A Formally Verified Interpreter for a Shell-like Programming Language

Claude Marché Nicolas Jeannerod Ralf Treinen

VSTTE, July 22, 2017

General goal

The CoLiS project. "Correctness of Linux Scripts"

Goal: Apply verification techniques to shell scripts in the Debian packages

```
set -e
eval "if true; then cmd='echo foo'; fi"
( cmd="$cmd bar" )
exit 1 | $cmd
"$cmd"
```

General goal

The CoLiS project. “Correctness of Linux Scripts”
Goal: Apply verification techniques to shell scripts in the Debian packages

```
set -e
eval "if true; then cmd=’echo foo’; fi"
( cmd="$cmd bar" )
exit 1 | $cmd
"$cmd"
```

General goal

The CoLiS project. “Correctness of Linux Scripts”
Goal: Apply verification techniques to shell scripts in the Debian packages

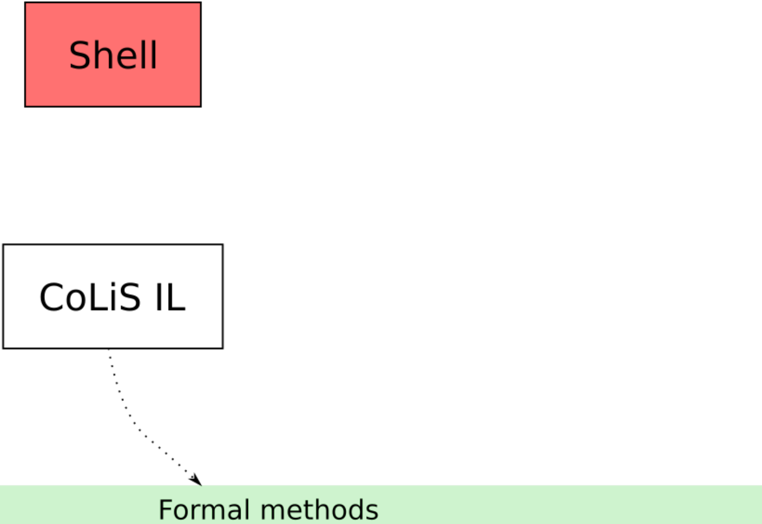
```
set -e
eval "if true; then cmd=’echo foo’; fi"
( cmd="$cmd bar" )
exit 1 | $cmd
"$cmd"
```

