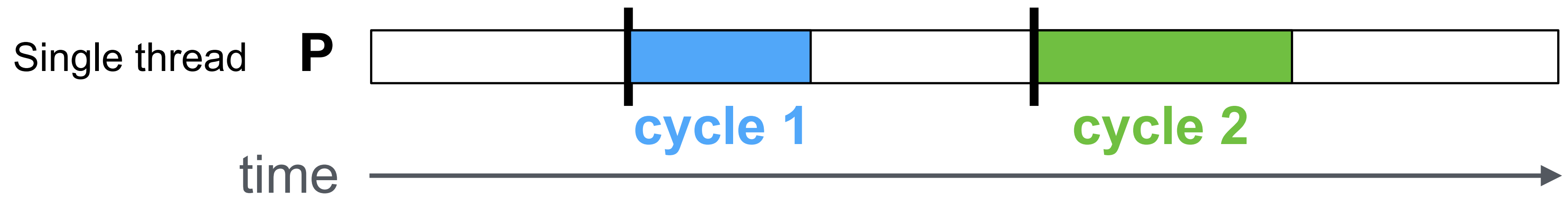
Verified Compilation of Linearisable Data Structures

Yannick Zakowski

David Cachera Delphine Demange David Pichardie

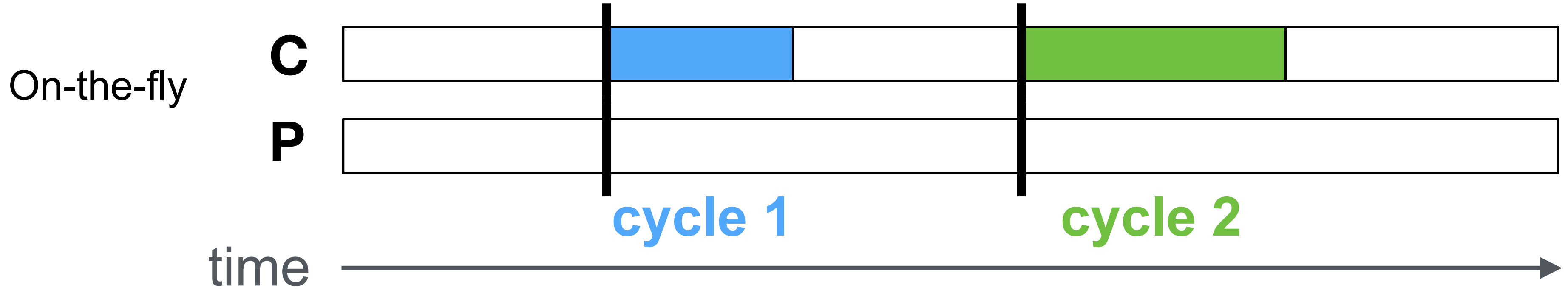
Introduction: a motivating example

Verifying an on-the-fly garbage collector



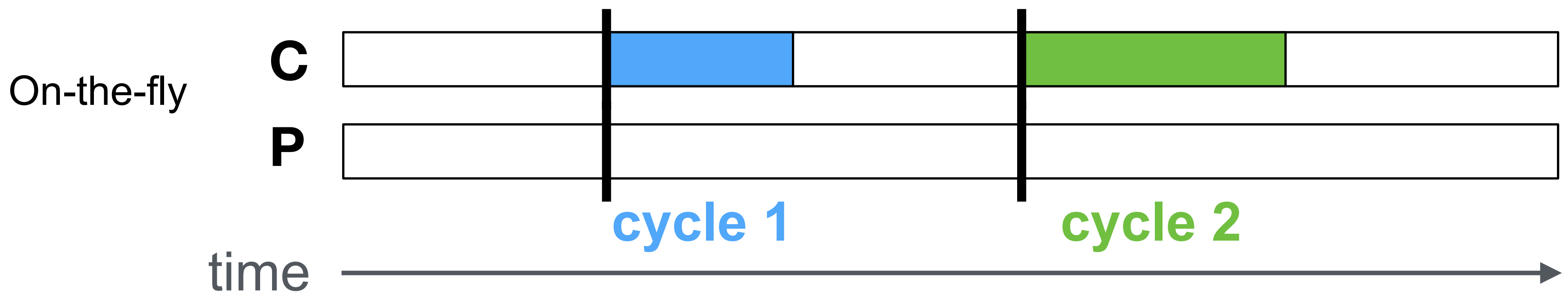
With a *sequential* GC, the main program pauses during collection

Verifying an on-the-fly garbage collector



An on-the-fly GC is hosted in a different thread, and collects the memory without ever pausing the main program

