Correct by Construction Concurrent Programs in Idris 2

Guillaume Allais

University of Strathclyde
Glasgow, UK

March 14th 2025

BOB2025

Table of Contents

Motivation: Correct Concurrent Programs

Hoare Logic for Correct Imperative Programs

Separation Logic for Correct Concurrent Programs

Correct by Construction Concurrent Programs

About Me

Lecturer at the University of Strathclyde (Glasgow, Scotland)

Interested in:

- Generic Programming and Proving
- Meta Programming and Proof Search
- Type-Directed Partial Evaluation
- Implementations of Type Theory
- Interactive Developer Tooling

Overarching Theme: Correctness by Construction

Table of Contents

Motivation: Correct Concurrent Programs

Hoare Logic for Correct Imperative Programs

Separation Logic for Correct Concurrent Programs

Correct by Construction Concurrent Programs

Sequential work

One worker mapping the 🪚 transformation on an array.

Sequential work

One worker mapping the 🪚 transformation on an array.

Sequential work

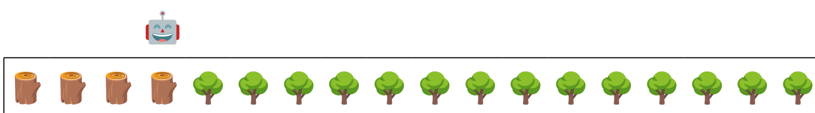
One worker mapping the 🪚 transformation on an array.

Sequential work

One worker mapping the transformation on an array.

Sequential work

One worker mapping the 🪚 transformation on an array.



Sequential work

One worker mapping the 🪚 transformation on an array.

Sequential work

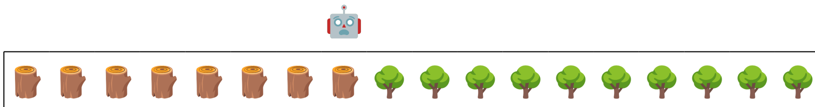
One worker mapping the 🪚 transformation on an array.

Sequential work

One worker mapping the 🪚 transformation on an array.

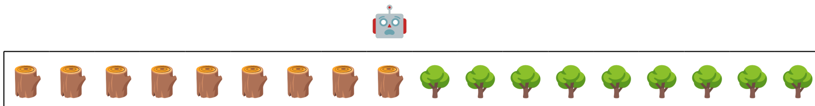
Sequential work

One worker mapping the 🪚 transformation on an array.



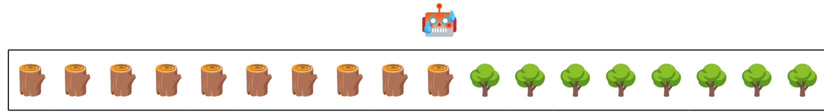
Sequential work

One worker mapping the 🪚 transformation on an array.



Sequential work

One worker mapping the 🪚 transformation on an array.



🤖 Can we maybe share the load?

Concurrent work

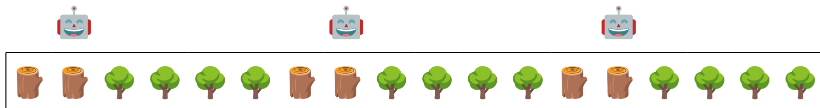
Three workers mapping the 🪚 transformation on a **shared** array.

Concurrent work

Three workers mapping the 🪚 transformation on a **shared** array.

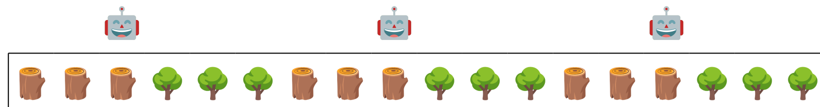
Concurrent work

Three workers mapping the 🪚 transformation on a **shared** array.



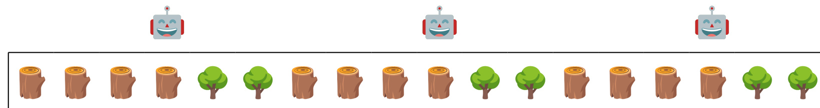
Concurrent work

Three workers mapping the 🪚 transformation on a **shared** array.



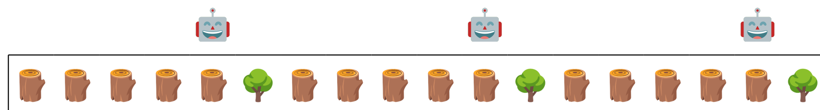
Concurrent work

Three workers mapping the 🪚 transformation on a **shared** array.



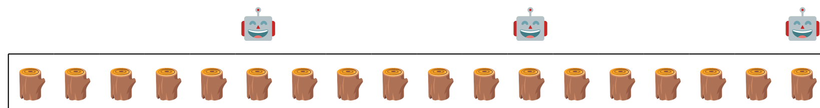
Concurrent work

Three workers mapping the 🪚 transformation on a **shared** array.