Big picture

Shell

Formal methods

Big picture

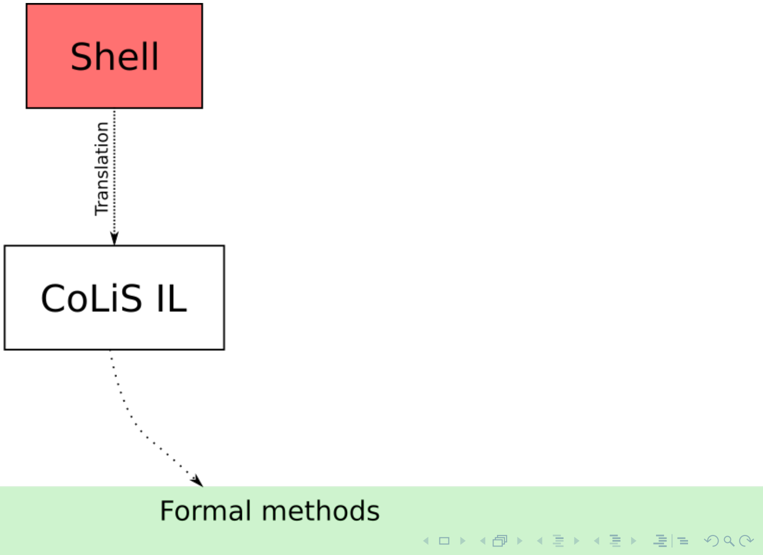


Shell

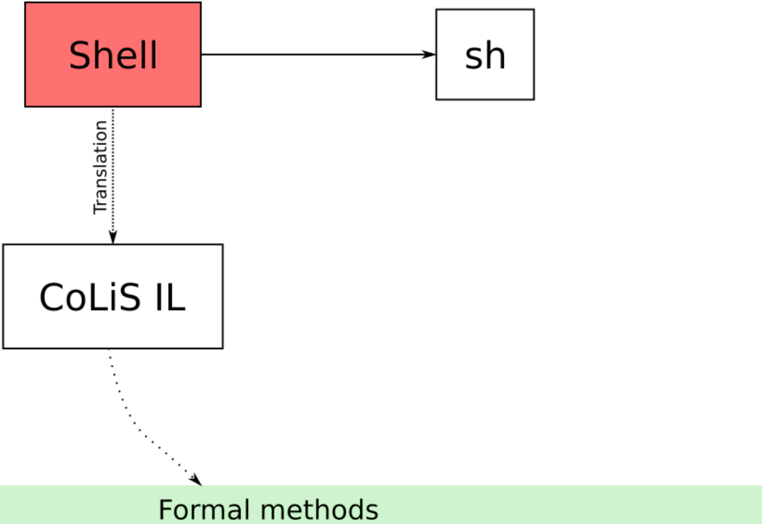
CoLiS IL

Formal methods

Big picture



Big picture


Shell

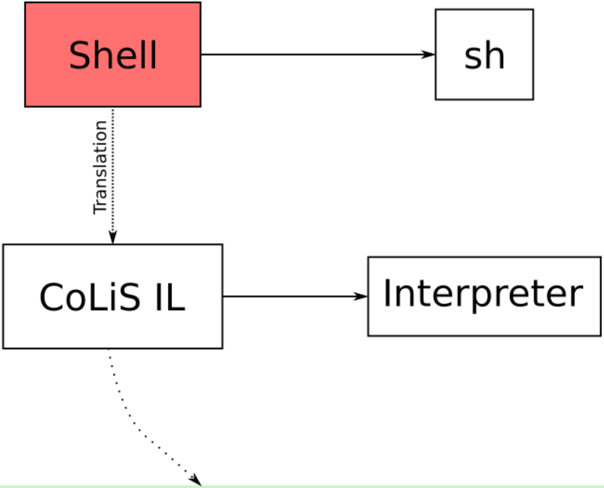
sh

Translation

CoLiS IL

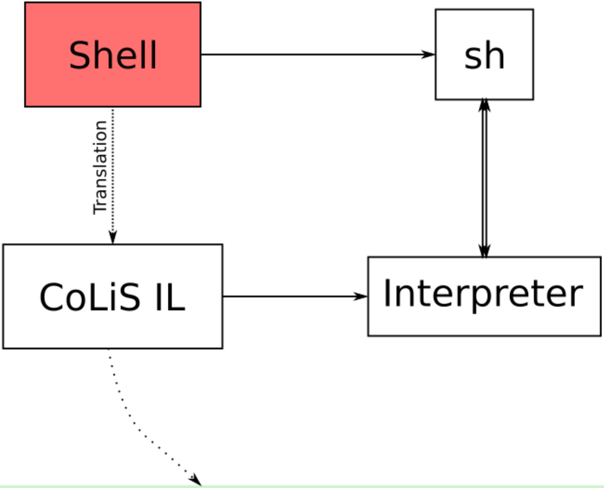
Formal methods

Big picture



Formal methods

Big picture



Formal methods

Table of Contents

Table of Contents

Requirements

- Intermediate language (not a replacement of Shell);
- Clean;
- With formal syntax and semantics;

- Statically typed: strings and lists;
- Variables and functions explicitely declared in a header;
- Dangerous structures made more explicit.

However, automatic translation from reasonnable Shell must be possible.