Verifying an on-the-fly garbage collector

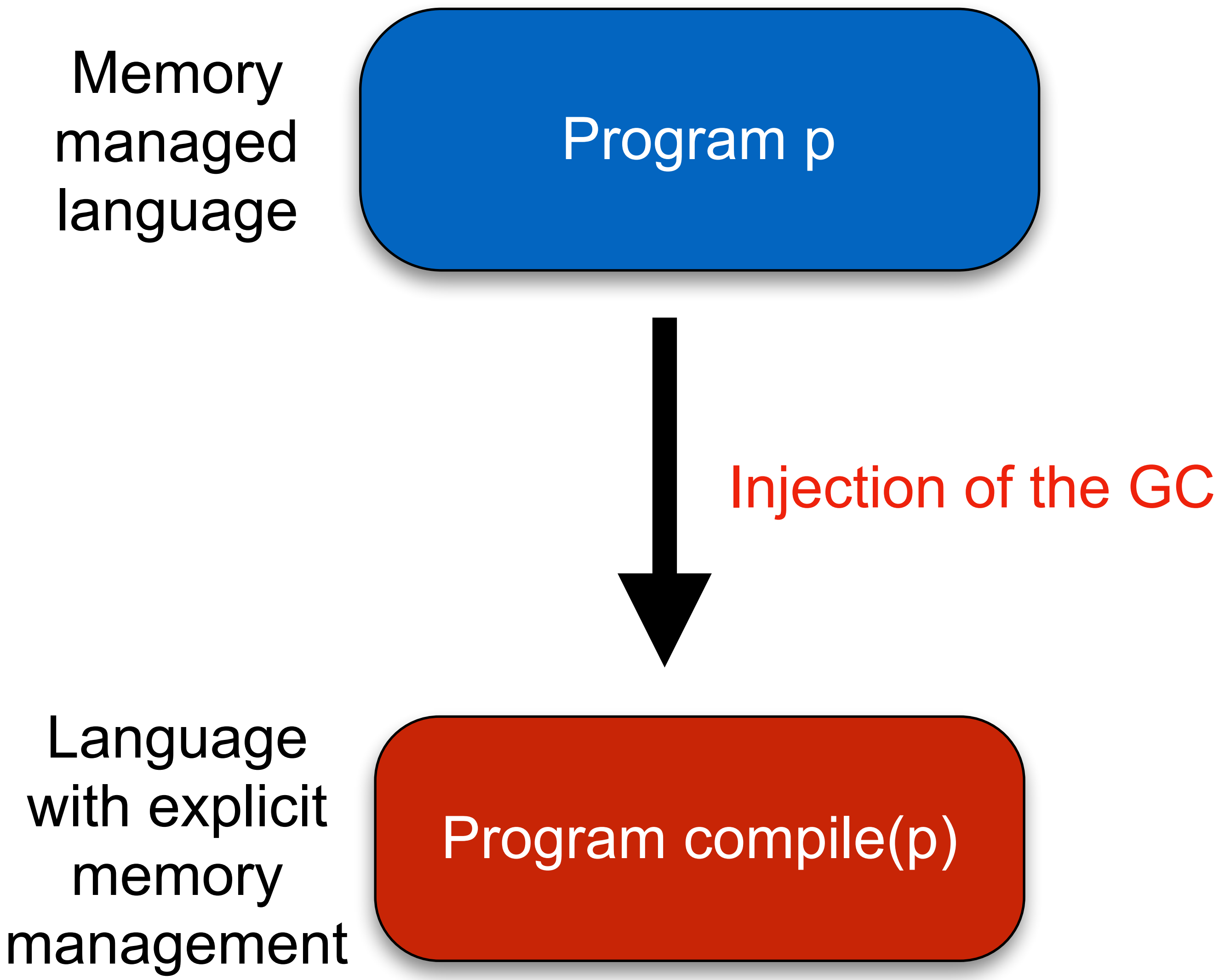


***An** on-the-fly GC is hosted in a different thread,*
and collects the memory without ever pausing the main program

Theorem (informal)

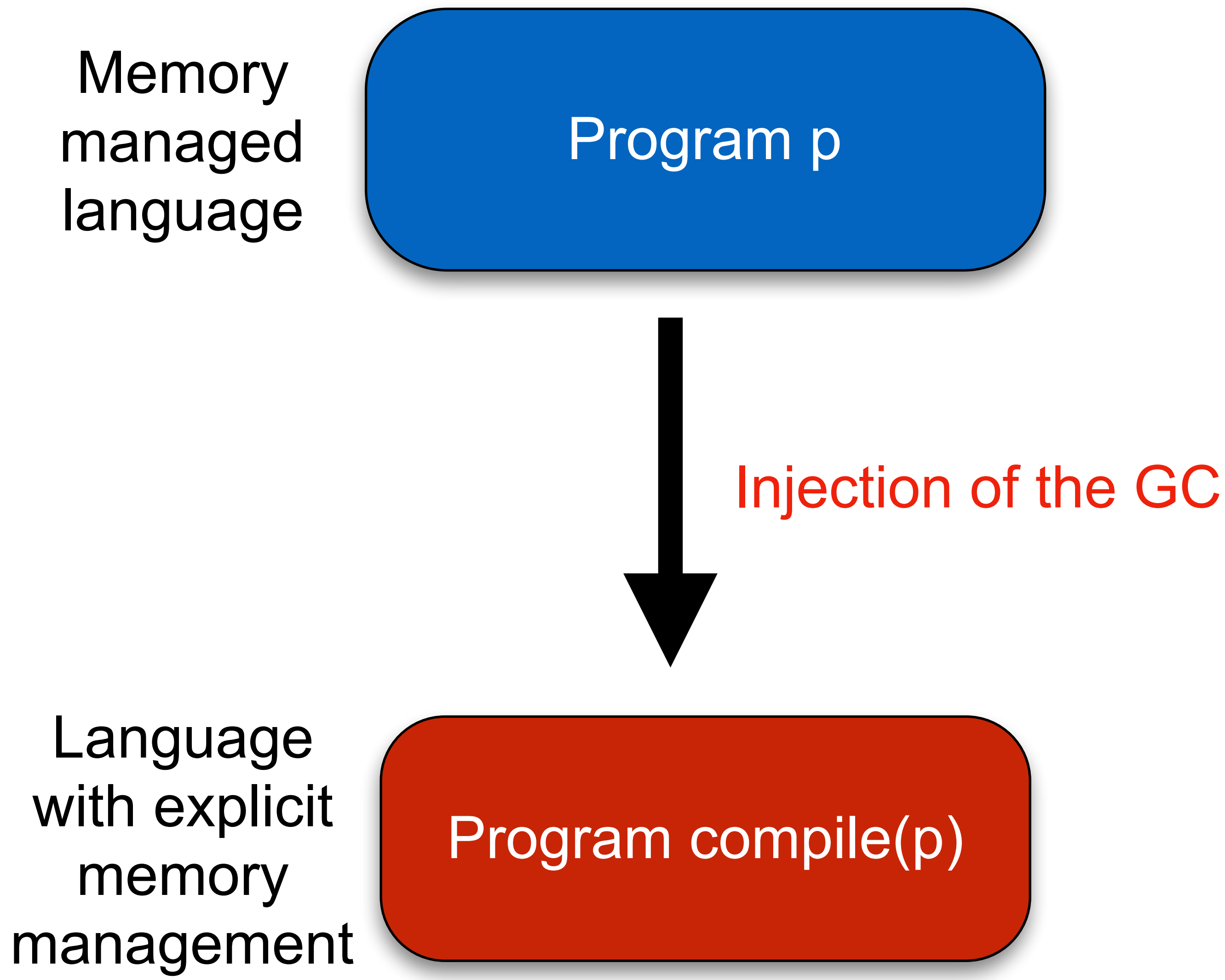
The collector never reclaims a part of the memory that can still be accessed by the program

Verifying an on-the-fly garbage collector
in the context of verified compilation



Verifying an on-the-fly garbage collector

in the context of verified compilation



Observational refinement

```
∀ P P′ obs,
        compiler P = OK P′ ∧
        low_exec P′ obs  ⇒
        high_exec P obs
```

A verified on-the-fly garbage collector

```
Scan:
 repeat
   no_gray = true;
   foreach x ∈ OBJECTS
     if x.color == GRAY
       no_gray = false;
       foreach f ∈ fields(x) do
          MarkGray(x.f);
       x.color = BLACK
 until no_gray
Sweep:
 foreach x ∈ OBJECTS
   if x.color == WHITE
   then FREE(x)
Clear:
 foreach x ∈ OBJECTS
   x.color = WHITE
```

```
if x.color = WHITE then
   push(buffer[m], x)
```

A verified on-the-fly garbage collector

```
Scan:
 repeat
   no_gray = true;
   foreach x ∈ OBJECTS
     if x.color == GRAY
       no_gray = false;
       foreach f ∈ fields(x) do
         MarkGray(x.f);
       x.color = BLACK
 until no_gray
Sweep:
 foreach x ∈ OBJECTS
   if x.color == WHITE
   then FREE(x)
Clear:
 foreach x ∈ OBJECTS
   x.color = WHITE
```

```
if x.color = WHITE then
  push(buffer[m], x)
```

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
m.next_write = nw
```

A verified on-the-fly garbage collector ?

?