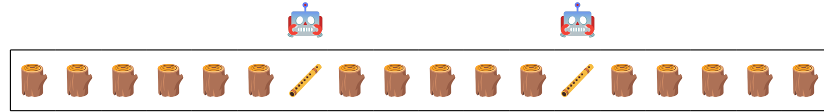
Concurrent work

Three workers mapping the 🪚 transformation on a **shared** array.



Concurrent work

Three workers mapping the 🪚 transformation on a **shared** array.



🤖 Faster... but wronger!

Table of Contents

Motivation: Correct Concurrent Programs

Hoare Logic for Correct Imperative Programs

Separation Logic for Correct Concurrent Programs

Correct by Construction Concurrent Programs

Hoare logic

A logic for imperative programs.
A memory model.
Statements of the form

$$\{P\}c\{Q\}$$

Hoare logic

A logic for imperative programs.
A memory model.
Statements of the form

$$\{P\}c\{Q\}$$

Hoare logic

A logic for imperative programs.
A memory model.
Statements of the form

Assuming that initially P holds

After executing the command c

$$\{P\}c\{Q\}$$

Hoare logic

A logic for imperative programs.
A memory model.
Statements of the form

Assuming that initially P holds

After executing the command c

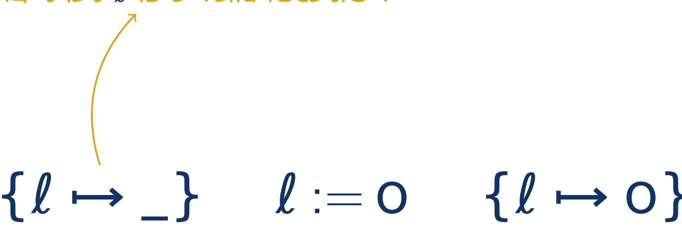
$$\{P\}c\{Q\}$$

We can guarantee that Q will hold

Assignment Axiom

$$\{\ell \mapsto _\} \quad \ell := 0 \quad \{\ell \mapsto 0\}$$

Assignment Axiom



$$\{\ell \mapsto _\} \quad \ell := 0 \quad \{\ell \mapsto 0\}$$

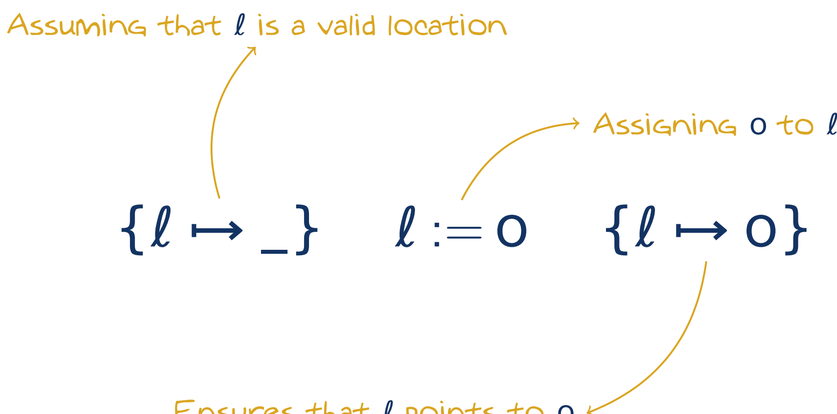
Assignment Axiom

Assuming that ℓ is a valid location

Assigning 0 to ℓ

$$\{\ell \mapsto _\} \quad \ell := 0 \quad \{\ell \mapsto 0\}$$

Assignment Axiom



Sequential Execution Axiom

$$\frac{\{P\}c_1\{Q\} \qquad \{Q\}c_2\{R\}}{\{P\}c_1;c_2\{R\}}$$

Sequential Execution Axiom

If c_1 takes us from P to Q

$$\frac{\{P\}c_1\{Q\} \qquad \{Q\}c_2\{R\}}{\{P\}c_1;c_2\{R\}}$$

Sequential Execution Axiom

If c_1 takes us from P to Q

And c_2 takes us from Q to R

$$\frac{\{P\}\, c_1 \,\{Q\} \qquad \{Q\}\, c_2 \,\{R\}}{\{P\}\, c_1; c_2 \,\{R\}}$$

Sequential Execution Axiom

If c_1 takes us from P to Q

And c_2 takes us from Q to R

$$\frac{\{P\}c_1\{Q\} \qquad \{Q\}c_2\{R\}}{\{P\}c_1;c_2\{R\}}$$

Then the composition $c_1;c_2$ takes us from P to R

A proof: swap with no allocation

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto b\}$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto b\}$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto xor(xor(a, b), b)\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \land \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \land \ell_2 \mapsto b\}$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \land \ell_2 \mapsto xor(xor(a, b), b)\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$xor(xor(a, b), b)$ equals a

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto b\}$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto a\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto b\}$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto a\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(xor(a, b), a) \wedge \ell_2 \mapsto a\}$$