Requirements

- Intermediate language (not a replacement of Shell);
- Clean;
- With formal syntax and semantics;
- Statically typed: strings and lists;
- Variables and functions explicitely declared in a header;
- Dangerous structures made more explicit.

However, automatic translation from reasonnable Shell must be possible.

Requirements

- Intermediate language (not a replacement of Shell);

- Clean;
- With formal syntax and semantics;

- Statically typed: strings and lists;
- Variables and functions explicitely declared in a header;
- Dangerous structures made more explicit.

However, automatic translation from reasonnable Shell must be possible.

Requirements

- Intermediate language (not a replacement of Shell);

- Clean;
- With formal syntax and semantics;

- Statically typed: strings and lists;
- Variables and functions explicitely declared in a header;
- Dangerous structures made more explicit.

However, automatic translation from reasonnable Shell must be possible.

A glimpse of the language

```
fruits="banana apple .."

{
  for fruit in $fruits
  do
    echo "$fruit"
  done
} | {
  while read line
  do
    echo "- $line"
  done
}
```

```
var fruits : list
var fruit : string
var line : string

begin
  fruits ::= [ ’banana’ ; ’apple’ ; .. ]

  pipe
    for fruit in [fruits]
    do
      call [ ’echo’ ; {fruit} ] ;
    done
  into
    while call [ ’read’ ; ’line’ ]
    do
      call [ ’echo’ ; {’- ’ , line} ] ;
  end
end
```

A glimpse of the language

```
fruits="banana apple .."

{
  for fruit in $fruits
  do
    echo "$fruit"
  done
} | {
  while read line
  do
    echo "- $line"
  done
}
```

```
var fruits : list
var fruit : string
var line : string

begin
  fruits ::= [ 'banana' ; 'apple' ; .. ]

  pipe
    for fruit in [fruits]
    do
      call [ 'echo' ; {fruit} ] ;
    done
  into
    while call [ 'read' ; 'line' ]
    do
      call [ 'echo' ; {'- ' , line} ] ;
  end
end
```

How behaviours are handled

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

Interactions between Do-While and Fatal

$$\dfrac{\textsc{DoWhile-Test-Fatal} \\ t_{1/\Gamma} \Downarrow \sigma_1 \star \mathsf{True}_{/\Gamma_1} \qquad t_{2/\Gamma_1} \Downarrow \sigma_2 \star \mathsf{Fatal}_{/\Gamma_2}}{\textbf{do } t_1 \textbf{ while } t_{2/\Gamma} \Downarrow \sigma_1\sigma_2 \star \mathsf{True}_{/\Gamma_2}}$$

$$\dfrac{\textsc{DoWhile-Body-Fatal} \\ t_{1/\Gamma} \Downarrow \sigma_1 \star \mathsf{Fatal}_{/\Gamma_1}}{\textbf{do } t_1 \textbf{ while } t_{2/\Gamma} \Downarrow \sigma_1 \star \mathsf{Fatal}_{/\Gamma_1}}$$

Interactions between Do-While and Fatal

$$\textsc{DoWhile-Test-Fatal}\quad \frac{t_{1/\Gamma} \Downarrow \sigma_1 \star \mathsf{True}_{/\Gamma_1} \qquad t_{2/\Gamma_1} \Downarrow \sigma_2 \star \mathsf{Fatal}_{/\Gamma_2}}{\textbf{do } t_1 \textbf{ while } t_{2/\Gamma} \Downarrow \sigma_1\sigma_2 \star \mathsf{True}_{/\Gamma_2}}$$

$$\textsc{DoWhile-Body-Fatal}\quad \frac{t_{1/\Gamma} \Downarrow \sigma_1 \star \mathsf{Fatal}_{/\Gamma_1}}{\textbf{do } t_1 \textbf{ while } t_{2/\Gamma} \Downarrow \sigma_1 \star \mathsf{Fatal}_{/\Gamma_1}}$$

