*

Explaining Proof Failures with Giant-Step Runtime Assertion Checking

Benedikt Becker, Cláudio Belo Lourenço, Claude Marché
Inria Saclay, France

6th Workshop on Formal Integrated Development Environment
24-25 May, 2021

Motivation: Proof failures

- Why3 platform for deductive program verification
- prove that program satisfies formal specification

```
use int.Int

let main1 (x: int)
= let y = x + 1 in
  assert { y <> 43 }
```

```
let f (x: int) : int
  ensures { result > x }
= x + 1

let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

Motivation: Proof failures

- Why3 platform for deductive program verification
- prove that program satisfies formal specification

```
use int.Int

let main1 (x: int)
= let y = x + 1 in
  assert { y <> 43 }
```

```
let f (x: int) : int
  ensures { result > x }
= x + 1

let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

Category of proof failure

- Non-conformance

Motivation: Proof failures

- Why3 platform for deductive program verification
- prove that program satisfies formal specification

```
use int.Int

let main1 (x: int)
= let y = x + 1 in
  assert { y <> 43 }
```

```
let f (x: int) : int
  ensures { result > x }
= x + 1

let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

Category of proof failure

- Non-conformance
- Sub-contract weakness

Motivation: Proof failures

- Why3 platform for deductive program verification
- prove that program satisfies formal specification

```
use int.Int

let main1 (x: int)
= let y = x + 1 in
  assert { y <> 43 }
```

```
let f (x: int) : int
  ensures { result > x }
= x + 1

let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

Category of proof failure

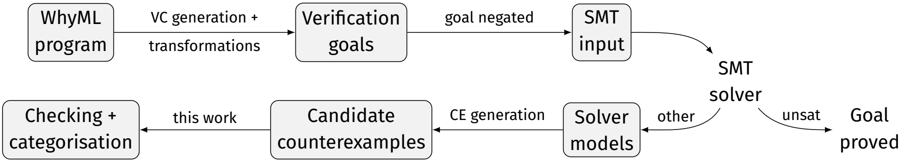
- Non-conformance
- Sub-contract weakness

Counterexamples

- `x=42, y=43`
- `x=0, y=2`

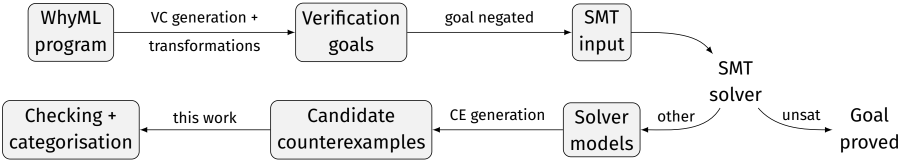
Candidate counterexample generation in Why3

(Dailler et al., 2018)



Candidate counterexample generation in Why3

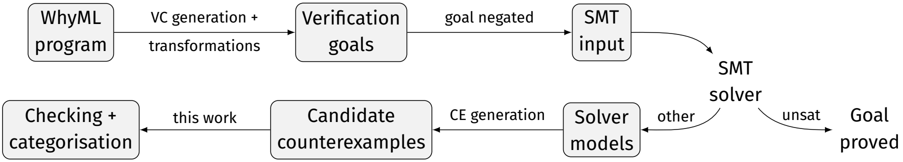
(Dailler et al., 2018)



- no guarantee on the validity of the solver models
  → potentially bad counterexamples
- no hints on the reason of the proof failure

Candidate counterexample generation in Why3

(Dailler et al., 2018)



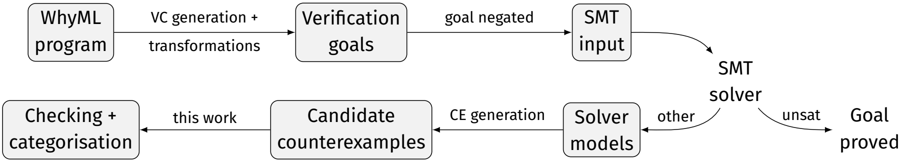
- no guarantee on the validity of the solver models
  → potentially bad counterexamples
- no hints on the reason of the proof failure

