

# Explicit Stabilisation

## for modular Rely-Guarantee reasoning

John Wickerson, Mike Dodds and Matthew Parkinson



We're interested in proving the safety of imperative programs, both concurrent and sequential. The 'Explicit Stabilisation' technique that I'm going to propose isn't intended to help you prove more programs. But it will help you prove programs more \*modularly\*.

# Talk outline

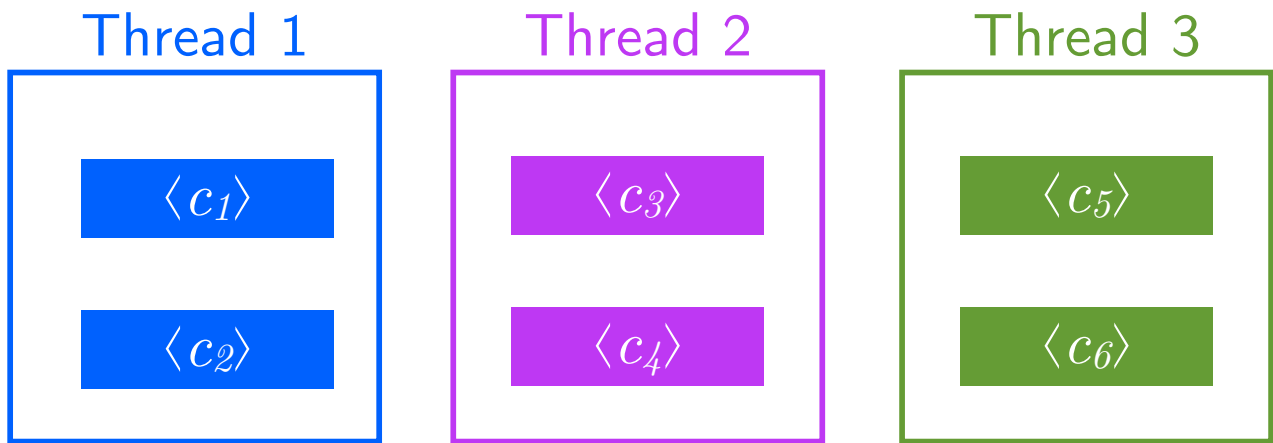
- ▶ Background: Rely-Guarantee
- ▶ Two challenges:
  - ▶ Modular specifications for concurrent libraries
  - ▶ Modular specification for a memory manager
- ▶ The solution: Explicit Stabilisation

2

I'll begin with an introduction to the Rely-Guarantee method, and what stability means. Then we'll look at two problems with the modularity of Rely-Guarantee, that will appear to you quite disparate.

Then we'll introduce our new approach to stability, called Explicit Stabilisation, and see how it tackles both of these problems.