Table of Contents

Why3

- Deductive verification platform;
- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Native support of imperative features:
  - references,
  - exceptions,
  - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Deductive verification platform;
- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Native support of imperative features:
  - references,
  - exceptions,
  - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Deductive verification platform;
- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Native support of imperative features:
  - references,
  - exceptions,
  - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Deductive verification platform;
- WhyML: language for both specification and programming;
- Standard library:
  - integer arithmetic,
  - boolean operations,
  - maps,
  - etc.;
- Native support of imperative features:
  - references,
  - exceptions,
  - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Deductive verification platform;
- WhyML: language for both specification and programming;
- Standard library:
    - integer arithmetic,
    - boolean operations,
    - maps,
    - etc.;
- Native support of imperative features:
    - references,
    - exceptions,
    - while and for loops;
- Proof obligations are given to external theorem provers;
- Possibility to extract WhyML code to OCaml.

Why3

- Deductive verification platform;

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

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

Syntax

```
type term =
  | TTrue
  | TFalse
  | TFatal
  | TReturn term
  | TExit term
  | TAsString svar sexpr
  | TAsList lvar lexpr
  | TSeq term term
  | TIf term term term
  | TFor svar lexpr term
  | TDoWhile term term
  | TProcess term
  | TCall lexpr
  | TShift
  | TPipe term term
```

```
with sexpr = list sfrag

with sfrag =
  | SLiteral string
  | SVar svar
  | SArg int
  | SProcess term

with lexpr = list lfrag

with lfrag =
  | LSingleton sexpr
  | LSplit sexpr
  | LVar lvar
```

Semantic judgments (excerpt)

```
inductive eval_term term context
                    string behaviour context

| EvalT_DoWhile_False : forall t1 Γ σ1 b1 Γ1 t3 σ3 b3 Γ3 t2.
  eval_term t1 Γ σ1 (BNormal b1) Γ1 ->
  eval_term t2 Γ1 σ2 b2 Γ2 ->
  (match b2 with BNormal False | BFatal -> true | _ -> false en
  eval_term (TDoWhile t1 t2) Γ (concat σ1 σ2) (BNormal b1) Γ2

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

Table of Contents

Interpreter (excerpt)

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

  | TDoWhile t₁ t₂ ->
    let (b₁, Γ₁) = interp_term t₁ Γ stdout in
    let (b₂, Γ₂) =
      try
        interp_term t₂ Γ₁ stdout
      with
        EFatal Γ₂ -> (false, Γ₂)
      end
    in
    if b₂ then
      interp_term t Γ₂ stdout
    else
      (b₁, Γ₂)
```

Table of Contents

Soundness of the interpreter

Theorem (Soundness of the interpreter)

For all t, Γ, σ, b and Γ': if

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

then

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

Contract (excerpt)

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

Contract (excerpt)

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

Why it is non trivial

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value:
  - May only be used for specification,
  - Must not affect the semantics of the program.
- Does not fit with exceptions;
- Forces us to use superposition provers.

Why it is non trivial

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value:
  - May only be used for specification,
  - Must not affect the semantics of the program.
- Does not fit with exceptions;
- Forces us to use superposition provers.

Why it is non trivial

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value:
  - May only be used for specification,
  - Must not affect the semantics of the program.
- Does not fit with exceptions;
- Forces us to use superposition provers.

Why it is non trivial

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value:
    - May only be used for specification,
    - Must not affect the semantics of the program.
- Does not fit with exceptions;
- Forces us to use superposition provers.

Why it is non trivial

- `stdout` is a reference:

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

- Usual fix: provide a witness as a ghost return value:
  - May only be used for specification,
  - Must not affect the semantics of the program.
- Does not fit with exceptions;
- Forces us to use superposition provers.

