γ σ1 b1 Γ1 ->
  eval_term t Γ σ2 b2 Γ2 ->
 σ1 = σ2 /\ b1 = b2 /\ Γ1 = Γ2
```

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

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

```
  variant { ??? }
```

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

Completeness of the interpreter in Why3

```
lemma functionality: forall t Γ σ1 σ2 b1 b2 Γ1 Γ2.
  eval_term t Γ σ1 b1 Γ1 ->
  eval_term t Γ σ2 b2 Γ2 ->
 σ1 = σ2 /\ b1 = b2 /\ Γ1 = Γ2
```

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

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

  variant { ??? }

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

Completeness of the interpreter in Why3

```
lemma functionality: forall t Γ σ1 σ2 b1 b2 Γ1 Γ2.
  eval_term t Γ σ1 b1 Γ1 ->
  eval_term t Γ σ2 b2 Γ2 ->
 σ1 = σ2 /\ b1 = b2 /\ Γ1 = Γ2
```

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

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

  variant { ??? }

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

Completeness of the interpreter in Why3

```
lemma functionality: forall t Γ σ1 σ2 b1 b2 Γ1 Γ2.
  eval_term t Γ σ1 b1 Γ1 ->
  eval_term t Γ σ2 b2 Γ2 ->
 σ1 = σ2 /\ b1 = b2 /\ Γ1 = Γ2
```

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

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

```
  variant { ??? }
```

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

Table of Contents

Why it is hard

- `stdout` is a reference.

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

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

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

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

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

  - 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.**

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

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

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

Put them everywhere – In the contract

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

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

  variant { sk }

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

Define some helpers

```
let ghost skeleton12 (sk: skeleton)
  requires { match sk with S1 _ | S2 _ _ -> true | _ -> false e
  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 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 | 
  = 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 (_, Γ1) = interp_term t1 Γ stdout sk1 in
  let ghost (_, sk2) = skeleton2 sk in
  interp_term t2 Γ1 stdout sk2

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

Put them everywhere – In the code

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

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

And it's all green!

	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