Verification Conditions: from Why3 to Coma

Andrei Paskevich

LMF, Université Paris-Saclay • Toccata, Inria Saclay

IFIP WG 1.9/2.15 • April 15, 2026

weakest preconditions are beautiful

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q) &\equiv Q \\
\mathrm{WP}(x \leftarrow t, Q) &\equiv Q[x \mapsto t] \\
\mathrm{WP}(e_1 \,;\, e_2, Q) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)) \\
\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) &\equiv \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q) \\
&\qquad\qquad \texttt{else } \mathrm{WP}(e_2, Q) \\
\mathrm{WP}(\texttt{while } t \texttt{ invt } J \texttt{ do } e \texttt{ done}, Q) &\equiv J \wedge \\
&\quad \forall \bar{x}.\, J \rightarrow \texttt{if } t \texttt{ then } \mathrm{WP}(e, J) \\
&\qquad\qquad\qquad \texttt{else } Q
\end{aligned}
$$

- simple rules • readable VCs • on-the-fly CPS transformation
- free variables of $\mathrm{WP}(e, Q)$ capture *precisely* the initial memory state
- \+ function calls, forward jumps, exceptions, variable binding, pattern matching, etc.

at night, the aliases come

The standard WP rule for assignment:

$$\mathrm{WP}(x \leftarrow 42, Q[x, y, z]) \;=\; Q[42, y, z]$$

But if x and z are two names for the same memory location

$$\mathrm{WP}(x \leftarrow 42, Q[x, y, z]) \quad \text{should be} \quad Q[42, y, 42]$$

The standard WP rule for assignment:

$$\mathrm{WP}(x \leftarrow 42, Q[x, y, z]) \;=\; Q[42, y, z]$$

But if x and z are two names for the same memory location

$$\mathrm{WP}(x \leftarrow 42, Q[x, y, z]) \quad \text{should be} \quad Q[42, y, 42]$$

Challenge: know, *statically*, when two references are aliased

Solutions: explicit locations — MM, component-as-array, type-as-array
mutability XOR aliasing — SL, permissions, borrowing

The standard WP rule for assignment:

$$\mathrm{WP}(x \leftarrow 42, Q[x, y, z]) \;=\; Q[42, y, z]$$

But if x and z are two names for the same memory location

$$\mathrm{WP}(x \leftarrow 42, Q[x, y, z]) \quad \text{should be} \quad Q[42, y, 42]$$

Challenge: know, *statically*, when two references are aliased

Solutions: explicit locations — MM, component-as-array, type-as-array
mutability XOR aliasing — SL, permissions, borrowing

Why3: locations-as-types

Every mutable type carries an *invisible identity token* — a region identifier:

$$x : \texttt{ref}\ \varrho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \varrho\ \texttt{int}$$

Every mutable type carries an *invisible identity token* — a region identifier:

$x : \texttt{ref}\ \varrho\ \texttt{int}$ $y : \texttt{ref}\ \pi\ \texttt{int}$ $z : \texttt{ref}\ \varrho\ \texttt{int}$

Now, some programs no longer typecheck: `if ... then` x `else` y : ?

Every mutable type carries an *invisible identity token* — a region identifier:

$$x : \texttt{ref}\ \varrho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \varrho\ \texttt{int}$$

Now, some programs no longer typecheck: $\texttt{if}\ \ldots\ \texttt{then}\ x\ \texttt{else}\ y : ?$

...fortunately: $\mathrm{WP}(\texttt{let}\ r = x\ \textit{or maybe}\ y\ \texttt{in}\ r \leftarrow 42, Q[x, y, z]) = ?$

Every mutable type carries an *invisible identity token* — a region identifier:

$$x : \texttt{ref}\ \varrho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \varrho\ \texttt{int}$$

Now, some programs no longer typecheck: $\texttt{if}\ \ldots\ \texttt{then}\ x\ \texttt{else}\ y : ?$

...fortunately: $\mathrm{WP}(\texttt{let}\ r = x\ \textit{or maybe}\ y\ \texttt{in}\ r \leftarrow 42, Q[x, y, z]) = ?$

ML-style type inference reveals the identity of every mutable value

- no additional annotations required in the source code
- function parameters, global references are assumed to be separated