```
Scan:
 repeat
   no_gray = true;
   foreach x ∊ OBJECTS
     if x.color == GRAY
       no_gray = false;
       foreach f ∊ fields(x) do
         MarkGray(x.f);
       x.color = BLACK
 until no_gray
Sweep:
 foreach x ∊ OBJECTS
   if x.color == WHITE
   then FREE(x)
Clear:
 foreach x ∊ OBJECTS
   x.color = WHITE
```

```
if x.color = WHITE then
  push(buffer[m], x)
```

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
m.next_write = nw
```

1. Linearisability

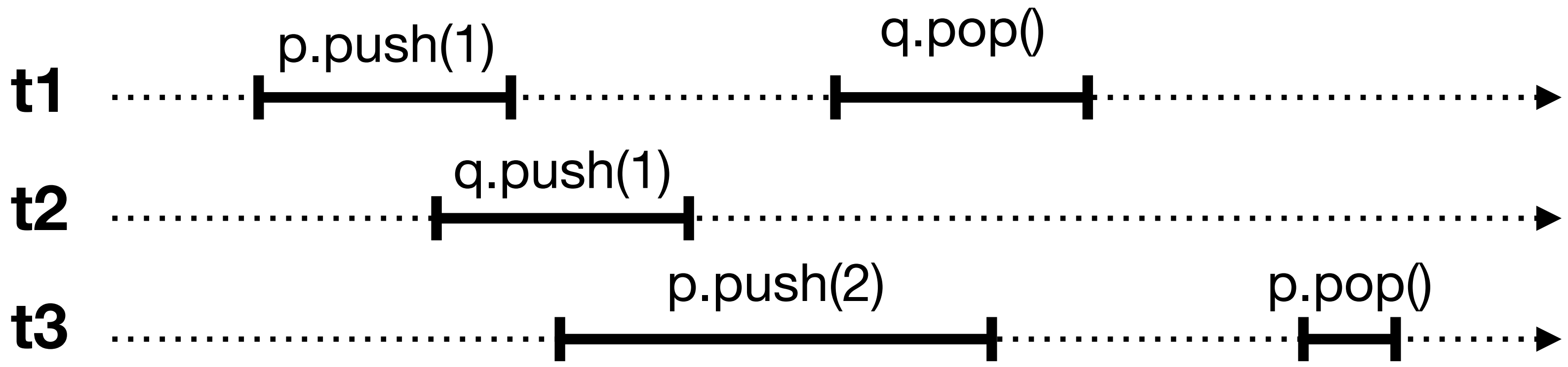
2. Using our theorem: proving linearisability through Rely-Guarantee

3. Under the hood: systematic derivation of a simulation

Linearisability

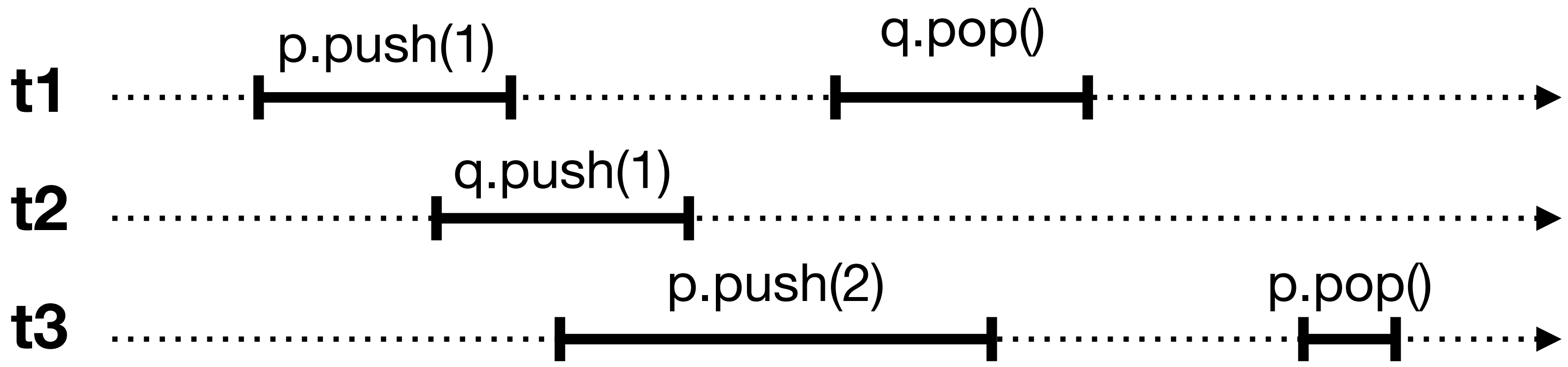
Linearisability
[Herlihy and Wing 90]

A notion of coherence for concurrent data structures



Linearisability
[Herlihy and Wing 90]

A notion of coherence for concurrent data structures

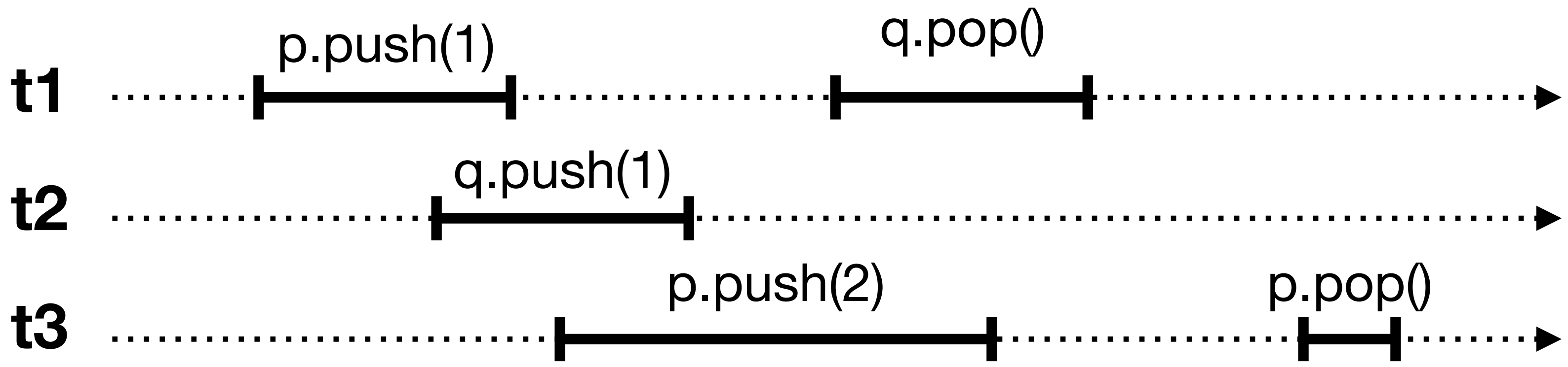


Principle 1.
Any method should appear to happen in a one-at-a-time order

Linearisability

[Herlihy and Wing 90]