Table of Contents

Completeness of the interpreter

Theorem (Completeness of the interpreter)

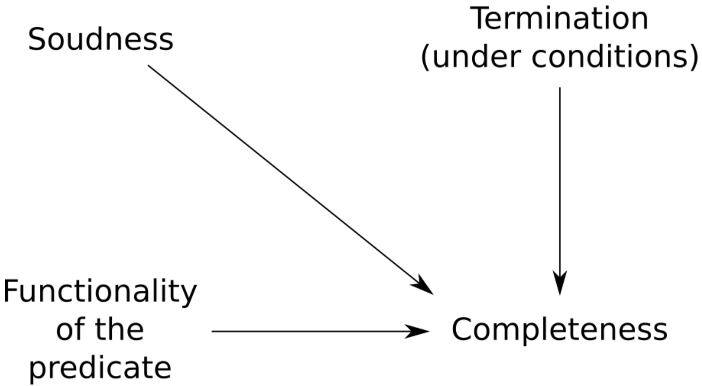
For all t, Γ, σ, b and Γ': if

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

then

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

Proofs dependencies



Soudness

Termination
(under conditions)

Functionality
of the
predicate

Completeness

Why

- If:

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

- then the interpreter terminates:

$$t_{/\Gamma} \mapsto \sigma_1 \star b_{1/\Gamma_1}$$

- then (Soundness):

$$t_{/\Gamma} \Downarrow \sigma_1 \star b_{1/\Gamma_1}$$

- then (Functionality):

$$\sigma = \sigma_1 \ \wedge \ b = b_1 \ \wedge \ \Gamma' = \Gamma_1$$

Why

- If:

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

- then the interpreter terminates:

$$t_{/\Gamma} \mapsto \sigma_1 \star {b_1}_{/\Gamma_1}$$

- then (Soundness):

$$t_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}$$

- then (Functionality):

$$\sigma = \sigma_1 \ \wedge \ b = b_1 \ \wedge \ \Gamma' = \Gamma_1$$

Why

- If:

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

- then the interpreter terminates:

$$t_{/\Gamma} \mapsto \sigma_1 \star {b_1}_{/\Gamma_1}$$

- then (Soundness):

$$t_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}$$

- then (Functionality):

$$\sigma = \sigma_1 \;\wedge\; b = b_1 \;\wedge\; \Gamma' = \Gamma_1$$

Why

- If:

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

- then the interpreter terminates:

$$t_{/\Gamma} \mapsto \sigma_1 \star {b_1}_{/\Gamma_1}$$

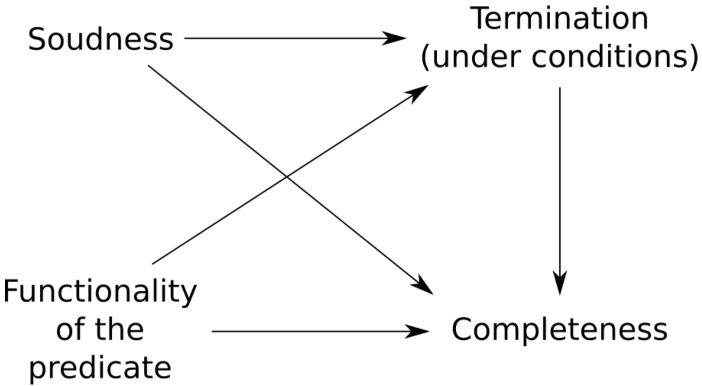
- then (Soundness):

$$t_{/\Gamma} \Downarrow \sigma_1 \star {b_1}_{/\Gamma_1}$$

- then (Functionality):

$$\sigma = \sigma_1 \;\wedge\; b = b_1 \;\wedge\; \Gamma' = \Gamma_1$$

Proofs dependencies



Soudness

Termination (under conditions)

Functionality of the predicate

Completeness

Why do we need all this?