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto b\}$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto a\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(xor(a, b), a) \wedge \ell_2 \mapsto a\}$$

$xor(xor(a, b), a)$ equals b

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto b\}$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto xor(a, b) \wedge \ell_2 \mapsto a\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\{\ell_1 \mapsto b \wedge \ell_2 \mapsto a\}$$

A proof: swap with no allocation

$$\{\ell_1 \mapsto a \wedge \ell_2 \mapsto b\}$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

$$\ell_2 := xor(\ell_1, \ell_2);$$

$$\ell_1 := xor(\ell_1, \ell_2);$$

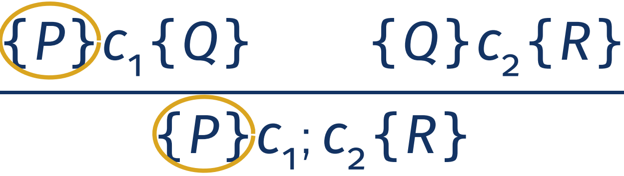
$$\{\ell_1 \mapsto b \wedge \ell_2 \mapsto a\}$$

Combining proofs

$$\frac{\{P\}c_1\{Q\} \qquad \{Q\}c_2\{R\}}{\{P\}c_1;c_2\{R\}}$$

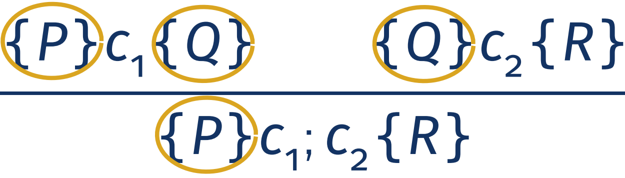
The sequential composition rule is, ironically, anti-compositional: each subprogram needs to talk about the **entire** world no matter what they **actually** use!

Combining proofs



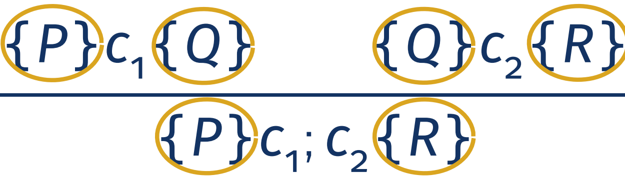
The sequential composition rule is, ironically, anti-compositional: each subprogram needs to talk about the **entire** world no matter what they **actually** use!

Combining proofs



The sequential composition rule is, ironically, anti-compositional: each subprogram needs to talk about the **entire** world no matter what they **actually** use!

Combining proofs



The sequential composition rule is, ironically, anti-compositional: each subprogram needs to talk about the **entire** world no matter what they **actually** use!

What we *want*: lifting results

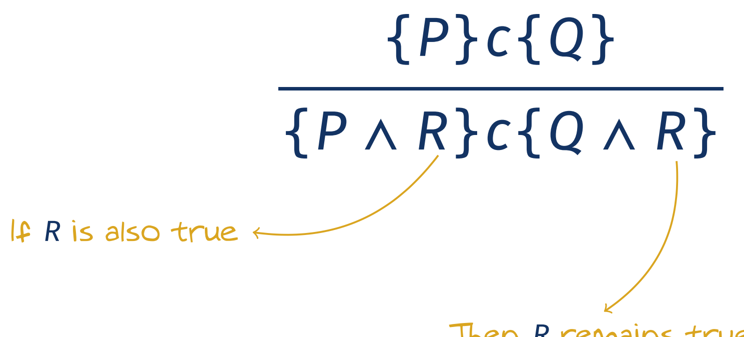
$$\frac{\{P\}c\{Q\}}{\{P \wedge R\}c\{Q \wedge R\}}$$

What we *want*: lifting results

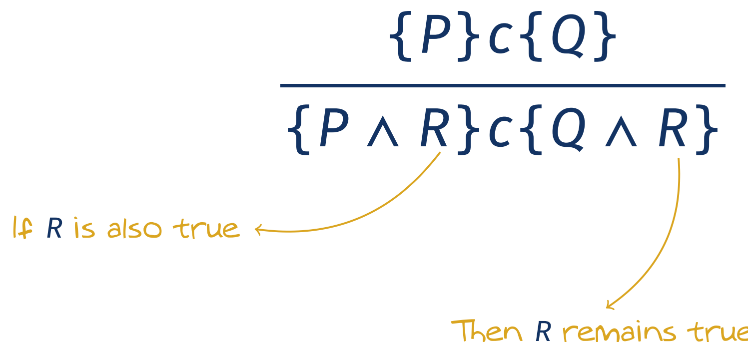
$$\frac{\{P\}c\{Q\}}{\{P \wedge R\}c\{Q \wedge R\}}$$

If R is also true

What we *want*: lifting results



What we *want*: lifting results



In a sense R is **independent** from P & Q

What we *get*: nonsense!