Every mutable type carries an *invisible identity token* — a region identifier:

$$x : \texttt{ref}\ \varrho\ \texttt{int} \qquad y : \texttt{ref}\ \pi\ \texttt{int} \qquad z : \texttt{ref}\ \varrho\ \texttt{int}$$

Now, some programs no longer typecheck: $\texttt{if}\ \ldots\ \texttt{then}\ x\ \texttt{else}\ y : ?$

...fortunately: $\mathrm{WP}(\texttt{let}\ r = x\ \textit{or maybe}\ y\ \texttt{in}\ r \leftarrow 42,\ Q[x, y, z]) = ?$

ML-style type inference reveals the identity of every mutable value

- no additional annotations required in the source code
- function parameters, global references are assumed to be separated

Revised WP rule for assignment: $\mathrm{WP}(x_\tau \leftarrow t, Q) = Q\sigma$

where σ replaces in Q every variable $y : \pi[\tau]$ with a reassembled value

- an alias of x can be stored inside a reference inside a record inside a tuple
- **however**: no mutable components inside recursive or abstract data structures

Poor man's resizable array:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
```

Poor man's resizable array:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
```

Let us resize it:

```
let olda = !resa (* olda : array ϱ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa := newa (* newa : array ϱ2 int *)
```

Poor man's resizable array:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
```

Let us resize it:

```
let olda = !resa (* olda : array ϱ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa := newa (* newa : array ϱ2 int *)
```

Type mismatch: We break the regions ↔ aliases correspondence!

Poor man's resizable array:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
```

Let us resize it:

```
let olda = !resa (* olda : array ϱ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa := newa (* newa : array ϱ2 int *)
```

Type mismatch: We break the regions ↔ aliases correspondence!

Change the type of `resa`? What about `if ... then resa := newa`?

Let everybody keep their type:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
let olda = !resa (* olda : array ϱ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ϱ2 int *)
```

`newa`, `olda` — the witnesses of the type system corruption

Let everybody keep their type:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
let olda = !resa (* olda : array ϱ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ϱ2 int *)
```

`newa`, `olda` — the witnesses of the type system corruption

What do we do with undesirable witnesses? — A.G. Capone

Let everybody keep their type:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
let olda = !resa (* olda : array ϱ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ϱ2 int *)
```

Type-changing expressions have a special effect:

$$\texttt{writes}\ \varrho \cdot \texttt{resets}\ \varrho_1, \varrho_2$$

e_1 ; e_2 is well-typed $\Longrightarrow$ in every free variable of e_2, every region reset by e_1 occurs under a region written by e_1

Let everybody keep their type:

```
let resa = ref (Array.make 10 0) in
 (* resa : ref ϱ (array ϱ1 int) *)
let olda = !resa (* olda : array ϱ1 int *) in
let newa = Array.make (2 * length olda) 0 in
Array.blit olda 0 newa 0 (length olda);
resa.contents ← newa (* newa : array ϱ2 int *)
```

Type-changing expressions have a special effect:

$$\texttt{writes}\ \varrho \cdot \texttt{resets}\ \varrho_1, \varrho_2$$

e_1 ; e_2 is well-typed $\Rightarrow$ in every free variable of e_2, every region reset by e_1 occurs under a region written by e_1

Thus: `resa` and its aliases survive, but `olda` and `newa` are invalidated.

$e_1 \; ; \; e_2$ is well-typed $\Longrightarrow$ in every free variable of e_2, every region reset by e_1 occurs under a region written by e_1

$e_1 \; ; \; e_2$ is well-typed $\Longrightarrow$ in every free variable of e_2, every region reset by e_1 occurs under a region written by e_1

The reset effect also expresses freshness:

If we create a fresh mutable value and give it region ϱ,
we invalidate all existing variables whose type contains ϱ.

e_1 ; e_2 is well-typed $\Rightarrow$ in every free variable of e_2, every region reset by e_1 occurs under a region written by e_1

The reset effect also expresses freshness:

> If we create a fresh mutable value and give it region ϱ,
> we invalidate all existing variables whose type contains ϱ.

Effect union (for sequence or branching):

> x_τ survives $\varepsilon_1 \sqcup \varepsilon_2$ $\Leftrightarrow$ x_τ survives both ε_1 and ε_2

$e_1 \,;\, e_2$ is well-typed $\Rightarrow$ in every free variable of e_2, every region reset by e_1 occurs under a region written by e_1

The reset effect also expresses freshness:

> If we create a fresh mutable value and give it region ϱ,
> we invalidate all existing variables whose type contains ϱ.

Effect union (for sequence or branching):

x_τ survives $\varepsilon_1 \sqcup \varepsilon_2$ $\Leftrightarrow$ x_τ survives both ε_1 and ε_2

Thus:

- the reset regions of ε_1 and ε_2 are added together,
- the written regions of ε_i reset by ε_{2-i} are invalidated.

a small matter of broken invariants

In WHY3, type invariants are expressed as axioms.