Objective in this presentation

- check candidate counterexamples and categorise proof failures
  using normal + *giant-step* runtime assertion checking

Candidate counterexample generation in Why3

(Dailler et al., 2018)



- no guarantee on the validity of the solver models
  → potentially bad counterexamples
- no hints on the reason of the proof failure

Objective in this presentation

- check candidate counterexamples and categorise proof failures using normal + *giant-step* runtime assertion checking
- inspired by Petiot et al. (2018): *How testing helps to diagnose proof failures.*

Outline

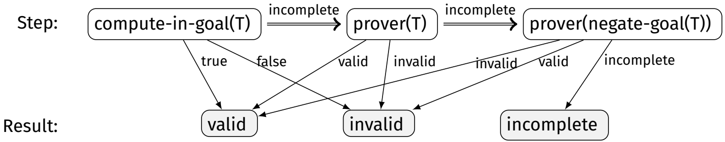
1. Runtime assertion checking in Why3
2. *Giant-step* runtime assertion checking
3. Validation of counterexamples and categorisation of proof failures

Normal runtime assertion checking

- normal program execution, validity of annotations are checked
- invalid annotations terminate execution
  - Failure for assertions
  - Stuck for assumptions

Normal runtime assertion checking

- normal program execution, validity of annotations are checked
- invalid annotations terminate execution
  - Failure for assertions
  - Stuck for assumptions
- Why3's annotation language is not executable
  1. steps to check an annotation



  2. "incomplete" may or may not terminate execution (configurable)
  3. checked annotations as preconditions for subsequent checks

Runtime assertion checking of a counterexample

```
let main1 (x: int)
= let y = x + 1 in
  assert { y <> 43 }
```

▷ counterexample: `x=42`

Preparation

1. find program function from where the verification goal originates
2. initialise arguments for initial function call and global variables with values from counterexample

Runtime assertion checking of a counterexample

```
let main1 (x: int)
= let y = x + 1 in
  assert { y <> 43 }
```

- counterexample: `x=42`
  - normal RAC: `main 42` ⇝ Failure

Preparation

1. find program function from where the verification goal originates
2. initialise arguments for initial function call and global variables with values from counterexample

Intermediate result

- failure in normal RAC ⇒ non-conformance in program

But how to identify a sub-contract weakness?

Giant-step runtime assertion checking

Deductive program verification is modular

- from the outside, function and loops are defined by their post-condition and invariants (sub-contracts), not their bodies
- counterexamples values for function calls and loops comply to the sub-contracts (usually!)

Giant-step runtime assertion checking

Deductive program verification is modular

- from the outside, function and loops are defined by their post-condition and invariants (sub-contracts), not their bodies
- counterexamples values for function calls and loops comply to the sub-contracts (usually!)

Idea of giant-step RAC: like normal RAC but

- don't execute function bodies, don't iterate loop bodies
- retrieve return values and values of written variables from oracle

Giant-step runtime assertion checking

Function calls

RAC execution of a function call $f\ v_1 \cdots v_n$ at location p
in environment Γ, with

```
let f x₁...xₙ writes { y₁,...,yₘ }
    requires { φ_pre } ensures { φ_post } = e
```

1. bind arguments to parameters $\Gamma_1 := \Gamma[\ldots, x_i \leftarrow v_i, \ldots]$
2. assert pre-conditions $\Gamma_1 \vdash \phi_{\text{pre}}$
3. normal RAC:
   evaluate body e to result value v,
   modifying written variables by side-effect $(v, \Gamma_2) := eval(e, \Gamma_1)$
4. assert post-conditions $\Gamma_2[\text{result} \leftarrow v] \vdash \phi_{\text{post}}$
5. return value v (v, Γ_2)

Giant-step runtime assertion checking

Function calls

**

RAC execution of a function call $f\ v_1 \cdots v_n$ at location p in environment Γ and oracle Ω, with

$$\texttt{let}\ f\ x_1 \ldots x_n\ \texttt{writes}\ \{\ y_1, \ldots, y_m\ \}$$
$$\quad\texttt{requires}\ \{\ \phi_{\mathsf{pre}}\ \}\ \texttt{ensures}\ \{\ \phi_{\mathsf{post}}\ \}\ \texttt{=}\ e$$