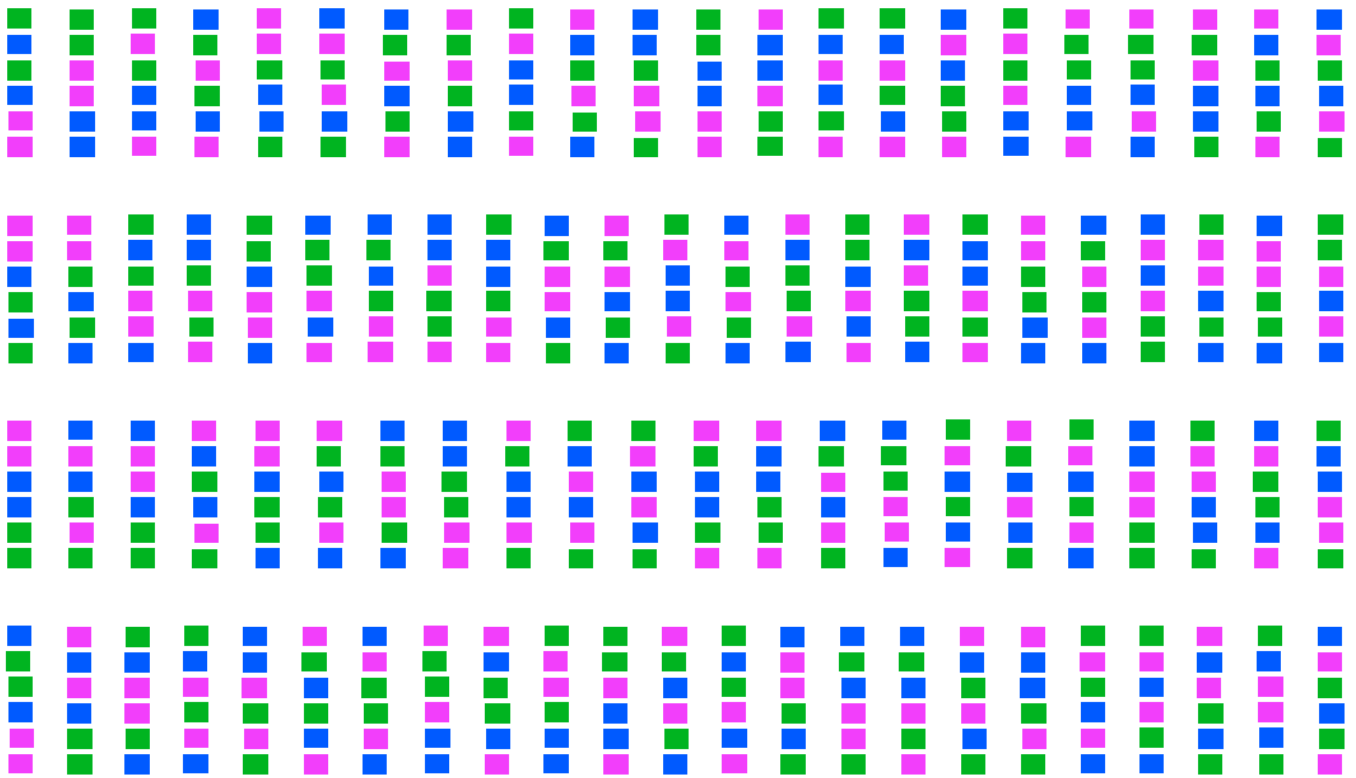
# Rely-Guarantee



3

As a gentle introduction consider a concurrent program with three threads, each executing two atomic instructions. This tiny program can be sequentialised in 90 different ways!

# Rely-Guarantee

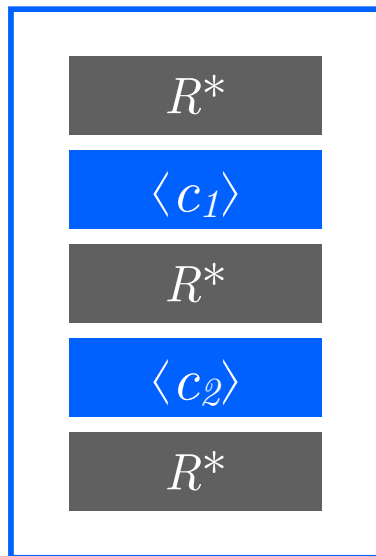


4

The rely-guarantee method tames this combinatorial explosion through abstraction. We treat each thread as being of the following form...

# Rely-Guarantee

Thread 1



- ▶  $G$  = set of all the state transitions the thread can do
- ▶  $R$  = set of all the state transitions other threads can do (i.e. union of all their  $G$ s)

5

... insert between each command an instruction that represents an arbitrary sequence of instructions by other threads.

More formally,  $G$  is the set of all state transitions a thread can do, then form  $R$  by combining all the other threads' guarantees. This gives the set of all state transitions the other threads can do, and we insert the reflexive transitive closure of that between each command.

The postcondition of  $c_1$ , if it is to be used as the precondition for  $c_2$ , must be "stable" under  $R$ , that is, the transitions in  $R$  must preserve its validity. This is manifested in the following proof rule...