```
axiom array'invariant:
  forall a: array α. a.length ≥ 0
```

- uncluttered contracts, no problem with containers
- **however**: our assignment rule is now unsound

In WHY3, type invariants are expressed as axioms.

```
axiom array'invariant:
  forall a: array α. a.length ≥ 0
```

- uncluttered contracts, no problem with containers
- **however**: our assignment rule is now unsound

```
type hanoi_rod = { mutable disks: list int }
  invariant { (* disks are strictly increasing *) }

type hanoi = { rodA, rodB, rodC: hanoi_rod }
  invariant { (* no disk appears on two rods *) }

    let { rodA = a; rodB = b } = h in
    b.disks ← Cons (head a.disks) b.disks
```

In WHY3, type invariants are expressed as axioms.

```
axiom array'invariant:
  forall a: array α. a.length ≥ 0
```

- uncluttered contracts, no problem with containers
- **however**: our assignment rule is now unsound

```
type hanoi_rod = { mutable disks: list int }
  invariant { (* disks are strictly increasing *) }

type hanoi = { rodA, rodB, rodC: hanoi_rod }
  invariant { (* no disk appears on two rods *) }
```

$$
\begin{aligned}
&\mathrm{WP}(\texttt{let } \{\texttt{ rodA = a; rodB = b }\} \texttt{ = h in} \\
&\quad \texttt{b.disks} \leftarrow \texttt{Cons (head a.disks) b.disks}, Q) \\
&= Q[\texttt{h} \mapsto \{\texttt{ h.rodA; \{ Cons (head h.rodA.disks) h.rodB.disks \}; h.rodC }\}]
\end{aligned}
$$

We must verify the type invariant before reassembling a modified value.

However: cannot expect the invariant to hold after *every* modification.

```
type hanoi_rod = { mutable disks: list int }
  invariant { (* disks are strictly increasing *) }

type hanoi = { rodA, rodB, rodC: hanoi_rod }
  invariant { (* no disk appears on two rods *) }

    let { rodA = a; rodB = b } = h in
    b.disks ← Cons (head a.disks) b.disks;
    a.disks ← tail a.disks
```

We must verify the type invariant before reassembling a modified value.

However: cannot expect the invariant to hold after *every* modification.

```
type hanoi_rod = { mutable disks: list int }
  invariant { (* disks are strictly increasing *) }

type hanoi = { rodA, rodB, rodC: hanoi_rod }
  invariant { (* no disk appears on two rods *) }
```

$$
\begin{aligned}
&\mathrm{WP}(\texttt{let } \{\texttt{ rodA = a; rodB = b }\} = \texttt{h in} \\
&\quad \texttt{b.disks} \leftarrow \texttt{Cons (head a.disks) b.disks;} \\
&\quad \texttt{a.disks} \leftarrow \texttt{tail a.disks}, Q) \\
&= (Q[\texttt{h} \mapsto \{\, \{\texttt{ tail h.rodA.disks }\}\texttt{; h.rodB; h.rodC }\}] \\
&\quad [\texttt{h} \mapsto \{\texttt{ h.rodA; }\{\texttt{ Cons (head h.rodA.disks) h.rodB.disks }\}\texttt{; h.rodC }\}]
\end{aligned}
$$

We must verify the type invariant before reassembling a modified value.

However: cannot expect the invariant to hold after *every* modification.