1. bind arguments to parameters — $\Gamma_1 := \Gamma[\ldots, x_i \leftarrow v_i, \ldots]$
2. assert pre-conditions — $\Gamma_1 \vdash \phi_{\mathsf{pre}}$
3. giant-step RAC:
   retrieve result value v and — $v = \Omega(\mathsf{result}, p)$
   update written variables from oracle — $\Gamma_2 := \Gamma_1[\ldots, y_i \leftarrow \Omega(y_i, p), \ldots]$
4. assume post-conditions — $\Gamma_2[\mathsf{result} \leftarrow v] \vdash \phi_{\mathsf{post}}$
5. return value v — (v, Γ_2)

Giant-step runtime assertion checking

While loops

**

RAC execution of a while loop at location p in environment Γ and oracle Ω:

```
while e₁ writes { y₁,...,yₙ }
  invariant { ϕ_inv } do e₂ done
```

1. assert invariant (initialisation) $\Gamma \vdash \phi_{\mathsf{inv}}$
2. giant-step:
   - ▷ update written variables from oracle $\Gamma_1 := \Gamma[\ldots, y_i \leftarrow \Omega(y_i, p), \ldots]$
   - ▷ assume invariant $\Gamma_1 \vdash \phi_{\mathsf{inv}}$
3. if condition e_1 is true $(\mathit{true}, \Gamma_2) := \mathit{eval}(e_1, \Gamma_1)$
   - ▷ evaluate loop body e_2 $((), \Gamma_3) := \mathit{eval}(e_1, \Gamma_2)$
   - ▷ assert invariant (preservation) $\Gamma_3 \vdash \phi_{\mathsf{inv}}$
   - ▷ stuck
4. else done $((), \Gamma_2)$

Identification of a sub-contract weakness

Giant-step RAC of a counterexample

```
let f (x: int) : int
  ensures { result > x }
= x + 1
```

```
let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

- counterexample: `x=0`, `y=2`

- find program function from where the verification goal originates
- two executions: normal RAC and giant-step RAC
- counterexample as oracle for
  - initial values of global variables + arguments for initial function call
  - written variables and return values in giant-step RAC

Identification of a sub-contract weakness

Giant-step RAC of a counterexample

```
let f (x: int) : int
  ensures { result > x }
= x + 1
```

```
let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

- counterexample: `x=0`, `y=2`
  - normal RAC: `main2 0` ⇝ Normal termination
- find program function from where the verification goal originates
- two executions: normal RAC and giant-step RAC
- counterexample as oracle for
  - initial values of global variables + arguments for initial function call
  - written variables and return values in giant-step RAC

Identification of a sub-contract weakness

Giant-step RAC of a counterexample

```
let f (x: int) : int
  ensures { result > x }
= x + 1
```

```
let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

- counterexample: `x=0`, `y=2`
  - normal RAC: `main2 0` ⇝ Normal termination
  - giant-step RAC: `main2 0`, `f x = 2` ⇝ Failure

- find program function from where the verification goal originates
- two executions: normal RAC and giant-step RAC
- counterexample as oracle for
  - initial values of global variables + arguments for initial function call
  - written variables and return values in giant-step RAC

Classification of candidate counterexamples (CE)

Normal RAC	Giant-step RAC			
	Failure	Normal	Stuck	Incomplete
Failure matches goal	Non-conformity			
Failure elsewhere	Bad CE (invalid assertion elsewhere)			
Stuck	Invalid assumption			
Normal	Sub-contract weakness	Bad CE (no failure)	Bad CE (bad values)	Incomplete
Incomplete	Non-conformity or sub-contract weakness	Incomplete	Bad CE (bad values)	Incomplete

Examples in Why3

(Adapted from Petiot et al., 2018)

```
use int.Int, lib.IntRef

let isqrt (n: int)
  requires { 0 <= n }
  ensures { result * result <= n <
              (result + 1) * (result + 1) }
= let r = ref n in
  let y = ref (n * n) in
  let z = ref (-2 * n + 1) in
  while !y > n do
    invariant { 0 <= !r <= n }
    invariant { !y = !r * !r }
    invariant { n < (!r+1) * (!r+1) }
    invariant { !z = -2 * !r + 1 }
    variant { !r }
    y := !y + !z;
    z := !z + 2;
    r := !r  1
  done;
  !r
```