Case of the sequence:

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

- By hypothesis / pre-condition, there is σ, b and Γ'' such that:

$$(t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''}$$

- By structure of the predicate, there is σ', b', and Γ' such that:

$$t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \ \wedge\ t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''}$$

- By soundness and functionality, $\Gamma' = \Gamma_1$.

Why do we need all this?

Case of the sequence:

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

- By hypothesis / pre-condition, there is σ, b and Γ'' such that:

$$(t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''}$$

- By structure of the predicate, there is σ', b', and Γ' such that:

$$t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \;\wedge\; t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''}$$

- By soundness and functionality, $\Gamma' = \Gamma_1$.

Why do we need all this?

Case of the sequence:

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

- By hypothesis / pre-condition, there is σ, b and Γ'' such that:

$$(t_1\ ;\ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''}$$

- By structure of the predicate, there is σ', b', and Γ' such that:

$$t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \ \wedge\ t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''}$$

- By soundness and functionality, $\Gamma' = \Gamma_1$.

Why do we need all this?

Case of the sequence:

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

- By hypothesis / pre-condition, there is σ, b and Γ'' such that:

$$(t_1 \ ; \ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''}$$

- By structure of the predicate, there is σ', b', and Γ' such that:

$${t_1}_{/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \ \wedge \ {t_2}_{/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''}$$

- By soundness and functionality, $\Gamma' = \Gamma_1$.

Why do we need all this?

Case of the sequence:

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

- By hypothesis / pre-condition, there is σ, b and Γ'' such that:

$$(t_1 \ ; \ t_2)_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma''}$$

- By structure of the predicate, there is σ', b', and Γ' such that:

$$t_{1/\Gamma} \Downarrow \sigma' \star b'_{/\Gamma'} \ \wedge \ t_{2/\Gamma'} \Downarrow \sigma \star b_{/\Gamma''}$$

- By soundness and functionality, $\Gamma' = \Gamma_1$.

Termination of the interpreter, in Why3

```
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' }

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

  variant { ... }
```

Termination of the interpreter, in Why3

```
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' }

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

  variant { ... }
```

Termination of the interpreter, in Why3

```
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' }

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

  variant { ... }
```

Table of Contents

Let us find a variant

- CoLiS programs are structurally decreasing? **Wrong.**

$$\textsc{DoWhile-True}\quad \dfrac{t_{1/\Gamma} \Downarrow \sigma_1 \star \mathsf{True}_{/\Gamma_1} \qquad t_{2/\Gamma_1} \Downarrow \sigma_2 \star \mathsf{True}_{/\Gamma_2} \qquad \textbf{do } t_1 \textbf{ while } t_{2/\Gamma_2} \Downarrow \sigma_3 \star b_{3/\Gamma_3}}{\textbf{do } t_1 \textbf{ while } t_{2/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_{3/\Gamma_3}}$$

- Derivation trees of the semantics are structurally decreasing? **True**, but we cannot manipulate them in Why3.

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

Let us find a variant

- CoLiS programs are structurally decreasing? **Wrong.**

$$\textsc{DoWhile-True}\quad \frac{t_{1/\Gamma} \Downarrow \sigma_1 \star \mathsf{True}_{/\Gamma_1} \qquad t_{2/\Gamma_1} \Downarrow \sigma_2 \star \mathsf{True}_{/\Gamma_2} \qquad \mathbf{do}\ t_1\ \mathbf{while}\ t_{2/\Gamma_2} \Downarrow \sigma_3 \star b_{3/\Gamma_3}}{\mathbf{do}\ t_1\ \mathbf{while}\ t_{2/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_{3/\Gamma_3}}$$

- Derivation trees of the semantics are structurally decreasing? **True**, but we cannot manipulate them in Why3.