A notion of coherence for concurrent data structures



Principle 1.
Any method should appear to happen in a one-at-a-time order

Principle 2. (Linearisability)
Any method should appear to take effect instantaneously
at some moment between its call and return

Linearisability

Original formal definition

- Expressed in terms of traces of events (histories)
- For all possible history, there exists an “equivalent” well-behaved history
- Great, but does not fit our story

Two main caveats

- The property is not explicitly usable for verified compilation purpose

  ⟶ Change definition!
- Histories are global objects, difficult to reason about

  ⟶ Derive it from RG proof obligations!

Linearisability as an observational refinement

We see refinement as a compilation pass

- Source language:
  - abstract data structure
  - atomic operations over it
- Target language:
  only concrete operations
- Compilation pass:
  provides a concrete implementation

Linearisability as an observational refinement

We see refinement as a compilation pass

- Source language:
  - abstract data structure
  - atomic operations over it
- Target language:
  only concrete operations
- Compilation pass:
  provides a concrete implementation

```
if x.color = WHITE then
   push(buffer[m], x)
```

↓

```
if x.color = WHITE then
   nw = m.next_write
   nr = m.next_read
   d = m.data
   d[nw] = x
   nw = (nw+1) mod SIZE
   assume (nr == nw)
   m.next_write = nw
```

Linearisability as an observational refinement

We see refinement as a compilation pass

- Source language:
  - abstract data structure
  - atomic operations over it
- Target language:
  only concrete operations
- Compilation pass:
  provides a concrete implementation

$$\mathrm{Obs}(\mathcal{T}(\mathrm{p})) \subseteq Obs(p)$$

```
if x.color = WHITE then
   push(buffer[m], x)
```

↓

```
if x.color = WHITE then
   nw = m.next_write
   nr = m.next_read
   d = m.data
   d[nw] = x
   nw = (nw+1) mod SIZE
   assume (nr == nw)
   m.next_write = nw
```

Using our result: proving linearisability via Rely-Guarantee

Rely Guarantee reasoning

[Jones81]

$$R, G, I \vdash \{P\}\ c\ \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness
Invariant

Annotations

Rely Guarantee reasoning

[Jones81]

$$R, G, I \ \vdash \{P\} \ c \ \{Q\}$$

Environment
R: Rely
G: Guarantee

Global Correctness
Invariant

Annotations

Rely Guarantee reasoning
[Jones81]

$$R, G, I \vdash \{P\}\ c\ \{Q\}$$

Environment
R: Rely
G: Guarantee

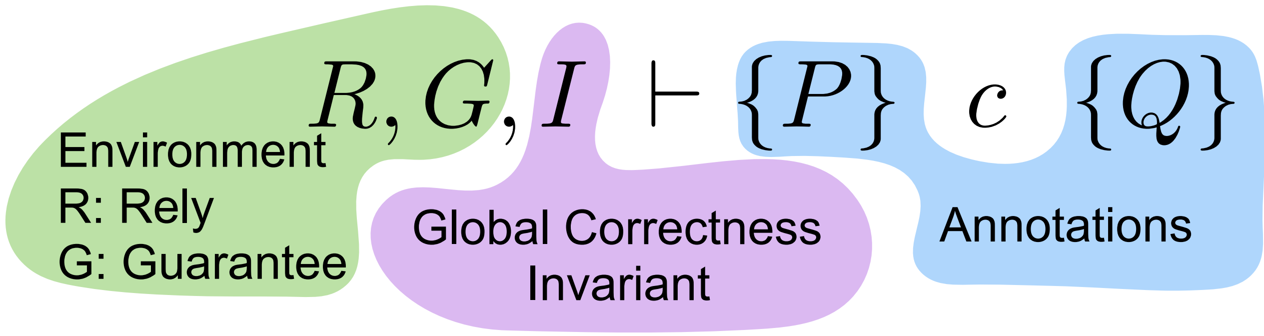
Global Correctness Invariant

Annotations

Rely Guarantee reasoning

[Jones81]

$$R, G, I \vdash \{P\}\ c\ \{Q\}$$



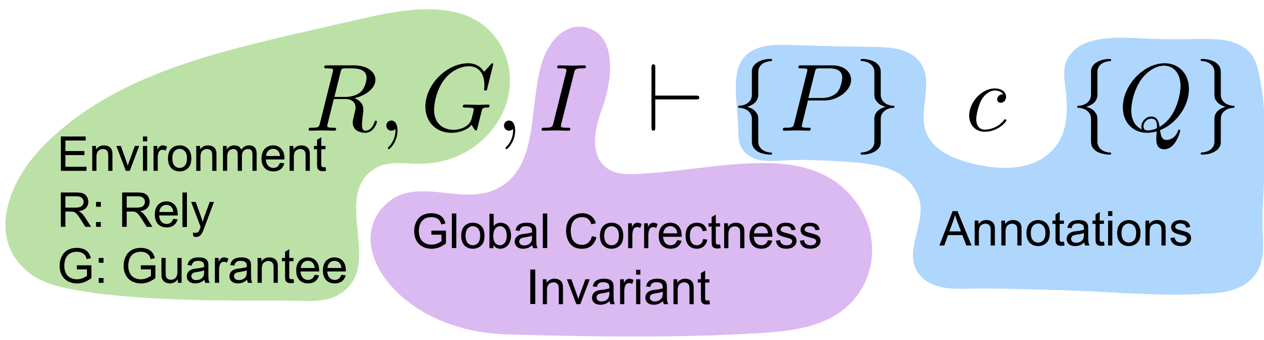
R : Rely, approximates the effect of the environment

G : Guarantee, approximates the effect of the thread

Rely Guarantee reasoning

[Jones81]

$$R, G, I \vdash \{P\}\ c\ \{Q\}$$



R : Rely, approximates the effect of the environment

G : Guarantee, approximates the effect of the thread

A thread is proved against a contract.
The notion of interference is checked against this contract.

Reasoning about linearisation using Rely-Guarantee

Introduction of an intermediate, *instrumented*, language.

- Explicit annotation of *linearisation points*
- Hybrid states, both concrete and abstract
- Linearisation points trigger the abstract semantics

Reasoning about linearisation using Rely-Guarantee

Introduction of an intermediate, *instrumented*, language.