```
$ bin/why3 prove -P z3 -L . isqrt.mlw
File isqrt.mlw:
Goal isqrt'vc.
Prover result is: Valid.
```

Examples in Why3

Non-conformity

```
1  use int.Int, lib.IntRef
2
3  let isqrt (n: int)
4    requires { 0 <= n }
5    ensures { result * result <= n <
6               (result + 1) * (result + 1) }
7  = let r = ref n in
8    let y = ref (n * n) in
9    let z = ref (-2 * n + 1) in
10   while !y > n do
11     invariant { 0 <= !r <= n }
12     invariant { !y = !r * !r }
13     invariant { n < (!r+1) * (!r+1) }
14     invariant { !z = -2 * !r + 1 }
15     variant { !r }
16     y := !y - !z;
17     z := !z + 2;
18     r := !r  1
19   done;
20   !r
```

```
$ why3 prove -a split_vc -P cvc4-ce -L . isqrt.mlw \
    --check-ce --rac-prover=cvc4 --rac-try-negate
File "isqrt.mlw", line 12, characters 19-31:
Sub-goal Loop invariant preservation of goal isqrt'vc
Prover result is: Unknown
The program does not comply to the verification
  goal, for example during the following execution:
- call main function ‘isqrt‘ with args: 4
- call function ‘ref‘ with args: 4
- call function ‘( * )‘ with args: 4, 4
- call function ‘ref‘ with args: 16
- call function ‘( * )‘ with args: -2, 4
- call function ‘(+)‘ with args: -8, 1
- call function ‘ref‘ with args: -7
- call function ‘(>)‘ with args: 16, 4
- iterate loop:
- call function ‘(-)‘ with args: 16, -7
- call function ‘(:=)‘ with args: y, 23
- call function ‘(+)‘ with args: -7, 2
- call function ‘(:=)‘ with args: z, -5
- call function ‘(-)‘ with args: 4, 1
- call function ‘(:=)‘ with args: r, 3
- failure at loop invariant preservation with:
    !r = 3, !y = 23
```

Examples in Why3

Sub-contract weakness

```
1  use int.Int, lib.IntRef
2
3  let isqrt (n: int)
4    requires { 0 <= n }
5    ensures { result * result <= n <
6              (result + 1) * (result + 1) }
7  = let r = ref n in
8    let y = ref (n * n) in
9    let z = ref (-2 * n + 1) in
10   while !y > n do
11     invariant { 0 <= !r <= n }
12     invariant { !y = !r * !r }
13     invariant {  true  }
14     invariant { !z = -2 * !r + 1 }
15     variant { !r }
16     y := !y + !z;
17     z := !z + 2;
18     r := !r  1
19   done;
20   !r
```

```
$ why3 prove -a split_vc -P cvc4-ce -L . isqrt.mlw \
    --check-ce --rac-prover=cvc4 --rac-try-negate
File "isqrt.mlw", line 6, characters 12-62:
Sub-goal Postcondition of goal isqrt'vc.
Prover result is: Unknown
The contracts of some function or loop are under-
  specified, for example:
- call main function ‘isqrt‘ with args: 1
- giant-step call function ‘int_ref‘ with args: 1
    -> {contents= 1}
- call function ‘( * )‘ with args: 1, 1
- giant-step call function ‘ref‘ with args: 1
    -> {contents= 1}
- call function ‘( * )‘ with args: -2, 1
- call function ‘(+)‘ with args: -2, 1
- giant-step call function ‘ref‘ with args: -1
    -> {contents= (-1)}
- giant-step iterate loop with: r:=0, y:=0, z:=1
- giant-step call function ‘(!)‘ with args: y -> 0
- call function ‘(>)‘ with args: 0, 1
- exit loop
- giant-step call function ‘(!)‘ with args: r -> 0
- failure at postcondition of ‘isqrt‘ with:
     n = 1, result = 0
```

Future work

- identifying single sub-contract weaknesses
- integration with other language front-ends of Why3, e.g. Ada/SPARK
- dealing with incomplete oracles

Thank you.

Questions?

* M. v. Schwind, 1845.

** H. Leutemann, 1856.

Examples in Why3