```
type hanoi_rod = { mutable disks: list int }
  invariant { (* disks are strictly increasing *) }

type hanoi = { rodA, rodB, rodC: hanoi_rod }
  invariant { (* no disk appears on two rods *) }
```

$$
\begin{aligned}
&\mathrm{WP}(\texttt{let } \{\texttt{ rodA = a; rodB = b }\} = \texttt{h in}\\
&\quad \texttt{b.disks} \leftarrow \texttt{Cons (head a.disks) b.disks;}\\
&\quad \texttt{a.disks} \leftarrow \texttt{tail a.disks}, Q)\\
&= (Q[\texttt{h} \mapsto \{\,\{\texttt{ tail h.rodA.disks }\}\texttt{; h.rodB; h.rodC }\}]\\
&\quad [\texttt{h} \mapsto \{\texttt{ h.rodA; }\{\texttt{ Cons (head h.rodA.disks) h.rodB.disks }\}\texttt{; h.rodC }\}]\\
&= (Q[\texttt{h} \mapsto \{\,\{\texttt{ tail h.rodA.disks }\}\texttt{;}\\
&\qquad\qquad \{\texttt{ Cons (head h.rodA.disks) h.rodB.disks }\}\texttt{; h.rodC }\}]
\end{aligned}
$$

Let us do VC deforestation.

First, we build a glutton VC, reassembling values after every write:

```
(* let { rodA = a; rodB = b } = h in *)
let a = h.rodA and b = h.rodB in
(* b.disks ← Cons (head a.disks) b.disks; *)
let b = { disks = Cons (head a.disks) b.disks } in
let h = { rodA = h.rodA; rodB = b; rodC = h.rodC } in
(* a.disks ← tail a.disks *)
let a = { disks = tail a.disks } in
let h = { rodA = a; rodB = h.rodB; rodC = h.rodC } in
(* postcondition *)
Q
```

Let us do VC deforestation.

First, we build a glutton VC, reassembling values after every write.

Then we erase *unused* reassembled values, and track fields separately:

```
(* let { rodA = a; rodB = b } = h in *)
let a = h.rodA and b = h.rodB in
(* b.disks ← Cons (head a.disks) b.disks; *)
let b = { disks = Cons (head a.disks) b.disks } in
(* do not reassemble h, mark that h.rodB is now b *)
(* a.disks ← tail a.disks *)
let a = { disks = tail a.disks } in
let h = { rodA = a; rodB = b; rodC = h.rodC } in
(* postcondition *)
Q
```

Let us do VC deforestation.

First, we build a glutton VC, reassembling values after every write.

Then we erase *unused* reassembled values, and track fields separately:

```
(* let { rodA = a; rodB = b } = h in *)
let a = h.rodA and b = h.rodB in
(* b.disks ← Cons (head a.disks) b.disks; *)
let b = { disks = Cons (head a.disks) b.disks } in
(* do not reassemble h, mark that h.rodB is now b *)
(* a.disks ← tail a.disks *)
let a = { disks = tail a.disks } in
let h = { rodA = a; rodB = b; rodC = h.rodC } in
(* postcondition *)
Q
```

A value is used if we refer to it as a whole: pass it as an argument, return as a result or keep in the memory by the end of the function.

exercising restraint

The standard WP rules for branching duplicate the postcondition:

$$\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) \equiv \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q) \texttt{ else } \mathrm{WP}(e_2, Q)$$

Flanagan and Saxe (POPL 2001) proposed compact verification conditions:

$$\mathrm{WP}(e, Q) \equiv \mathrm{Safety}(e) \wedge \forall \bar{x}'. \mathrm{PrePost}(e) \longrightarrow Q[\bar{x} \mapsto \bar{x}']$$

The standard WP rules for branching duplicate the postcondition:

$$\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) \equiv \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q) \texttt{ else } \mathrm{WP}(e_2, Q)$$

Flanagan and Saxe (POPL 2001) proposed compact verification conditions:

$$\begin{aligned}
\mathrm{WP}(e, Q) &\equiv \mathrm{Safety}(e) \wedge \forall \bar{x}'.\, \mathrm{PrePost}(e) \longrightarrow Q[\bar{x} \mapsto \bar{x}'] \\
\mathrm{Safety}(\texttt{skip}) &\equiv \top \\
\mathrm{PrePost}(\texttt{skip}) &\equiv x_1' = x_1 \wedge \cdots \wedge x_n' = x_n
\end{aligned}$$

The standard WP rules for branching duplicate the postcondition:

$$\mathrm{WP}(\texttt{if}\ t\ \texttt{then}\ e_1\ \texttt{else}\ e_2, Q) \equiv \texttt{if}\ t\ \texttt{then}\ \mathrm{WP}(e_1, Q)\ \texttt{else}\ \mathrm{WP}(e_2, Q)$$

Flanagan and Saxe (POPL 2001) proposed compact verification conditions:

$$\begin{aligned}
\mathrm{WP}(e, Q) &\equiv \mathrm{Safety}(e) \wedge \forall \bar{x}'.\, \mathrm{PrePost}(e) \longrightarrow Q[\bar{x} \mapsto \bar{x}'] \\
\mathrm{Safety}(\texttt{skip}) &\equiv \top \\
\mathrm{PrePost}(\texttt{skip}) &\equiv x_1' = x_1 \wedge \cdots \wedge x_n' = x_n \\
\mathrm{Safety}(x \leftarrow t) &\equiv \top \\
\mathrm{PrePost}(x \leftarrow t) &\equiv x_1' = x_1 \wedge \cdots \wedge x' = t \wedge \cdots \wedge x_n' = x_n
\end{aligned}$$

The standard WP rules for branching duplicate the postcondition:

$$\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) \equiv \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q) \texttt{ else } \mathrm{WP}(e_2, Q)$$

Flanagan and Saxe (POPL 2001) proposed compact verification conditions:

$$\begin{aligned}
\mathrm{WP}(e, Q) &\equiv \mathrm{Safety}(e) \wedge \forall \bar{x}'. \mathrm{PrePost}(e) \longrightarrow Q[\bar{x} \mapsto \bar{x}'] \\
\mathrm{Safety}(\texttt{skip}) &\equiv \top \\
\mathrm{PrePost}(\texttt{skip}) &\equiv x_1' = x_1 \wedge \cdots \wedge x_n' = x_n \\
\mathrm{Safety}(x \longleftarrow t) &\equiv \top \\
\mathrm{PrePost}(x \longleftarrow t) &\equiv x_1' = x_1 \wedge \cdots \wedge x' = t \wedge \cdots \wedge x_n' = x_n \\
\mathrm{Safety}(e_1 \,;\, e_2) &\equiv \mathrm{WP}(e_1, \mathrm{Safety}(e_2)) \\
\mathrm{PrePost}(e_1 \,;\, e_2) &\equiv \exists \bar{z}. \mathrm{PrePost}(e_1)[\bar{x}' \mapsto \bar{z}] \wedge \mathrm{PrePost}(e_2)[\bar{x} \mapsto \bar{z}]
\end{aligned}$$

The standard WP rules for branching duplicate the postcondition:

$$\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) \equiv \texttt{if } t \texttt{ then } \mathrm{WP}(e_1, Q) \texttt{ else } \mathrm{WP}(e_2, Q)$$

Flanagan and Saxe (POPL 2001) proposed compact verification conditions:

$$\begin{aligned}
\mathrm{WP}(e, Q) &\equiv \mathrm{Safety}(e) \wedge \forall \bar{x}'.\, \mathrm{PrePost}(e) \longrightarrow Q[\bar{x} \mapsto \bar{x}'] \\
\mathrm{Safety}(\texttt{skip}) &\equiv \top \\
\mathrm{PrePost}(\texttt{skip}) &\equiv x_1' = x_1 \wedge \cdots \wedge x_n' = x_n \\
\mathrm{Safety}(x \leftarrow t) &\equiv \top \\
\mathrm{PrePost}(x \leftarrow t) &\equiv x_1' = x_1 \wedge \cdots \wedge x' = t \wedge \cdots \wedge x_n' = x_n \\
\mathrm{Safety}(e_1 \,;\, e_2) &\equiv \mathrm{WP}(e_1, \mathrm{Safety}(e_2)) \\
\mathrm{PrePost}(e_1 \,;\, e_2) &\equiv \exists \bar{z}.\, \mathrm{PrePost}(e_1)[\bar{x}' \mapsto \bar{z}] \wedge \mathrm{PrePost}(e_2)[\bar{x} \mapsto \bar{z}] \\
\mathrm{Safety}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2) &\equiv \texttt{if } t \texttt{ then } \mathrm{Safety}(e_1) \texttt{ else } \mathrm{Safety}(e_2) \\
\mathrm{PrePost}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2) &\equiv \texttt{if } t \texttt{ then } \mathrm{PrePost}(e_1) \texttt{ else } \mathrm{PrePost}(e_2)
\end{aligned}$$

Flanagan and Saxe (POPL 2001):

$$
\begin{aligned}
\mathrm{WP}(e, Q) &\equiv \mathrm{Safety}(e) \wedge \forall \bar{x}'.\, \mathrm{PrePost}(e) \longrightarrow Q[\bar{x} \mapsto \bar{x}'] \\
\mathrm{Safety}(\texttt{skip}) &\equiv \top \\
\mathrm{PrePost}(\texttt{skip}) &\equiv x_1' = x_1 \wedge \cdots \wedge x_n' = x_n \\
\mathrm{Safety}(x \longleftarrow t) &\equiv \top \\
\mathrm{PrePost}(x \longleftarrow t) &\equiv x_1' = x_1 \wedge \cdots \wedge x' = t \wedge \cdots \wedge x_n' = x_n \\
\mathrm{Safety}(e_1 \,;\, e_2) &\equiv \mathrm{WP}(e_1, \mathrm{Safety}(e_2)) \\
\mathrm{PrePost}(e_1 \,;\, e_2) &\equiv \exists \bar{z}.\, \mathrm{PrePost}(e_1)[\bar{x}' \mapsto \bar{z}] \wedge \mathrm{PrePost}(e_2)[\bar{x} \mapsto \bar{z}] \\
\mathrm{Safety}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2) &\equiv \texttt{if } t \texttt{ then } \mathrm{Safety}(e_1) \texttt{ else } \mathrm{Safety}(e_2) \\
\mathrm{PrePost}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2) &\equiv \texttt{if } t \texttt{ then } \mathrm{PrePost}(e_1) \texttt{ else } \mathrm{PrePost}(e_2)
\end{aligned}
$$

The 2001 paper is much more brilliant than I have time to show here:

- easily extends to multiple outcomes: $\mathrm{PrePost}_{\texttt{break}}$, $\mathrm{PrePost}_{\texttt{raise}}$, etc.
- safety is actually expressed as $\neg\mathrm{PrePost}_{\texttt{fail}}$: a single composition rule!
- **however**: variable management and effect alignment require some care.

In WHY3, we wanted to have both methods and combine them.
Translating WHYML to a small common IR seemed a good idea.