- Explicit annotation of *linearisation points*
- Hybrid states, both concrete and abstract
- Linearisation points trigger the abstract semantics

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>
```

Reasoning about linearisation using Rely-Guarantee

Introduction of an intermediate, *instrumented*, language.

- Explicit annotation of *linearisation points*
- Hybrid states, both concrete and abstract
- Linearisation points trigger the abstract semantics

local map	ρ_1
shared heap	σ_1
abstract data-structure	p_1

```
→ nw = m.next_write
  nr = m.next_read
  d = m.data
  d[nw] = x
  nw = (nw+1) mod SIZE
  assume (nr == nw)
  <m.next_write = nw; LIN>
```

Reasoning about linearisation using Rely-Guarantee

Introduction of an intermediate, *instrumented*, language.

- Explicit annotation of *linearisation points*
- Hybrid states, both concrete and abstract
- Linearisation points trigger the abstract semantics

local map	ρ_1	ρ_2	ρ_3	ρ_4	ρ_4	ρ_5	ρ_5
shared heap	σ_1	σ_1	σ_1	σ_1	σ_2	σ_2	σ_2
abstract data-structure	p_1	p_1	p_1	p_1	p_1	p_1	p_1

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>
```

Reasoning about linearisation using Rely-Guarantee

Introduction of an intermediate, *instrumented*, language.

- Explicit annotation of *linearisation points*
- Hybrid states, both concrete and abstract
- Linearisation points trigger the abstract semantics

local map	ρ_1	ρ_2	ρ_3	ρ_4	ρ_4	ρ_5	ρ_5	ρ_5
shared heap	σ_1	σ_1	σ_1	σ_1	σ_2	σ_2	σ_2	σ_3
abstract data-structure	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_2

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>
```

→

Reasoning about linearisation using Rely-Guarantee

Introduction of an intermediate, *instrumented*, language.

- Explicit annotation of *linearisation points*
- Hybrid states, both concrete and abstract
- Linearisation points trigger the abstract semantics

local map	ρ_1	ρ_2	ρ_3	ρ_4	ρ_4	ρ_5	ρ_5	ρ_5
shared heap	σ_1	σ_1	σ_1	σ_1	σ_2	σ_2	σ_2	σ_3
abstract data-structure	p_1	p_1	p_1	p_1	p_1	p_1	p_1	p_2
linearisation state	B	B	B	B	B	B	B	$A(v) \rightarrow$

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>
```

Proving linearisability: the perspective of a user

Proving linearisability: the perspective of a user

Abstract data structure

Buf := Empty | Cons x b

b.Push(x) = Cons x b

Proving linearisability: the perspective of a user

Abstract data structure

Buf := Empty | Cons x b

b.Push(x) = Cons x b

Concrete implementation of methods

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>
```

Proving linearisability: the perspective of a user

Abstract data structure

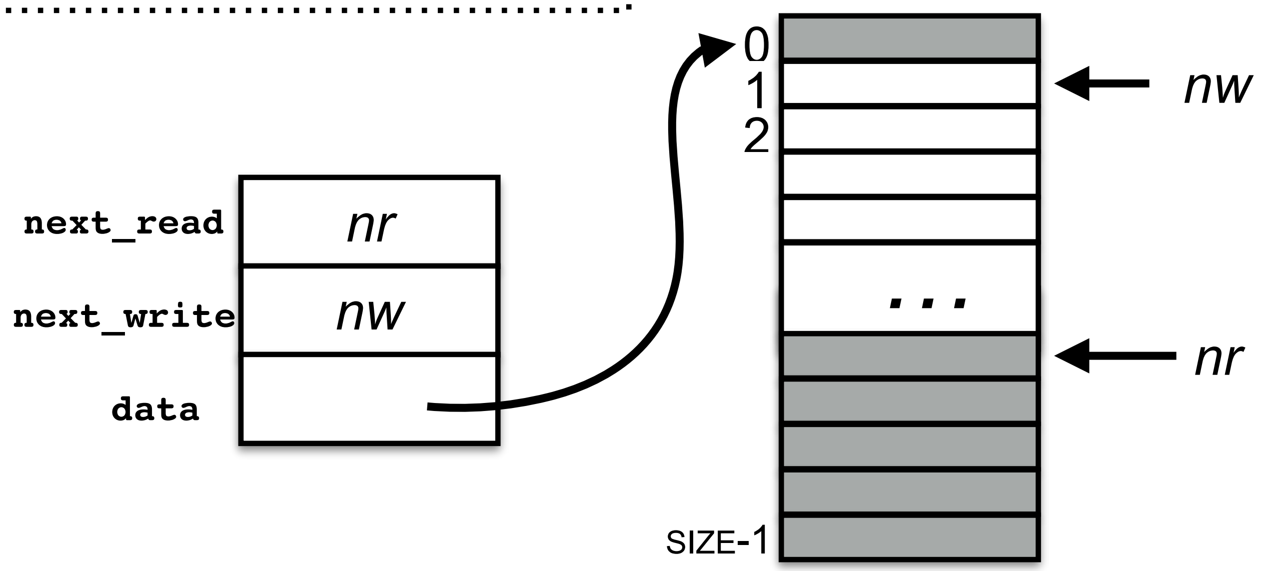
Buf := Empty | Cons x b

b.Push(x) = Cons x b

Concrete implementation of methods

nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>

Coherence invariant I_c



Proving linearisability: the perspective of a user

Abstract data structure

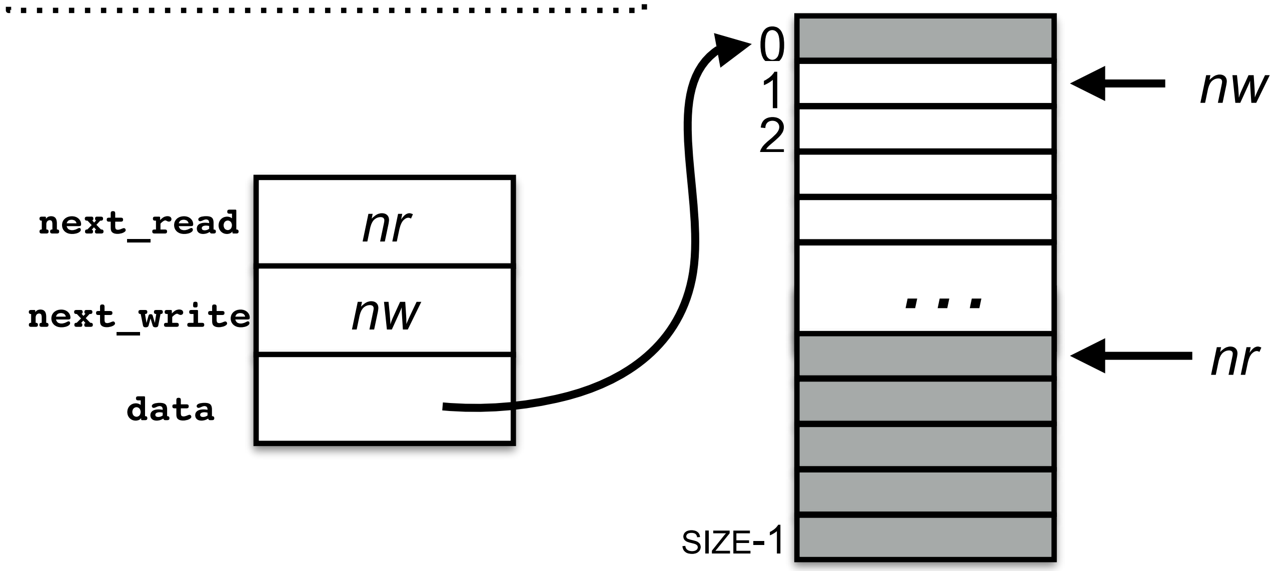
Buf := Empty | Cons x b b.Push(x) = Cons x b

Concrete implementation of methods

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>
```