```
(* ex1.mlw *)
use int.Int
let main1 (x: int)
= let y = x + 1 in
  assert { y <> 43 }
```

```
$ why3 prove -a split_vc -P cvc4-ce ex1.mlw \
    --check-ce --rac-prover=cvc4 --rac-try-negate
Sub-goal Assertion of goal main1'vc.
Prover result is: Unknown
The program does not comply to the verification
  goal, for example during the execution:
File ex1.mlw:
  Line 2:
    Execution of main function ‘main1‘ with args:
      x = 42
  Line 3:
    Execution of function ‘(+)‘
      with args: 42, 1
  Line 4:
    Property failure at assertion with: y = 43
```

```
(* ex2.mlw *)
use int.Int
let f (x: int) : int
  ensures { result > x }
= x + 1
let main2 (x: int)
= let y = f x in
  assert { y = x + 1 }
```

```
$ why3 prove -a split_vc -P cvc4-ce ex2.mlw \
    --check-ce --rac-prover=cvc4 --rac-try-negate
Sub-goal Assertion of goal main2'vc.
Prover result is: Unknown
The contracts of some function or loop are under-
  specified, for example during the execution:
File ex2.mlw:
  Line 5:
    Execution of main function ‘main2‘ with args:
      x = 0
  Line 6:
    Giant-step execution of function ‘f‘
      with args: x = 1
    result of ‘f‘ = 2
  Line 7:
    Property failure at assertion
      with: x = 0, y = 2
```

The μWhy language

p	::=	$d_1 \cdots d_n$	program
d	::=	`var` x `:` τ `=` e	global variable declaration
	|	`fun` f $(x_1 : \tau_1)$ `...` $(x_n : \tau_n)$ `:` τ	function declaration
		`requires` { ϕ_{pre} } `ensures` { ϕ_{post} } `writes` { $y_1, \ldots, y_k$ } `=` e	
τ	::=	`bool` | `int` | `unit`	type
ϕ	::=	$\top \mid \bot \mid t_1\ op\ t_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid$	
		$\forall x : \tau.\phi \mid \exists x : \tau.\phi$	formula
t	::=	$l \mid x \mid t_1\ op\ t_2$	pure term
l	::=	`()` | `true` | `false` | `0` | `1` | $\cdots$	literal
e	::=	t	pure expression
	|	$x \leftarrow e$	assignment
	|	`var` $x : \tau$ `=` e_1 `in` e_2	local binding
	|	`if` e_1 `then` e_2 `else` e_3	conditional
	|	`assert` { ϕ }	assertion
	|	`stuck`	diverging statement
	|	`while` e_1 `do invariant` { ϕ_{inv} } `writes` { $y_1, \ldots, y_k$ } e_2 `done`	loop
	|	$f\ x_1 \cdots x_n$	function call

Small-step runtime assertion checking

- semantic judgement with global and local variable environment Γ, Π

$$\Gamma, \Pi, e \rightsquigarrow \Gamma', \Pi', e' \qquad \text{(execution step)}$$

$$\Gamma, \Pi, e \ \lightning\ \xi \qquad \text{(execution stuck, } \xi \in \{\text{Failure}, \text{Stuck}\}\text{)}$$

- semantic rules

$$\textsc{Global-Variable}\quad \frac{\Gamma(x) = v}{\Gamma, \Pi, x \rightsquigarrow \Gamma, \Pi, v} \qquad \textsc{Local-Variable}\quad \frac{\Pi(x) = v}{\Gamma, \Pi, x \rightsquigarrow \Gamma, \Pi, v}$$