# Rely-Guarantee

- ▶ Axiom for basic commands:

$$\frac{\begin{array}{l} \vdash \{p\} c \{q\} \quad c \subseteq G \\ p \text{ stable } R \quad q \text{ stable } R \end{array}}{R, G \vdash \{p\} c \{q\}}$$

- ▶ Operational Semantics:

$$\frac{(\sigma, \sigma') \in R}{\langle C, \sigma \rangle \xrightarrow{R} \langle C, \sigma' \rangle}$$

6

This is a rule for these basic, atomic instructions (like assignment). Start with a sequential spec for  $c$ . Ensure that any state transition  $c$  can do is within its guarantee (so its guarantee is valid). Ensure that the pre and postcondition are stable under  $R$ . The meaning of this judgement is the same as the usual Hoare triple,  $\{p\} C \{q\}$ , plus, the environment may do any state transition in  $R$ , and we may do any state transition in  $G$ .

Just a brief comment on how RG manifests in the operational semantics: the small-step transition relation is now parameterised by the rely, and we have this rule that simulates environmental interference, whereby the environment can change the state under our feet while we're evaluating command  $C$ .

# Challenge 1:

Specifications for concurrent libraries  
that can be used for any client

Going to use RG to specify a concurrent library.  
Traditionally, this is something the Rely-Guarantee  
method can't do. We want a *\*single\** specification for  
our library that can be used to verify all of its clients.

# A library function

Library:

$\text{foo}() \stackrel{\text{def}}{=} \langle x++ \rangle$

A client:

```
assume(x = 3)
  foo()
assert(x = 4)
```

rely:  $\emptyset$ , guar:  $x++$   
 $\vdash \{x = n\} \text{foo}() \{x = n + 1\}$

Another client:

```
assume(x = 3)
(foo() || ⟨x++⟩)
assert(x ≥ 4)
```

rely:  $x++$ , guar:  $x++$   
 $\vdash \{x \geq n\} \text{foo}() \{x \geq n + 1\}$

Yet another client:

```
assume(x = 3)
(foo() || ⟨x--⟩)
assert(x ≤ 4)
```

rely:  $x--$ , guar:  $x++$   
 $\vdash \{x \leq n\} \text{foo}() \{x \leq n + 1\}$

8

Explain the stabilisation: start with postcondition  $x=n+1$ , but need to weaken it until it's stable under the rely that might increment  $x$ .

# Most general spec?

rely:  $\emptyset$ , guar:  $x++$   
 $\vdash \{x = n\} \text{foo}() \{x = n + 1\}$

WEAKEN

$R', G' \vdash \{p'\} C \{q'\}$

$p \Rightarrow p' \quad q' \Rightarrow q$

$R \subseteq R' \quad G' \subseteq G$

---

$R, G \vdash \{p\} C \{q\}$

rely:  $x++$ , guar:  $x++$   
 $\vdash \{x \geq n\} \text{foo}() \{x \geq n + 1\}$

rely:  $x--$ , guar:  $x++$   
 $\vdash \{x \leq n\} \text{foo}() \{x \leq n + 1\}$

9

Which of these should be stored as the specification for foo? We need the biggest rely (because the Weaken rule only makes that smaller) but also the strongest postcondition (which the Weaken rule only makes weaker). But as the rely gets bigger, the postcondition has to get *\*WEAKER\**. Oh dear. This is why R/G cannot verify library code!

# Challenge 2:

Specifying a module while hiding  
“internal interference” from clients

This time a sequential module, so forget about concurrency. Tackling the other component of modularity, which is information hiding.

Up until now we've looked at some very simple programs, just manipulating global variables. Now we're going to concentrate on programs that manipulate the heap, so we're going to be calling on some more modern verification techniques, including Separation Logic.