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

Let us find a variant

- CoLiS programs are structurally decreasing? **Wrong.**

$$\textsc{DoWhile-True}\quad \frac{t_{2\,/\Gamma_1} \Downarrow \sigma_2 \star \mathsf{True}_{/\Gamma_2} \qquad \dfrac{t_{1\,/\Gamma} \Downarrow \sigma_1 \star \mathsf{True}_{/\Gamma_1}}{\textbf{do } t_1 \textbf{ while } t_{2\,/\Gamma_2} \Downarrow \sigma_3 \star b_{3\,/\Gamma_3}}}{\textbf{do } t_1 \textbf{ while } t_{2\,/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_{3\,/\Gamma_3}}$$

- Derivation trees of the semantics are structurally decreasing?
  True, but we cannot manipulate them in Why3.

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

Let us find a variant

- CoLiS programs are structurally decreasing? **Wrong.**

$$\textsc{DoWhile-True}\quad \frac{t_2{}_{/\Gamma_1} \Downarrow \sigma_2 \star \mathsf{True}_{/\Gamma_2} \qquad \begin{array}{c} t_1{}_{/\Gamma} \Downarrow \sigma_1 \star \mathsf{True}_{/\Gamma_1} \\ \textbf{do}\ t_1\ \textbf{while}\ t_2{}_{/\Gamma_2} \Downarrow \sigma_3 \star b_3{}_{/\Gamma_3} \end{array}}{\textbf{do}\ t_1\ \textbf{while}\ t_2{}_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_3{}_{/\Gamma_3}}$$

- Derivation trees of the semantics are structurally decreasing? **True**, but we cannot manipulate them in Why3.

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

Let us find a variant

- CoLiS programs are structurally decreasing? **Wrong.**

$$\textsc{DoWhile-True}\quad \frac{t_2{}_{/\Gamma_1} \Downarrow \sigma_2 \star \mathsf{True}_{/\Gamma_2} \qquad \overset{t_1{}_{/\Gamma} \Downarrow \sigma_1 \star \mathsf{True}_{/\Gamma_1}}{\textbf{do } t_1 \textbf{ while } t_2{}_{/\Gamma_2} \Downarrow \sigma_3 \star b_3{}_{/\Gamma_3}}}{\textbf{do } t_1 \textbf{ while } t_2{}_{/\Gamma} \Downarrow \sigma_1\sigma_2\sigma_3 \star b_3{}_{/\Gamma_3}}$$

- Derivation trees of the semantics are structurally decreasing? **True**, but we cannot manipulate them in Why3.

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

Why it does not work

- Superposition provers are bad with arithmetic.
- SMT solvers are bad with existential quantifications.

- We cannot deduce from the height of a derivation tree the heights of the premises.

- We cannot deduce from the size of a derivation tree the sizes of the premises.

Why it does not work

- Superposition provers are bad with arithmetic.
- SMT solvers are bad with existential quantifications.

- We cannot deduce from the height of a derivation tree the heights of the premises.

- We cannot deduce from the size of a derivation tree the sizes of the premises.

Why it does not work

- Superposition provers are bad with arithmetic.
- SMT solvers are bad with existential quantifications.

- We cannot deduce from the height of a derivation tree the heights of the premises.

- We cannot deduce from the size of a derivation tree the sizes of the premises.

Table of Contents

Back to square one

- We still want to say that proofs are structurally decreasing.
- We add a `skeleton` type:

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

- It represents the “shape” of the proof.

Back to square one

- We still want to say that proofs are structurally decreasing.
- We add a `skeleton` type:

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

- It represents the “shape” of the proof.

Back to square one

- We still want to say that proofs are structurally decreasing.
- We add a `skeleton` type:

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

- It represents the “shape” of the proof.