$$\textsc{Local-Variable-binding}\quad \frac{}{\Gamma, \Pi, \texttt{var}\ x = v\ \texttt{in}\ e \rightsquigarrow \Gamma, (x, v) \cdot \Pi, e}$$

$$\textsc{Global-Variable-assignment}\quad \frac{x \in dom(\Gamma)}{\Gamma, \Pi, x \leftarrow v \rightsquigarrow \Gamma[x \leftarrow v], \Pi, ()}$$

$$\textsc{Local-Variable-assignment}\quad \frac{x \in dom(\Pi)}{\Gamma, \Pi, x \leftarrow v \rightsquigarrow \Gamma, \Pi[x \leftarrow v], ()}$$

$$\textsc{Conditional-true}\quad \frac{}{\Gamma, \Pi, \texttt{if true then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \Gamma, \Pi, e_2}$$

$$\textsc{Conditional-false}\quad \frac{}{\Gamma, \Pi, \texttt{if false then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \Gamma, \Pi, e_3}$$

$$\textsc{Assertion-valid}\quad \frac{\Gamma, \Pi \vdash t}{\Gamma, \Pi, \texttt{assert}\ \{\ t\ \} \rightsquigarrow \Gamma, \Pi, ()}$$

Small-step runtime assertion checking

More rules

While-iterate

$$\frac{\Gamma, \Pi \vdash \phi_{inv}}{\begin{array}{c}\Gamma, \Pi, \texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done} \rightsquigarrow \\ \Gamma, \Pi, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{while}\ c\ \texttt{do invariant \{}\ \phi_{inv}\ \texttt{\} writes \{}\ \vec{y}\ \texttt{\}}\ e\ \texttt{done})\ \texttt{else}\ ()\end{array}}$$

Call

$$\frac{\Gamma(f) = \mathsf{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \qquad \Gamma, \Pi_2 \vdash \phi_{pre}}{\Gamma, \Pi_1, (f\ z_1 \cdots z_n) \rightsquigarrow \Gamma, \Pi_1, \mathsf{CallFrame}(\Pi_2, e_{body}, \phi_{post})}$$

CallFrame-execution

$$\frac{\Gamma_1, \Pi_1, e_1 \rightsquigarrow \Gamma_2, \Pi_2, e_2}{\Gamma_1, \Pi, \mathsf{CallFrame}(\Pi_1, e_1, \phi_{post}) \rightsquigarrow \Gamma_2, \Pi, \mathsf{CallFrame}(\Pi_2, e_2, \phi_{post})}$$

Return

$$\frac{\Gamma, \Pi_2[\mathsf{result} \leftarrow v] \vdash \phi_{post}}{\Gamma, \Pi_1, \mathsf{CallFrame}(\Pi_2, v, \phi_{post}) \rightsquigarrow \Gamma, \Pi_1, v}$$

Small-step runtime assertion checking

Blocking rules

$$\textsc{Assertion-invalid}\quad \frac{\Gamma, \Pi \nvDash \phi}{\Gamma, \Pi, \texttt{assert \{ } \phi \texttt{ \}} \rightsquigarrow \mathsf{Failure}}$$

$$\textsc{Stuck}\quad \frac{}{\Gamma, \Pi, \texttt{stuck} \rightsquigarrow \mathsf{Stuck}}$$

$$\textsc{While-invariant-failure}\quad \frac{\Gamma, \Pi \nvDash \phi_{inv}}{\Gamma, \Pi, \texttt{while } c \texttt{ do invariant \{ } \phi_{inv} \texttt{ \} writes \{ } \vec{y} \texttt{ \} } e \texttt{ done} \rightsquigarrow \mathsf{Failure}}$$

$$\textsc{Call-precondition-failure}\quad \frac{\Gamma(f) = \mathsf{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \qquad \Gamma, \Pi_2 \nvDash \phi_{pre}}{\Gamma, \Pi_1, (f\ z_1 \cdots z_n) \rightsquigarrow \mathsf{Failure}}$$