# A memory manager

- ▶ Case study: memory manager from Version 7 Unix (1979)
- ▶ First formal safety proof...
  - ▶ ... and we discovered a bug!

You get the bug if malloc fails, which leaves a pointer pointing into the middle of a block, and then do a successful call to malloc, which follows that pointer, which could give a segfault.

# Specifying the manager

## 1. VERBALLY

The `malloc(nb)` function allocates *nb* bytes of memory and returns a pointer to the allocated memory.

The `free(ptr)` function deallocates the memory allocation pointed to by *ptr*. If *ptr* is a NULL pointer, no operation is performed.

Assume malloc accepts a number of words.

# Specifying the manager

## 2. UNSOUNDLY

$$\{\text{emp}\} x := \text{malloc}(n) \{x \mapsto\_ * \dots * x+n-1 \mapsto\_ \}$$
$$\{x \mapsto\_ * \dots * x+n-1 \mapsto\_ \} \text{free}(x) \{\text{emp}\}$$

13

Just for the purposes of this talk, we'll keep things simple and pretend all calls to malloc are successful – just include an extra disjunction in the postcondition to model failure.

Problem: free's precondition is too weak. Only works when the block was allocated by malloc, not just on any old sequence of memory locations.

# Specifying the manager

## 3. USING A 'TOKEN'

$$\{\text{emp}\} \ x := \text{malloc}(n) \left\{ \begin{array}{l} \text{token}(x, n) \\ *x \mapsto \_ \ * \dots * \ x+n-1 \mapsto \_ \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{token}(x, n) \\ *x \mapsto \_ \ * \dots * \ x+n-1 \mapsto \_ \end{array} \right\} \text{free}(x) \ \{\text{emp}\}$$

Malloc returns a conceptual token, stating the address and size of the block. We arrange that these tokens only come from malloc and can't be duplicated -- details of how that's done are in the paper, plus a concrete definition for token. This is then used to prove to the free routine that the block is valid.

# Specifying the manager

## 4. INCLUDING INTERNAL STATE

$$\left\{ \boxed{\text{arena}} \right\} x := \text{malloc}(n) \left\{ \begin{array}{l} \boxed{\text{arena-with-gap}(x, n)} \\ * \text{token}(x, n) \\ * x \mapsto \_ * \dots * x+n-1 \mapsto \_ \end{array} \right\}$$
$$\left\{ \begin{array}{l} \boxed{\text{arena-with-gap}(x, n)} \\ * \text{token}(x, n) \\ * x \mapsto \_ * \dots * x+n-1 \mapsto \_ \end{array} \right\} \text{free}(x) \left\{ \boxed{\text{arena}} \right\}$$

