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.
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

Program compile(p)

Injection of the GC

Program p

Memory managed language

Language with explicit memory management
Verifying an on-the-fly garbage collector in the context of verified compilation

Memory managed language

Program $p$

Injection of the GC

Language with explicit memory management

Program compile($p$)

Observational refinement

$\forall P P' \text{ obs,}$

compiler $P = \text{OK } P' \land$

low_exec $P' \text{ obs } \Rightarrow$

high_exec $P \text{ obs}$
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
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)
        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

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

\[
\begin{align*}
\text{t1} & : p.\text{push}(1) \quad q.\text{pop}() \\
\text{t2} & : q.\text{push}(1) \\
\text{t3} & : p.\text{push}(2) \quad p.\text{pop}()
\end{align*}
\]
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
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

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

```plaintext
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

\[
\text{Obs}(T(p)) \subseteq \text{Obs}(p)
\]

if \(x.\text{color} = \text{WHITE}\) then
\[
\text{push(buffer}[m], x)
\]

if \(x.\text{color} = \text{WHITE}\) then
\[
\text{nw} = m.\text{next}_\text{write} \\
\text{nr} = m.\text{next}_\text{read} \\
d = m.\text{data} \\
d[\text{nw}] = x \\
\text{nw} = (\text{nw}+1) \mod \text{SIZE} \\
\text{assume } (\text{nr} == \text{nw}) \\
m.\text{next}_\text{write} = \text{nw}
\]
Using our result: proving linearisability via Rely-Guarantee
Rely Guarantee reasoning

[Jones81]

\[ R, G, I \vdash \{ P \} \quad c \quad \{ Q \} \]

Environment
R: Rely
G: Guarantee
Global Correctness
Invariant
Annotations
Rely Guarantee reasoning

[Jones81]

\[
\begin{align*}
R, G, I & \vdash \{P\} c \{Q\} \\
\end{align*}
\]

Annotations

Environment
R: Rely
G: Guarantee

Global Correctness
Invariant
Rely Guarantee reasoning

[Jones81]
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

Rely Guarantee reasoning

[Jones81]

A thread is proved against a contract.
The notion of interference is checked against this contract.
Reasoning about linearisation using Rely-Guarantee


- Explicit annotation of *linearisation points*
- Hybrid states, both concrete and abstract
- Linearisation points trigger the abstract semantics
Reasoning about linearisation using Rely-Guarantee


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

```plaintext
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

\[
\begin{align*}
\text{local map} & \quad \rho_1 \\
\text{shared heap} & \quad \sigma_1 \\
\text{abstract data-structure} & \quad \pi_1
\end{align*}
\]

\[
\begin{align*}
nw &= m.\text{next}_\text{write} \\
nr &= m.\text{next}_\text{read} \\
d &= m.\text{data} \\
d[nw] &= x \\
nw &= (nw+1) \mod \text{SIZE} \\
\text{assume } (nr == nw) \\
<m.\text{next}_\text{write} = nw; \text{LIN}>
\end{align*}
\]
Reasoning about linearisation using Rely-Guarantee


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

```
local map: \rho_1 \rho_2 \rho_3 \rho_4 \rho_4 \rho_5 \rho_5

shared heap: \sigma_1 \sigma_1 \sigma_1 \sigma_1 \sigma_2 \sigma_2 \sigma_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 \text{SIZE}
assume (nr == nw)
<m.next_write = nw; LIN>
```
Reasoning about linearisation using Rely-Guarantee


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

\[
\begin{align*}
\text{local map} & \quad \rho_1 \quad \rho_2 \quad \rho_3 \quad \rho_4 \quad \rho_4 \quad \rho_5 \quad \rho_5 \quad \rho_5 \\
\text{shared heap} & \quad \sigma_1 \quad \sigma_1 \quad \sigma_1 \quad \sigma_1 \quad \sigma_2 \quad \sigma_2 \quad \sigma_2 \quad \sigma_3 \\
\text{abstract data-structure} & \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_2
\end{align*}
\]