$$\textsc{Return-postcondition-failure}\quad \frac{\Gamma, \Pi_2[\mathsf{result} \leftarrow v] \nvDash \phi_{post}}{\Gamma, \Pi_1, \mathsf{CallFrame}(\Pi_2, v, \phi_{post}) \rightsquigarrow \mathsf{Failure}}$$

Giant-step semantics

Semantics

- semantic judgement with oracle O : *Pos* $\times$ *Ident* $\rightarrow$ *Value*

$$\Gamma, \Pi, e \overset{O}{\rightsquigarrow} \Gamma', \Pi', e'$$

$$\Gamma, \Pi, e \ \wr_O \ \xi$$

- giant-step rules for loops

While-invariant-initialisation-failure

$$\frac{\Gamma, \Pi \nvdash \phi_{inv}}{\Gamma, \Pi, \texttt{while}\ c\ \texttt{do invariant}\ \{\ \phi_{inv}\ \}\ \texttt{writes}\ \{\ \vec{y}\ \}\ e\ \texttt{done}\ \wr_O\ \text{Failure}}$$

While-any-iteration-stuck

$$\frac{\Gamma_1, \Pi_1 \vdash \phi_{inv} \qquad (\Gamma_2, \Pi_2) = (\Gamma_1, \Pi_1)[y_i \leftarrow O(p, y_i)]_{1 \leq i \leq k} \qquad \Gamma_2, \Pi_2 \nvdash \phi_{inv}}{\Gamma_1, \Pi_1, [p]\texttt{while}\ c\ \texttt{do invariant}\ \{\ \phi_{inv}\ \}\ \texttt{writes}\ \{\ \vec{y}\ \}\ e\ \texttt{done}\ \wr_O\ \text{Stuck}}$$

While-any-iteration

$$\frac{\Gamma_1, \Pi_1 \vdash \phi_{inv} \qquad (\Gamma_2, \Pi_2) = (\Gamma_1, \Pi_1)[y_i \leftarrow O(p, y_i)]_{1 \leq i \leq k} \qquad \Gamma_2, \Pi_2 \vdash \phi_{inv}}{\begin{array}{c}\Gamma_1, \Pi_1, [p]\texttt{while}\ c\ \texttt{do invariant}\ \{\ \phi_{inv}\ \}\ \texttt{writes}\ \{\ \vec{y}\ \}\ e\ \texttt{done} \overset{O}{\rightsquigarrow} \\ \Gamma_2, \Pi_2, \texttt{if}\ c\ \texttt{then}\ (e; \texttt{assert}\ \{\ \phi_{inv}\ \}; \texttt{stuck})\ \texttt{else}\ ()\end{array}}$$

Giant-step semantics

Call-precondition-failure

$$\frac{\Gamma_1(f) = \mathsf{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \qquad \Gamma_1, \Pi_2 \nvdash \phi_{pre}}{\Gamma_1, \Pi_1, (f\ z_1 \cdots z_n) \ \lightning_O\ \mathsf{Failure}}$$

Call-postcondition-stuck

$$\frac{\begin{array}{c}\Gamma_1(f) = \mathsf{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \\ \Gamma_1, \Pi_2 \vdash \phi_{pre} \qquad \Gamma_2 = \Gamma_1[y_i \leftarrow O(p, y_i)]_{1 \le i \le m} \qquad v = O(p, \mathsf{result}) \qquad \Gamma_2, \{\mathsf{result} \leftarrow v\} \cdot \Pi_2 \nvdash \phi_{post}\end{array}}{\Gamma_1, \Pi_1, [p](f\ z_1\ \ldots\ z_n) \ \lightning_O\ \mathsf{Stuck}}$$

Call-success

$$\frac{\begin{array}{c}\Gamma_1(f) = \mathsf{Func}(\vec{x}, \phi_{pre}, \phi_{post}, \vec{y}, e_{body}) \qquad \Pi_2 = \{x_i \leftarrow \Gamma_1 \oplus \Pi_1(z_i)\}_{1 \le i \le n} \\ \Gamma_1, \Pi_2 \vdash \phi_{pre} \qquad \Gamma_2 = \Gamma_1[y_i \leftarrow O(p, y_i)]_{1 \le i \le m} \qquad v = O(p, \mathsf{result}) \qquad \Gamma_2, \{\mathsf{result} \leftarrow v\} \cdot \Pi_2 \vdash \phi_{post}\end{array}}{\Gamma_1, \Pi_1, [p](f\ z_1\ \ldots\ z_n) \overset{O}{\rightsquigarrow} \Gamma_2, \Pi_1, v}$$