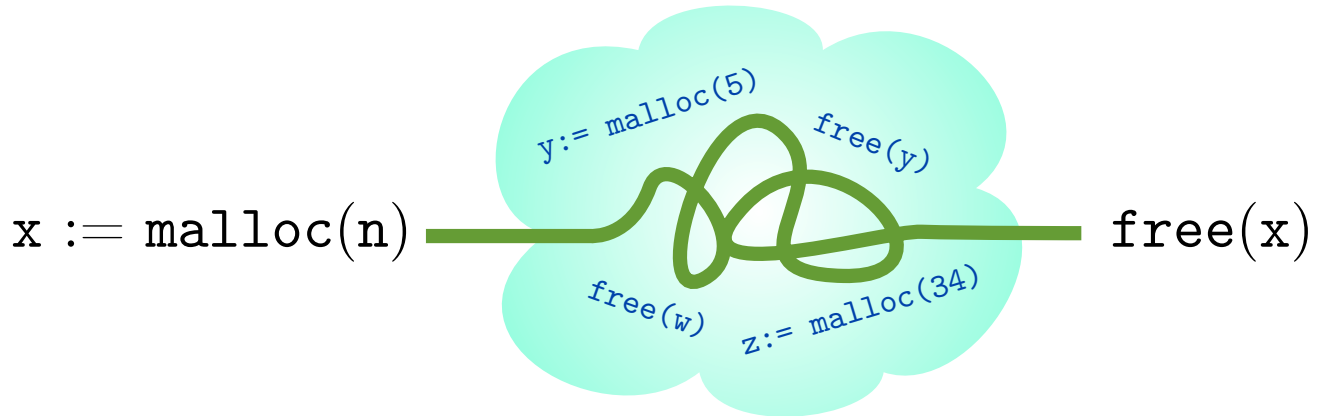
15

It currently looks like the blocks are created out of thin air. In fact, what happens is that the block's ownership transfers between the memory manager and the caller. Let's make that explicit.

The boxes are part of the syntax of RGSep. They allow you to conjoin several of these boxes together to describe one arena with several gaps in it.

Now, the thing about describing the internal state of the memory manager is: how do we know that the arena will still have a gap (in the right place, and of the right size) when we come to call free? That is...

# Crux of the proof



16

...will the gap survive?

This is a matter of stability (even in this sequential setting): Is arena-with-gap *\*stable\** under the actions of malloc and free?

Cloud contains calls to malloc and free – other calls won't change the internal state.

# Malloc specification

$\{ arena \}$

$x := \text{malloc}(n)$

**unstable**  $\left\{ \begin{array}{l} \boxed{arena-with-gap(x, n)} \\ * token(x, n) \\ * x \mapsto \_ \quad * \dots * \quad x+n-1 \mapsto \_ \end{array} \right\}$

Perhaps someone might free the block?

# Malloc specification

$\{ arena \}$

$x := \text{malloc}(n)$

**stable**  $\left\{ \begin{array}{l} arena\text{-with-gap}(x, n) \\ * token(x, n) \\ * x \mapsto \_ \quad * \dots * \quad x+n-1 \mapsto \_ \end{array} \right\}$

18

However, the presence of the token in our local state means that no other client can free our block, because one of the preconditions of the free routine is the relevant token!

So when accompanied by a token, the arena-with-gap predicate *is* stable. This is interesting – we have a piece of the module’s internal state, and a piece of my local state, and only together are they stable.

Problem: we don’t want the client to have to do this reasoning. And they will have to: if they change the block, is the assertion still stable? Well yes, but that’s only because we’ve just worked out that stability only depends on the first two parts. The client shouldn’t have to do this reasoning. And, by the principle of information hiding, they shouldn’t even be *able* to. Can we capture this stability argument *in* the assertion: state that this bit is crucial for stability, so don’t mess with it, but you can do whatever you like with this other bit?

# Explicit Stabilisation

# Explicit Stabilisation

$$\frac{\begin{array}{l} \vdash \{p\} c \{q\} \quad c \subseteq G \\ p \text{ stable } R \quad q \text{ stable } R \end{array}}{R, G \vdash \{p\} c \{q\}}$$

What do we do if  $p$  is not stable under  $R$ ? We stabilise it. Make it stronger until it's stable under  $R$ . Make  $q$  weaker until it's stable under  $R$ .

# Explicit Stabilisation

$$\frac{\vdash \{p\} c \{q\} \quad c \subseteq G}{R, G \vdash \{[p]_R\} c \{[q]_R\}}$$

21

This rule encodes that. Make  $p$  stronger until it's stable under  $R$ . Make  $q$  weaker until it's stable under  $R$ .

We make these stabilisations first-class operators; part of the syntax of assertions. We call it explicit stabilisation.

One immediate benefit: fewer side conditions.

Also: allows lazy evaluation of stability, just as an efficient implementation of these rules would do. We can stack up stabilisations and collapse them together using the equational properties that we'll look at shortly.