$$\frac{\{P\}c\{Q\}}{\{P \wedge R\}c\{Q \wedge R\}}$$

$$\{\ \ell \mapsto 1\ \}\ \ \ell := 0\ \ \{\ \ell \mapsto 0\ \}$$

What we *get*: nonsense!

$$\frac{\{P\}c\{Q\}}{\{P \wedge R\}c\{Q \wedge R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$

$$\left\{ \begin{array}{l} \qquad \ell \mapsto 1 \\ \wedge \end{array} \right\} \ \ell := 0 \ \left\{ \begin{array}{l} \qquad \ell \mapsto 0 \\ \wedge \end{array} \right\}$$

What we *get*: nonsense!

$$\frac{\{P\}c\{Q\}}{\{P \land R\}c\{Q \land R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$
We pick: $R = \ell \mapsto 1$

$$\left\{ \begin{array}{r} \ell \mapsto 1 \\ \land\, \ell \mapsto 1 \end{array} \right\} \ \ell := 0 \ \left\{ \begin{array}{r} \ell \mapsto 0 \\ \land\, \ell \mapsto 1 \end{array} \right\}$$

What we *get*: nonsense!

$$\frac{\{P\}c\{Q\}}{\{P \wedge R\}c\{Q \wedge R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$
We pick: $R = \ell \mapsto 1$

$$\left\{ \begin{array}{r} \ell \mapsto 1 \\ \wedge\, \ell \mapsto 1 \end{array} \right\} \ \ell := 0 \ \left\{ \begin{array}{r} \ell \mapsto 0 \\ \wedge\, \ell \mapsto 1 \end{array} \right\}$$

🤖 0 is equal to 1?!

What we *get*: nonsense!

$$\frac{\{P\}c\{Q\}}{\{P \land R\}c\{Q \land R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$
We pick: $R = \ell \mapsto 1$

$$\left\{ \begin{array}{r} \ell \mapsto 1 \\ \land\ \ell \mapsto 1 \end{array} \right\} \ \ell := 0 \ \left\{ \begin{array}{r} \ell \mapsto 0 \\ \land\ \ell \mapsto 1 \end{array} \right\}$$

🤖 0 is equal to 1?!

Nothing actually enforces that R is **independent** from P & Q!

Table of Contents

Motivation: Correct Concurrent Programs

Hoare Logic for Correct Imperative Programs

Separation Logic for Correct Concurrent Programs

Correct by Construction Concurrent Programs

What we *build*: a separating conjunction

- Make predicates support-aware
- Disallow claims over overlapping memory regions

	Predicate	Support

What we *build*: a separating conjunction

- Make predicates support-aware
- Disallow claims over overlapping memory regions

	Predicate	Support
Purely logical	$m + n = 3$	

What we *build*: a separating conjunction

- Make predicates support-aware
- Disallow claims over overlapping memory regions

	Predicate	Support
Purely logical	$m + n = 3$	
Points to	$\ell \mapsto v$	■

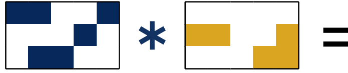
What we *build*: a separating conjunction

- Make predicates support-aware
- Disallow claims over overlapping memory regions

	Predicate	Support
Purely logical	$m + n = 3$	
Points to	$\ell \mapsto v$	■
Conjunction	$P * Q$	?

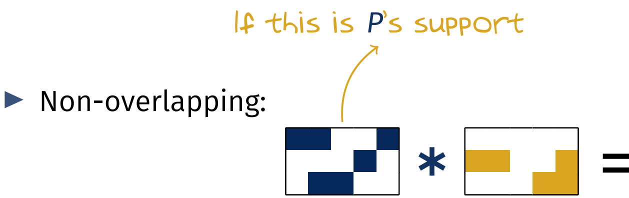
What we *build*: a separating conjunction

- Non-overlapping:



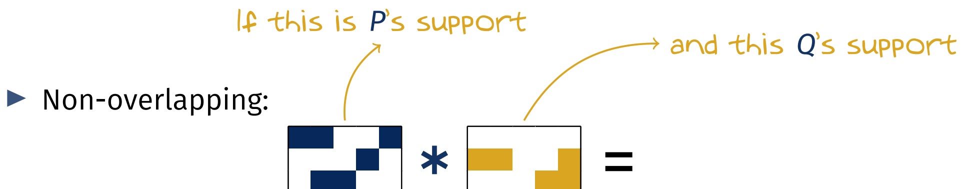
What we *build*: a separating conjunction

- Non-overlapping:



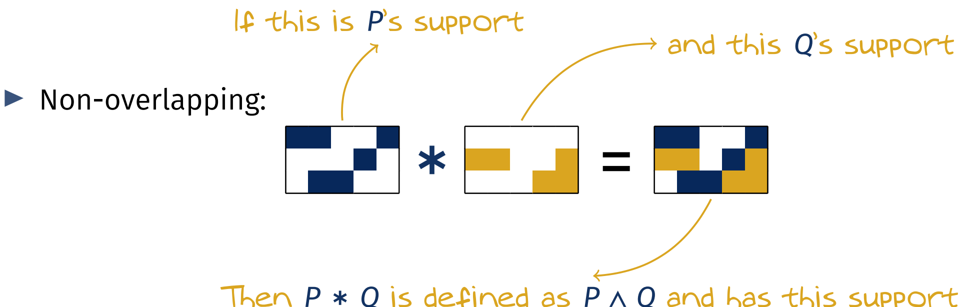
What we *build*: a separating conjunction

- Non-overlapping:



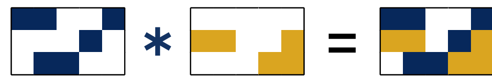
What we *build*: a separating conjunction

- Non-overlapping:

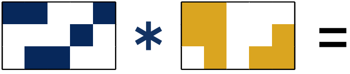


What we *build*: a separating conjunction

- Non-overlapping:

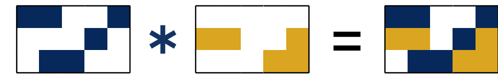


- Overlapping



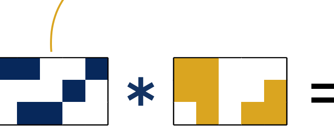
What we *build*: a separating conjunction

- Non-overlapping:



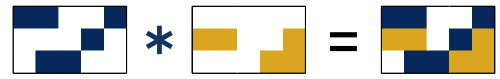
If this is P's support

- Overlapping

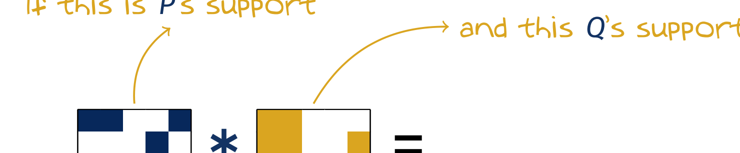


What we *build*: a separating conjunction

- Non-overlapping:

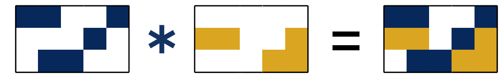


- Overlapping

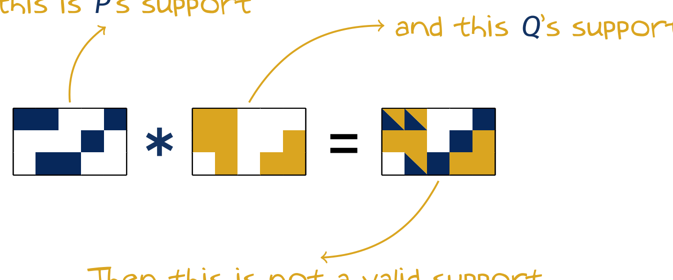


What we *build*: a separating conjunction

- Non-overlapping:



- Overlapping



What we *obtain*: the frame rule

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

What we *obtain*: the frame rule

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

If R is true and non-overlapping

What we *obtain*: the frame rule

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

If R is true and non-overlapping

Then R remains true and non-overlapping

Revisiting our footgun

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

$$\{\ \ell \mapsto 1\ \}\ \ \ell := 0\ \ \{\ \ell \mapsto 0\ \}$$

Revisiting our footgun

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$

$$\left\{ \begin{array}{c} \ell \mapsto 1 \\ * \end{array} \right\} \ell := 0 \left\{ \begin{array}{c} \ell \mapsto 0 \\ * \end{array} \right\}$$

Revisiting our footgun

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

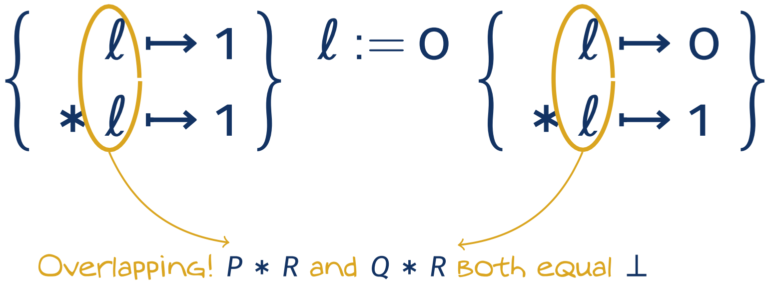
We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$
We pick: $R = \ell \mapsto 1$

$$\left\{\begin{array}{r} \ell \longmapsto 1 \\ * \, \ell \longmapsto 1 \end{array}\right\} \; \ell := 0 \; \left\{\begin{array}{r} \ell \longmapsto 0 \\ * \, \ell \longmapsto 1 \end{array}\right\}$$

Revisiting our footgun

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$
We pick: $R = \ell \mapsto 1$