Put them everywhere – In the predicate

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

| EvalT_DoWhile_True : forall t1 Γ σ1 b1 Γ1 t2 σ2 b2 Γ2 t3 sk1 sk2 sk3.
  eval_term t1 Γ σ1 (BNormal b1) Γ1 sk1 ->
  eval_term t2 Γ1 σ2 (BNormal True) Γ2 sk2 ->
  eval_term (TDoWhile t1 t2) Γ2 σ3 b3 Γ3 sk3 ->
  eval_term (TDoWhile t1 t2) Γ
            (concat (concat σ1 σ2) σ3) b3 Γ3 (S3 sk1 sk2 sk3)

| EvalT_DoWhile_False : forall t1 Γ σ1 b1 Γ1 t3 σ3 b3 Γ3 t2 sk1 sk2.
  eval_term t1 Γ σ1 (BNormal b1) Γ1 sk1 ->
  eval_term t2 Γ1 σ2 b2 Γ2 sk2 ->
  (match b2 with BNormal False | BFatal -> true | _ -> false en
  eval_term (TDoWhile t1 t2) Γ
            (concat σ1 σ2) (BNormal b1) Γ2 (S2 sk1 sk2)
```

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 }

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

  variant { sk }
```

Put them everywhere – In the code

```
| TDoWhile t1 t2 ->
  let ghost sk1 = get_skeleton123 sk in
  let (b1, Γ1) = interp_term t1 Γ stdout sk1 in
  let (b2, Γ2) =
    try
      let ghost (_, sk2) = get_skeleton23 sk in
      interp_term t2 Γ1 stdout sk2
    with
      EFatal Γ2 -> (false, Γ2)
    end
  in
  if b2 then
    let ghost (_, _, sk3) = get_skeleton3 sk in
    interp_term t Γ2 stdout
  else
    (b1, Γ2)
```

And it works!

- Soundness proof:
  - 120 proof obligations;
  - 190 seconds (i7 processor, no parallelisation);
  - Uses Alt-Ergo, Z3 and E (crucially);
  - Entirely automatic.
- Termination proof:
  - 230 proof obligations;
  - 510 seconds;
  - Uses Alt-Ergo, Z3 and E;
  - Still entirely automatic.

Conclusion

- CoLiS is an abstraction of a subset of Shell;
- Its syntax and semantics are formalised in Why3;
- The reference interpreter is proven sound and complete *w.r.t.* the semantics;
- This proof uses SMT solvers, superposition provers and proof trees as first class values.

Thank you for your attention!
Questions? Comments? Suggestions?

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR
ANR

Conclusion

- CoLiS is an abstraction of a subset of Shell;
- Its syntax and semantics are formalised in Why3;
- The reference interpreter is proven sound and complete *w.r.t.* the semantics;
- This proof uses SMT solvers, superposition provers and proof trees as first class values.

Thank you for your attention!
Questions? Comments? Suggestions?

PROJET FINANCÉ PAR L'ANR - PROJECT FUNDED BY THE ANR

Shell exemple

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

Shell exemple

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

Syntax – 1

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

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

Syntax – 2

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

Syntax – 3

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

Semantics – First definitions

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

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

Semantics – First definitions

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

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

Semantic judgments

Judgments: terms	$t_{/\Gamma} \Downarrow \sigma \star b_{/\Gamma'}$
Judgments: string fragment	${f_s}_{/\Gamma} \Downarrow_{sf} \sigma \star \beta_{/\Gamma'}$
Judgments: string expression	$s_{/\Gamma} \Downarrow_{s} \sigma \star \beta_{/\Gamma'}$
Judgments: list fragment	${f_l}_{/\Gamma} \Downarrow_{lf} \lambda \star \beta_{/\Gamma'}$
Judgments: list expression	$l_{/\Gamma} \Downarrow_{l} \lambda \star \beta_{/\Gamma'}$

A few rules – Sequence

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

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

A few rules – Sequence

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

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

A few rules – Branching

Branching-True

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

Branching-False

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

Branching-Exception

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

A few rules – Branching

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

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

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

A few rules – Branching

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

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

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

A few rules – Sequence

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

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

A few rules – Sequence

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

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

A few rules – Branching

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

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

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

A few rules – Branching

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


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


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

A few rules – Branching

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


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


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