Coherence invariant I_c

Relies and guarantees $R_m\ G_m$



Proving linearisability: the perspective of a user

Abstract data structure

Concrete implementation of methods

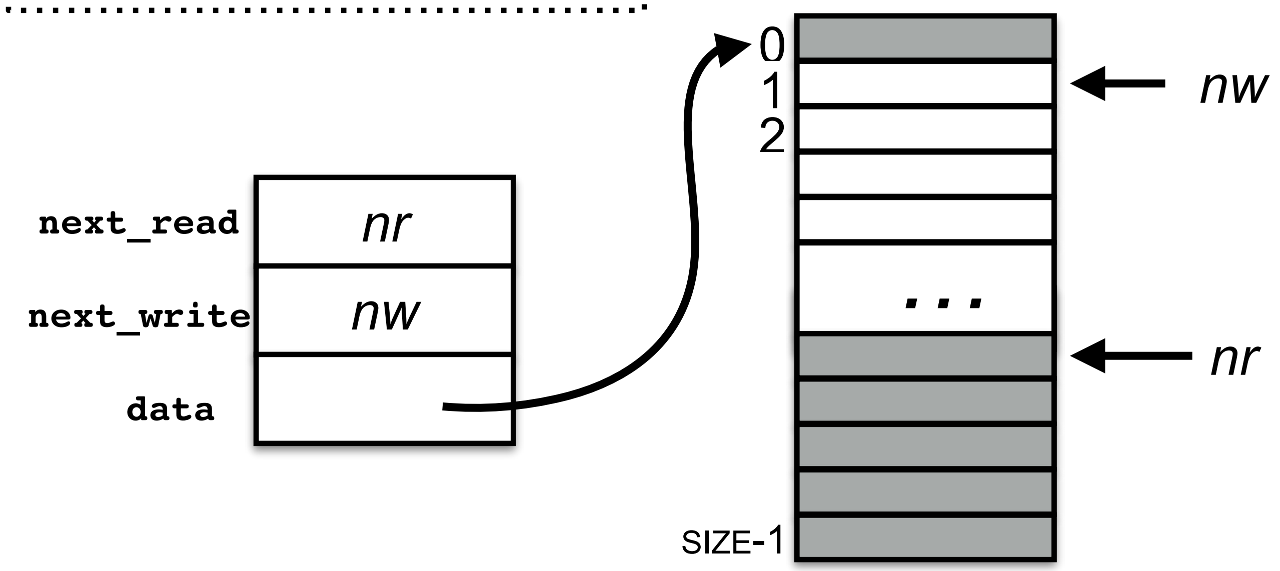
Coherence invariant I_c

Relies and guarantees R_m G_m

Buf := Empty | Cons x b

b.Push(x) = Cons x b

```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
<m.next_write = nw; LIN>
```



Proving linearisability: the perspective of a user

Abstract data structure

Concrete implementation of methods

Coherence invariant I_c

Relies and guarantees R_m G_m

Proving linearisability: the perspective of a user

Abstract data structure

Concrete implementation of methods

Coherence invariant I_c

Relies and guarantees $R_m \ G_m$

RG method specification

$$R_{push}, G_{push}, I_c \vdash$$
$$\{\texttt{ln} = B\}$$
$$p.push(v)$$
$$\{\texttt{ln} = A(v_1) \wedge \texttt{ret} = v_1\}$$

Proving linearisability: the perspective of a user

Abstract data structure

Concrete implementation of methods

Coherence invariant I_c

Relies and guarantees R_m G_m

RG method specification

Stability obligations

$$R_{push}, G_{push}, I_c \vdash$$
$$\{\texttt{ln} = B\}$$
$$p.push(v)$$
$$\{\texttt{ln} = A(v_1) \wedge \texttt{ret} = v_1\}$$

I_c stable under R_{push}

Proving linearisability: the perspective of a user

Abstract data structure

Concrete implementation of methods

Coherence invariant I_c

Relies and guarantees R_m G_m

RG method specification

Stability obligations

RG consistency

$$R_{push}, G_{push}, I_c \vdash$$
$$\{\texttt{ln} = B\}$$
$$p.push(v)$$
$$\{\texttt{ln} = A(v_1) \wedge \texttt{ret} = v_1\}$$

$$I_c \text{ stable under } R_{push}$$

$$G_{push} \subseteq R_{pop}$$
$$G_{pop} \subseteq R_{push}$$

Proving linearisability: the perspective of a user

Abstract data structure

Concrete implementation of methods

Coherence invariant I_c

Relies and guarantees R_m G_m

Reasoning locally exclusively on methods

↓ **Automatically obtain**

RG method specification

Stability obligations

RG consistency

Observational refinement of the compilation pass implementing the methods *for any client*

Refining linearisable data-structures

```
Scan:
  repeat
    no_gray = true;
    foreach x ∈ OBJECTS
      if x.color == GRAY
        no_gray = false;
        foreach f ∈ fields(x) do
          MarkGray(x.f);
        x.color = BLACK
  until no_gray
Sweep:
  foreach x ∈ OBJECTS
    if x.color == WHITE
    then FREE(x)
Clear:
  foreach x ∈ OBJECTS
    x.color = WHITE
```

```
if x.color = WHITE then
  push(buffer[m], x)
```

Refining linearisable data-structures

```
Scan:
 repeat
   no_gray = true;
   foreach x ∈ OBJECTS
     if x.color == GRAY
       no_gray = false;
       foreach f ∈ fields(x) do
         MarkGray(x.f);
       x.color = BLACK
 until no_gray
Sweep:
 foreach x ∈ OBJECTS
   if x.color == WHITE
   then FREE(x)
Clear:
 foreach x ∈ OBJECTS
   x.color = WHITE
```

```
if x.color = WHITE then
   push(buffer[m], x)
```