Revisiting our footgun

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$
We pick: $R = \ell \mapsto 1$

$$\{\bot\}\ \ell := 0\ \{\bot\}$$

Revisiting our footgun

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

We have: $P = \ell \mapsto 1$ and $Q = \ell \mapsto 0$
We pick: $R = \ell \mapsto 1$

$$\{\bot\}\ \ell := 0\ \{\bot\}$$

🤖 Garbage in; garbage out

From Points-to to Ownership

$$\ell \mapsto v$$

Meaning:

- used to be "ℓ points to v"
- now is "I **own** ℓ and it points to v"

From Points-to to Ownership

$$\ell \mapsto v$$

Meaning:

- used to be "ℓ points to v"
- now is "I **own** ℓ and it points to v"

Ownership:

- is globally unique
- is transferrable
- allows destructive updates

From Points-to to Ownership

$$\ell \mapsto v$$

Meaning:

- used to be "ℓ points to v"
- now is "I **own** ℓ and it points to v"

Ownership:

Somewhat paradoxically, this allows **local** reasoning

- is globally unique
- is transferrable
- allows destructive updates

From Points-to to Ownership

$$\ell \mapsto v$$

Meaning:

- used to be "ℓ points to v"
- now is "I **own** ℓ and it points to v"

Ownership:

- is globally unique
- is transferrable
- allows destructive updates

All of this is implicitly enforced by the rules of the logic

Table of Contents

Motivation: Correct Concurrent Programs

Hoare Logic for Correct Imperative Programs

Separation Logic for Correct Concurrent Programs

Correct by Construction Concurrent Programs

Old School Verification: Write, Test, Fix loop

```
10 WRITE CODE
20 DO FORMALISATION
30 IF (CONTAINS BUG) THEN
40    GOTO 10
50 END IF
```

Correct by Construction: Specify, Implement Correctly, Keep

Sometimes known as goal-driven development

1. Write a specification
2. In a dialogue with the compiler interactively refine it
   * Each step produces part of the program
   * Some step introduce some further goals too
3. Keep refining until all goals are trivials

In This Talk: Idris 2

- Functional (lambdas, pure functions, inductive types)

```
swap : (a, b) -> (b, a)
swap = \ (x, y) => (y, x)
```

In This Talk: Idris 2

- Functional (lambdas, pure functions, inductive types)
- First class types (i.e. types are standard values)

```
FileLoc : Type
FileLoc = (String, Nat, Nat)
```

In This Talk: Idris 2

- Functional (lambdas, pure functions, inductive types)
- First class types (i.e. types are standard values)
- Resource-aware (separation of specification vs. runtime)

```
id : {0 a : Type} -> a -> a
id x = x
```

In This Talk: Idris 2

- Functional (lambdas, pure functions, inductive types)
- First class types (i.e. types are standard values)
- Resource-aware (separation of specification vs. runtime)

```
id : {0 a : Type} -> a -> a
id x = x
```

Quantity 0: erased during compilation

In This Talk: Idris 2

- Functional (lambdas, pure functions, inductive types)
- First class types (i.e. types are standard values)
- Resource-aware (separation of specification vs. runtime)
- Strict (with explicit Laziness annotations)

In This Talk: Idris 2

- Functional (lambdas, pure functions, inductive types)
- First class types (i.e. types are standard values)
- Resource-aware (separation of specification vs. runtime)
- Strict (with explicit Laziness annotations)
- Compiled to ChezScheme (great target for a functional language)

In This Talk: Idris 2

- Functional (lambdas, pure functions, inductive types)
- First class types (i.e. types are standard values)
- Resource-aware (separation of specification vs. runtime)
- Strict (with explicit Laziness annotations)
- Compiled to ChezScheme (great target for a functional language)
- Self-hosted (reasonably fast!)

In This Talk: Core Idea

Define a Domain Specific Language internalising Separation logic ideas

In This Talk: Core Idea

Define a Domain Specific Language internalising Separation logic ideas

- Linearity (ab)used to ensure global uniqueness

In This Talk: Core Idea

Define a Domain Specific Language internalising Separation logic ideas

- Linearity (ab)used to ensure global uniqueness
- Ownership proofs instead of raw pointers

In This Talk: Core Idea

Define a Domain Specific Language internalising Separation logic ideas

- Linearity (ab)used to ensure global uniqueness
- Ownership proofs instead of raw pointers
- Erasure to get rid of specification data (values showing up in Ps, Qs, Rs)

Ownership Type

$$region[start, end] \mapsto vs$$

```
data Owned :
   (region : Region) -> (start, end : Nat) ->
   (vs : List Bits8) -> Type where
```

Read

$$\left\{ \qquad \right\}$$

$$v = \texttt{getBits8}(idx);$$

$$\left\{ \qquad \right\}$$

Read

$$\left\{ \quad region[start, end] \mapsto vs \quad \right\}$$

$$v = \texttt{getBits8}(idx);$$

$$\left\{ \qquad \qquad \right\}$$

Read

$$\left\{ \begin{array}{rl} & region[start, end] \mapsto vs \\ * & 0 \leq idx < |vs| \end{array} \right\}$$

$$v = \texttt{getBits8}(idx);$$

$$\left\{ \quad \right\}$$

Read

$$\left\{ \begin{array}{ll} & region[start, end] \mapsto vs \\ * & 0 \leq idx < |vs| \end{array} \right\}$$

$$v = \texttt{getBits8}(idx);$$

$$\left\{ \begin{array}{ll} & region[start, end] \mapsto vs \\ * & v = vs[idx] \end{array} \right\}$$