```
type kode =
  | Kcont  of int                                  (* 0: skip, N: raise *)
  | Kseq   of kode * int * kode              (* 0: sequence, N: try-with *)
  | Kpar   of kode * kode                    (* non-deterministic choice *)
  | Kif    of pvsymbol * kode * kode           (* deterministic choice *)
  | Kcase  of pvsymbol * (pattern * kode) list    (* pattern matching *)
  | Khavoc of pvsymbol option Mpv.t Mreg.t  (* writes and assignments *)
  | Klet   of pvsymbol * term * term           (* let v = t such that f *)
  | Kval   of pvsymbol list * term           (* let vl = any such that f *)
  | Kcut   of term                           (* assert: check and assume *)
  | Kstop  of term                           (* check and halt execution *)
  | Kaxiom of kode                    (* axiom-functions: assume the VC *)
  | Ktag   of ktag * kode            (* switch VCgen or mark to push up *)
```

In WHY3, we wanted to have both methods and combine them.
Translating WHYML to a small common IR seemed a good idea.

```
type kode =
  | Kcont  of int                                  (* 0: skip, N: raise *)
  | Kseq   of kode * int * kode                (* 0: sequence, N: try-with *)
  | Kpar   of kode * kode                      (* non-deterministic choice *)
  | Kif    of pvsymbol * kode * kode             (* deterministic choice *)
  | Kcase  of pvsymbol * (pattern * kode) list     (* pattern matching *)
  | Khavoc of pvsymbol option Mpv.t Mreg.t   (* writes and assignments *)
  | Klet   of pvsymbol * term * term           (* let v = t such that f *)
  | Kval   of pvsymbol list * term          (* let vl = any such that f *)
  | Kcut   of term                             (* assert: check and assume *)
  | Kstop  of term                             (* check and halt execution *)
  | Kaxiom of kode                    (* axiom-functions: assume the VC *)
  | Ktag   of ktag * kode             (* switch VCgen or mark to push up *)
```

In WHY3, we wanted to have both methods and combine them.
Translating WHYML to a small common IR seemed a good idea.