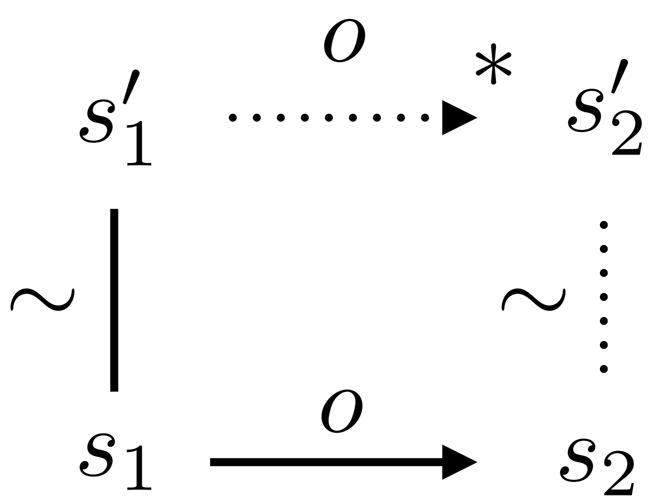
```
nw = m.next_write
nr = m.next_read
d = m.data
d[nw] = x
nw = (nw+1) mod SIZE
assume (nr == nw)
m.next_write = nw
```

A quick peak under the hood

Backward simulations

Inductive step used to prove observational refinement

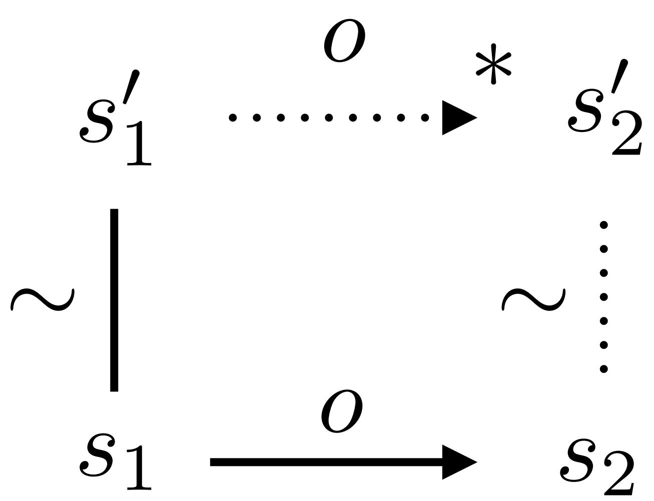
$\sim$ Relation between states of the source and target language