Read

$$\left\{ \begin{array}{ll} & region[start, end] \mapsto vs \\ * & 0 \leq idx < |vs| \end{array} \right\}$$

$$v = \texttt{getBits8}(idx);$$

$$\left\{ \begin{array}{ll} & region[start, end] \mapsto vs \\ * & v = vs[idx] \end{array} \right\}$$

```
getBits8 :
  LinearIO io =>
  {start, end : Nat} ->
  (1 _ : Owned region start end vs) ->
  (idx : Nat) -> (0 _ : InBounds idx vs) ->
  L1 io (WithVal (Owned region start end vs)
                 (Singleton (index idx vs)))
```

Write

$$\Big\{ \qquad \Big\}$$

$\texttt{setBits8}(idx, val);$

$$\Big\{ \qquad \Big\}$$

Write

$$\left\{ \; region[start, end] \mapsto vs \; \right\}$$

```
setBits8(idx, val);
```

$$\left\{ \quad \right\}$$

Write

$$\left\{ \begin{array}{rl} & region[start, end] \mapsto vs \\ * & 0 \leq idx < |vs| \end{array} \right\}$$

$$\texttt{setBits8}(idx, val);$$

$$\left\{ \qquad \right\}$$

Write

$$\left\{ \begin{array}{rl} & region[start, end] \mapsto vs \\ * & 0 \leq idx < |vs| \end{array} \right\}$$

$$\texttt{setBits8}(idx, val);$$

$$\{ \quad region[start, end] \mapsto vs[idx := val] \quad \}$$

Write

$$\left\{ \begin{array}{rl} & region[start, end] \mapsto vs \\ * & 0 \leq idx < |vs| \end{array} \right\}$$

$$\texttt{setBits8}(idx, val);$$

$$\left\{ \quad region[start, end] \mapsto vs[idx := val] \quad \right\}$$

```
setBits8 :
  LinearIO io =>
  {start : Nat} ->
  (1 _ : Owned region start end vs) ->
  (idx : Nat) -> (0 _ : InBounds idx vs) ->
  (val : Bits8) ->
  L1 io (Owned region start end (replaceAt idx val vs))
```

Split

$$\Big\{ \qquad \Big\}$$

`splitAt`(m);

$$\Big\{ \qquad \Big\}$$

Split

$$\left\{ \quad region[start, end] \mapsto vs +\!\!+\, ws \quad \right\}$$

$\texttt{splitAt}(m);$

$$\left\{ \qquad \qquad \right\}$$

Split

$$\left\{ \begin{array}{ll} & region[start, end] \mapsto vs +\!\!+\, ws \\ * & |vs| = m \end{array} \right\}$$

$$\texttt{splitAt}(m);$$

$$\left\{ \qquad \right\}$$

Split

$$\left\{ \begin{array}{ll} & region[start, end] \mapsto vs +\!\!+\, ws \\ * & |vs| = m \end{array} \right\}$$

$$\texttt{splitAt}(m);$$

$$\left\{ \begin{array}{ll} & region[start, start + m] \mapsto vs \\ * & region[start + m, end] \mapsto ws \end{array} \right\}$$

Split

$$\left\{ \begin{array}{ll} & region[start, end] \mapsto vs \mathbin{+\!\!+} ws \\ * & |vs| = m \end{array} \right\}$$

$$\texttt{splitAt}(m);$$

$$\left\{ \begin{array}{ll} & region[start, start + m] \mapsto vs \\ * & region[start + m, end] \mapsto ws \end{array} \right\}$$

```
splitAt :
  {0 vs, ws : List Bits8} ->
  {m : Nat} -> (0 _ : HasLength m vs) ->
  Owned region start end (vs ++ ws) -@
  LPair (Owned region start (start + m) vs)
        (Owned region (start + m) end ws)
```

Combine

$$\left\{ \vphantom{\begin{matrix}a\\b\end{matrix}} \right.\qquad\qquad\qquad\qquad\left. \vphantom{\begin{matrix}a\\b\end{matrix}} \right\}$$

`combine();`

$$\left\{ \vphantom{a} \right.\qquad\qquad\qquad\qquad\left. \vphantom{a} \right\}$$

Combine

$$\left\{ \begin{array}{rl} & region[start, middle] \mapsto vs \\ * & region[middle, end] \mapsto ws \end{array} \right\}$$

```
combine();
```

$$\left\{ \quad \right\}$$

Combine

$$\left\{ \begin{array}{rl} & region[start, middle] \mapsto vs \\ * & region[middle, end] \mapsto ws \end{array} \right\}$$

```
combine();
```

$$\left\{ \quad region[start, end] \mapsto vs \mathbin{+\!\!+} ws \quad \right\}$$