```
type kode =
  | Kcont  of int                                  (* 0: skip, N: raise *)
  | Kseq   of kode * int * kode                (* 0: sequence, N: try-with *)
  | Kpar   of kode * kode                      (* non-deterministic choice *)
  | Kif    of pvsymbol * kode * kode             (* deterministic choice *)
  | Kcase  of pvsymbol * (pattern * kode) list     (* pattern matching *)
  | Khavoc of pvsymbol option Mpv.t Mreg.t  (* writes and assignments *)
  | Klet   of pvsymbol * term * term             (* let v = t such that f *)
  | Kval   of pvsymbol list * term           (* let vl = any such that f *)
  | Kcut   of term                             (* assert: check and assume *)
  | Kstop  of term                             (* check and halt execution *)
  | Kaxiom of kode                    (* axiom-functions: assume the VC *)
  | Ktag   of ktag * kode             (* switch VCgen or mark to push up *)
```

A **`reflow`** procedure pushes outcome handlers (= continuations) up towards their calling sites (`Kcont`), resulting in simpler Safety and PrePost formulas.

midlife crisis

...

3. If normal and exceptional outcomes are treated exactly the same by VCgen, why differentiate them in WHYML? Aren't they all just continuations?

...

3. If normal and exceptional outcomes are treated exactly the same by VCgen, why differentiate them in WHYML? Aren't they all just continuations?

...

5. If WP and F&S produce equivalent VCs, why are the procedures so different? Isn't F&S just factorisation of continuations? (Also see Leino 2005.)

$$(P_1 \longrightarrow \mathrm{WP}(e, Q)) \wedge (P_2 \longrightarrow \mathrm{WP}(e, Q)) \quad \text{vs.} \quad (P_1 \vee P_2) \longrightarrow \mathrm{WP}(e, Q)$$

...

3. If normal and exceptional outcomes are treated exactly the same by VCgen, why differentiate them in WHYML? Aren't they all just continuations?

...

5. If WP and F&S produce equivalent VCs, why are the procedures so different? Isn't F&S just factorisation of continuations? (Also see Leino 2005.)

$$(P_1 \longrightarrow \mathrm{WP}(e, Q)) \wedge (P_2 \longrightarrow \mathrm{WP}(e, Q)) \quad \text{vs.} \quad (P_1 \vee P_2) \longrightarrow \mathrm{WP}(e, Q)$$

...

8. Aren't postconditions just preconditions of continuations? Aren't write effects just additional parameters of continuations? (Also see monadic translation.)

...

3. If normal and exceptional outcomes are treated exactly the same by VCgen, why differentiate them in WHYML? Aren't they all just continuations?

...

5. If WP and F&S produce equivalent VCs, why are the procedures so different? Isn't F&S just factorisation of continuations? (Also see Leino 2005.)

$$(P_1 \longrightarrow \mathrm{WP}(e, Q)) \wedge (P_2 \longrightarrow \mathrm{WP}(e, Q)) \quad \text{vs.} \quad (P_1 \vee P_2) \longrightarrow \mathrm{WP}(e, Q)$$

...

8. Aren't postconditions just preconditions of continuations? Aren't write effects just additional parameters of continuations? (Also see monadic translation.)

...

13. If an outcome of a function call has the same VC as a function definition, isn't the former just an on-the-fly definition of a continuation?

...

```
let product (a: int) (b: int) : int
= let p,q,r = ref a, ref b, ref 0 in
  while !q > 0 do
    if !q mod 2 = 1 then r := !r + !p;
    p := !p + !p;
    q := !q div 2
  done;
  !r
```

```
product (a: int) (b: int) { b ≥ 0 } (return: (c: int) { c = a·b })
= loop
  / loop [p q r] { q ≥ 0 } { p·q+r = a·b }
    = if (q > 0) next last
      / last → return r
      / next → if (q mod 2 = 1) write_r write_p
        / write_r → assign &r (r + p) write_p
        / write_p [r] → assign &p (p + p) write_q
        / write_q [p r] → assign &q (q div 2) loop
  / &p, &q, &r = a, b, 0
```

and then COMA