\[
\begin{align*}
\text{nw} & = \text{m.next}_\text{write} \\
\text{nr} & = \text{m.next}_\text{read} \\
\text{d} & = \text{m.data} \\
\text{d}[\text{nw}] & = \text{x} \\
\text{nw} & = (\text{nw}+1) \mod \text{SIZE} \\
\text{assume} & \ (\text{nr} == \text{nw}) \\
<\text{m.next}_\text{write} = \text{nw} ; \text{LIN}> \\
\end{align*}
\]
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

\[
\begin{align*}
\text{local map} & \quad \rho_1 \quad \rho_2 \quad \rho_3 \quad \rho_4 \quad \rho_4 \quad \rho_5 \quad \rho_5 \quad \rho_5 \\
\text{shared heap} & \quad \sigma_1 \quad \sigma_1 \quad \sigma_1 \quad \sigma_1 \quad \sigma_2 \quad \sigma_2 \quad \sigma_2 \quad \sigma_3 \\
\text{abstract} & \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_1 \quad \rho_2 \\
\text{data-structure} & \quad B \quad B \quad B \quad B \quad B \quad B \quad B \quad A(v) \\
\end{align*}
\]

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

Concrete implementation of methods

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 $I_c$

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

$nw = m\text{.next\_write}$
$nr = m\text{.next\_read}$
$d = m\text{.data}$
$d[nw] = x$
$nw = (nw+1) \mod SIZE$
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 $R_m \quad G_m$

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>

next_read
next_write
data

SIZE-1
Proving linearisability: the perspective of a user

Abstract data structure
Concrete implementation of methods
Coherence invariant $I_c$
Relies and guarantees $R_m G_m$

Buf := Empty | Cons x b  \(\text{b.Push(x) = Cons x b}\)

\(\text{nw} = \text{m.next\_write}\)
\(\text{nr} = \text{m.next\_read}\)
\(d = \text{m.data}\)
\(d[\text{nw}] = x\)
\(\text{nw} = (\text{nw}+1) \mod \text{SIZE}\)
assume (\(\text{nr} == \text{nw}\))
\(<\text{m.next\_write} = \text{nw}; \text{LIN}>\)
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 $R_m \ G_m$

\[
\begin{align*}
R_{push}, G_{push}, I_c & \vdash \\
\{ \ln = B \} & \\
p.push(v) & \\
\{ \ln = A(v_1) \land \text{ret} = v_1 \} 
\end{align*}
\]
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

$R_{push}, G_{push}, I_c \vdash$

$\{ \text{ln} = B \}$

$p.push(v)$

$\{ \text{ln} = A(v_1) \land \text{ret} = v_1 \}$

$I_c$ 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

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

$I_c$ 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 $R_m, G_m$

RG method specification
Stability obligations
RG consistency

Reasoning locally exclusively on methods
Automatically obtain
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

\[ \sim \quad \text{Relation between states of the source and target language} \]

\[
\begin{array}{c}
S_1' \quad \overset{O}{\longrightarrow}^* \quad S_2' \\
\sim \quad \vdots \\
S_1 \quad \overset{O}{\longrightarrow} \quad S_2
\end{array}
\]
Backward simulations

Inductive step used to prove observational refinement

\[ \sim \quad \text{Relation between states of the source and target language} \]

\[
s'_1 \xrightarrow{O} * \xrightarrow{O} \sim s'_2
\]

\[
s_1 \xrightarrow{O} s_2
\]

\[
p.\text{push}(x)
\]

\[
\begin{align*}
\text{nw} &= \text{p.next}_\text{write} \\
\text{nr} &= \text{p.next}_\text{read} \\
d &= \text{p.data} \\
d[\text{nw}] &= x \\
\text{nw} &= (\text{nw}+1) \mod \text{SIZE} \\
\text{assume} (\text{nr} == \text{nw}) \\
p.\text{next}_\text{write} &= \text{nw}
\end{align*}
\]
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 resources.

\[
[r \mapsto v] = \{h \mid h(r) = v \land \text{dom}(h) = \{r\}\}
\]

Achieves great modularity through the \textit{frame rule}

\[
\vdash \{P\} c \{Q\} \\
\vdash \{P * R\} c \{Q * R\}
\]

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