Combine

$$\left\{ \begin{array}{ll} & \mathit{region}[\mathit{start}, \mathit{middle}] \mapsto \mathit{vs} \\ * & \mathit{region}[\mathit{middle}, \mathit{end}] \mapsto \mathit{ws} \end{array} \right\}$$

$$\texttt{combine}();$$

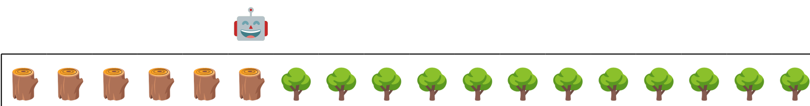
$$\left\{ \mathit{region}[\mathit{start}, \mathit{end}] \mapsto \mathit{vs} \mathbin{+\!\!+} \mathit{ws} \right\}$$

```
(++) :
  Owned region start middle vs -@
  Owned region middle end ws -@
  Owned region start end (vs ++ ws)
```

Map Type

```
Map : (Type -> Type) -> Type
Map io =
  forall region. {start, end : Nat} ->
  {0 trees : List Bits8} ->
  (saw : Bits8 -> Bits8) ->
  (1 _ : Owned region start end           trees) ->
  L1 io (Owned region start end (map saw trees))
```

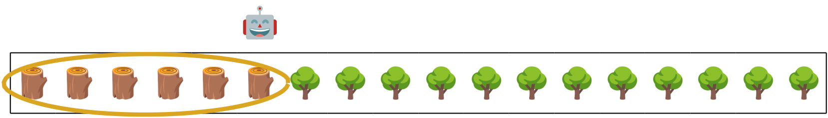
Sequential Map - Loop Type



Sequential Map - Loop Type

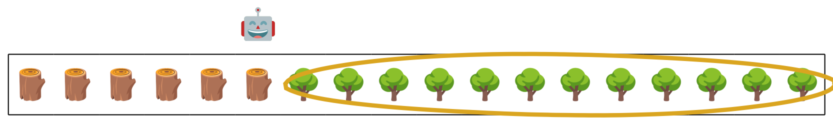
```
(1 _ : Owned region start end ((map saw treesL) <>> treesR)) ->
L1 io (Owned region start end (map saw (treesL <>> treesR)))
```

Sequential Map - Loop Type



```
(1 _ : Owned region start end ((map saw treesL) <>> treesR)) ->
L1 io (Owned region start end (map saw (treesL <>> treesR)))
```

Sequential Map - Loop Type



```
(1 _ : Owned region start end ((map saw treesL) <>> treesR)) ->
L1 io (Owned region start end (map saw (treesL <>> treesR)))
```

Parallel Map

Parallel Map

```
halve :
  {start, end : Nat} ->
  (1 _ : Owned region start end trees) ->
  Res Nat (\ m =>
  LPair (Owned region start         (start + m)  (take m trees))
        (Owned region (start + m)  end          (drop m trees)))
```

Parallel Map

```
halve :
  {start, end : Nat} ->
  (1 _ : Owned region start end trees) ->
  Res Nat (\ m =>
  LPair (Owned region start           (start + m)  (take m trees))
        (Owned region (start + m)  end             (drop m trees)))
```

```
par1 : L1 IO a -@ L1 IO b -@ L1 IO (LPair a b)
```

Parallel Map

```
halve :
  {start, end : Nat} ->
  (1 _ : Owned region start end trees) ->
  Res Nat (\ m =>
  LPair (Owned region start          (start + m)  (take m trees))
        (Owned region (start + m) end             (drop m trees)))
```

```
par1 : L1 IO a -@ L1 IO b -@ L1 IO (LPair a b)
```

```
parMapRec : Map IO -> Map IO
parMapRec subMap saw buf
   = do let (m # lbuf # rbuf) = halve buf
        (lbuf # rbuf) <- par1 (subMap saw lbuf) (subMap saw rbuf)
        let 1 buf = lbuf ++ rbuf
        pure1 (reindex (mapTakeDrop saw m trees) buf)
```

Parallel Reduce

Apply the same principles to get a parallel reduce
Relying on monoid laws to prove correctness

What's next?

Separation logic has a lot more to offer!

- Partial ownership (shared reads, owned writes)
- Locks (non-deterministic access to shared resources)
- Ghost states (stateful specification data)

Use these building blocks!

- Richly typed parallel skeletons
- Reintroduce layers of abstractions (e.g. inductive types)
- Seamless programming over serialised data
- Concurrent programs

Happy to Chat! See You in Glasgow?

🌐 https://gallais.github.io

https://mamot.fr/@gallais

TYPES 2025 — 9–13 June
Glasgow, Scotland
https://msp.cis.strath.ac.uk/types2025/