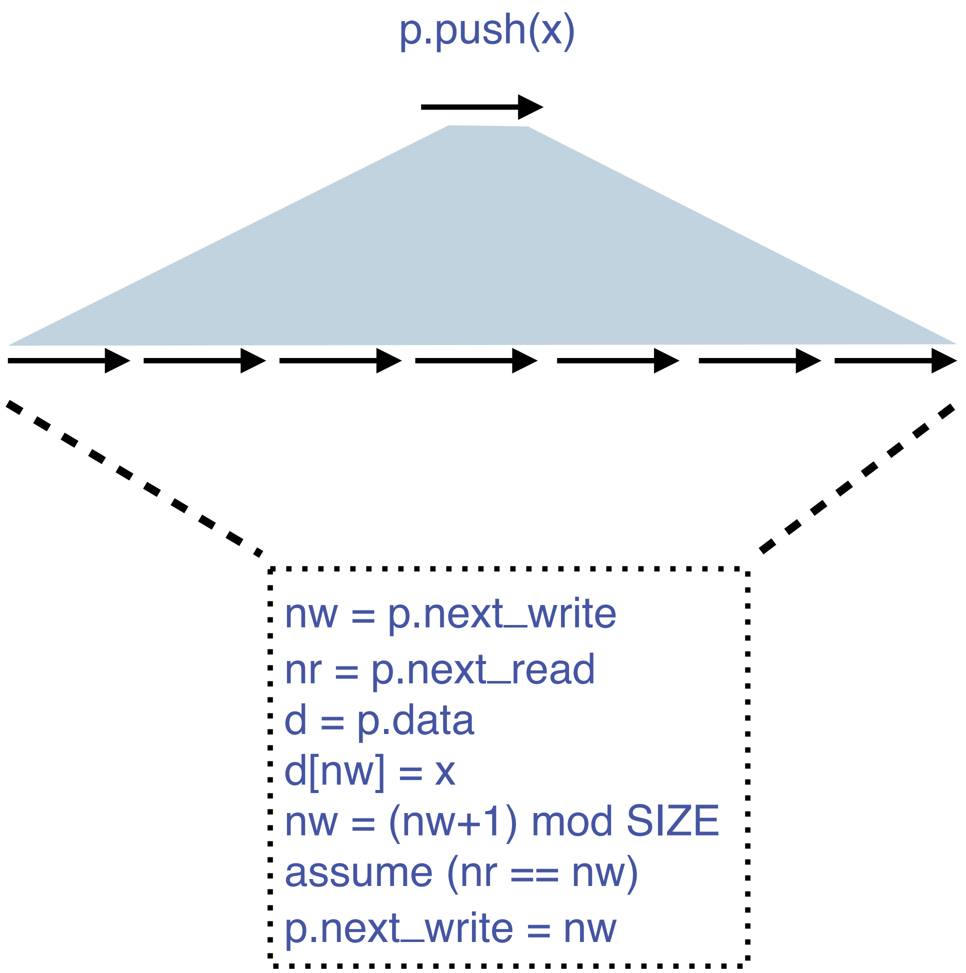


Backward simulations

Inductive step used to prove observational refinement

$\sim$ Relation between states of the source and target language



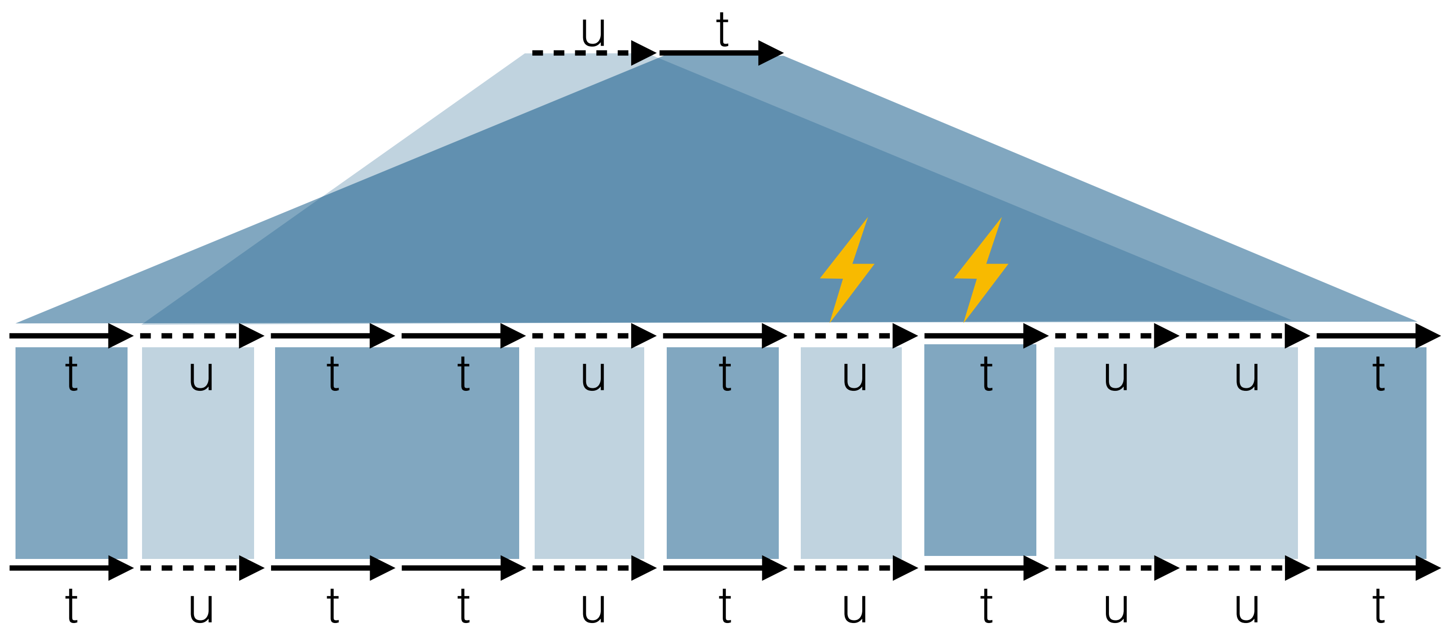


Two simulations composed

The compilation pass is split in two phases

- Implementation of the data structure
- Cleaning of the instrumentation

We therefore build two simulations, and compose them



Structure of the proof: an intuition

Design and prove a rich invariant at the instrumented level

Objective: carry enough information to leverage the RG specification

- Maintains the coherence invariant
- Builds partial executions of encountered methods

Prove thread local simulations

- For each thread, build a simulation parameterised by its rely
- Use the partial execution of methods to invoke the RG specification when needed

Combine the simulations using the stability assumptions

Conclusion

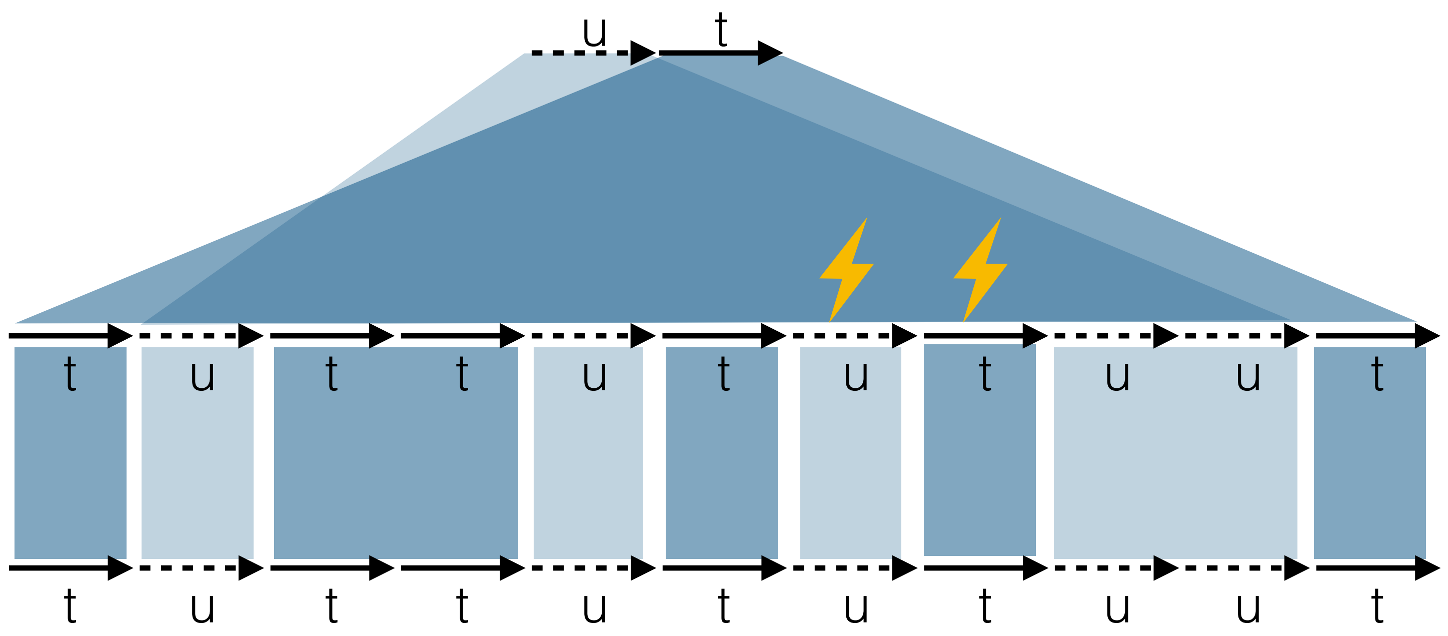
- Linearisability expressed in term of observational refinement
- A *local*, sufficient condition expressed in terms of Rely-Guarantee
- A *generic* meta-theorem: can be instantiated with any data structure

  (provided you manage to discharge the proof obligations
- Provide strong semantic foundations:

  - all theorem expressed wrt an operational semantics

  - everything formalised in Coq
- Instantiated on a realistic example used in another project
- ~13.5 kloc

Thank you



Appendix

Linearisability: limits of our result

Future-dependent linearisation points

- Example: pair snapshot
- Linearisation is confirmed at a later point of execution
- Need: Maintain two speculative simulations in parallel

Helping-based linearisation

- Example: HSY elimination-based stack
- Linearisation of thread A is performed by a step from thread B
- Need: Global view of the situation of each thread inside their method

Separation logic

- **Rely-Guarantee: reasoning about races**
- **Separation logic: proving concisely the absence of races**

Assertions describe more precisely the memory.
They can be interpreted as ownership of ressources.

$$[\![r \mapsto v]\!] = \{h \mid h(r) = v \wedge dom(h) = \{r\}\}$$

Achieves great modularity through the *frame rule*

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

Several works combine RG and SL: RGSep, SAGL, Iris, …