# Explicit Stabilisation

- ▶ Syntax:

$p ::= \dots \mid \lfloor p \rfloor_R \mid \lceil p \rceil_R$  **all reachable states also satisfy  $p$**

- ▶ Semantics:

$\sigma \models \lfloor p \rfloor_R \Leftrightarrow \forall \sigma'. (\sigma, \sigma') \in R^* \Rightarrow \sigma' \models p$

$\sigma \models \lceil p \rceil_R \Leftrightarrow \exists \sigma'. \sigma' \models p \wedge (\sigma', \sigma) \in R^*$

**is reachable from some state that satisfies  $p$**

# Stabilisation

- ▶ *Quiz.* Stabilise the assertions  $x=0$  and  $x \neq 0$  under the rely  $R$  that can increment  $x$ .

$$[x = 0]_R \Leftrightarrow \boxed{?}$$

$$[x \neq 0]_R \Leftrightarrow \boxed{?}$$

$$[x = 0]_R \Leftrightarrow \boxed{?}$$

$$[x \neq 0]_R \Leftrightarrow \boxed{?}$$

# A few properties

- ▶ if  $p$  stable  $R$  then  $\lfloor p \rfloor_R \Leftrightarrow \lceil p \rceil_R \Leftrightarrow p$
- ▶ if  $R \subseteq R'$  then:
  - ▶  $\lfloor \lfloor p \rfloor_R \rfloor_{R'} \Leftrightarrow \lfloor \lfloor p \rfloor_{R'} \rfloor_R \Leftrightarrow \lceil \lceil p \rceil_{R'} \rceil_R \Leftrightarrow \lceil p \rceil_{R'}$
  - ▶  $\lceil \lceil p \rceil_R \rceil_{R'} \Leftrightarrow \lceil \lceil p \rceil_{R'} \rceil_R \Leftrightarrow \lfloor \lfloor p \rfloor_{R'} \rfloor_R \Leftrightarrow \lfloor p \rfloor_{R'}$

1: If assertion is already stable, stabilisation has no effect. \*In fact this one's an if-and-only-if.

2. Stacked-up stabilisations. Behaves like floor and ceiling in arithmetic.

# **Solution to Challenge 1:**

Specifications for concurrent libraries  
that can be used for any client

# Most general spec?

rely:  $R$ , guar:  $x++$   
 $\vdash \{[x = n]_R\} \text{foo}() \{[x = n + 1]_R\}$

rely:  $\emptyset$ , guar:  $x++$   
 $\vdash \{x = n\} \text{foo}() \{x = n + 1\}$

rely:  $x++$ , guar:  $x++$   
 $\vdash \{x \geq n\} \text{foo}() \{x \geq n + 1\}$

rely:  $x--$ , guar:  $x++$   
 $\vdash \{x \leq n\} \text{foo}() \{x \leq n + 1\}$

26

Using explicit stabilisation, we can use an arbitrary rely (subject to a few constraints detailed in the paper), which we then refer to (using ceiling) in the pre- and postcondition. When  $R$  is empty, the stabilisation has no effect, and we get the first spec. When  $R$  is  $x++$  or  $x--$ , the other two specs can be derived.

Note the separation of the effect of  $\text{foo}()$  from the effect of the environment.

# **Solution to Challenge 2:**

Specifying a module while hiding  
“internal interference” from clients

# Malloc specification

$$\{ \boxed{\textit{arena}} \}$$
$$x := \text{malloc}(n)$$
$$\left\{ \begin{array}{l} \boxed{\textit{arena-with-gap}(x, n)} \\ * \textit{token}(x, n) \\ * x \mapsto \_ \quad * \dots * \quad x+n-1 \mapsto \_ \end{array} \right\}$$

28

Let  $G$  be all the actions that malloc and free can do. It is the relation under which these assertions must be stable. We've worked out that these assertions are stable under  $G$ , and that the stability depended only upon the *arena-with-gap* and the *token*. But how do we tell the client this?

# Malloc specification

$$\{ \boxed{\text{arena}} \}_G$$
$$x := \text{malloc}(n)$$

Treat  $G$  like an abstract relation

$$\left\{ \begin{array}{l} \boxed{\text{arena-with-gap}(x, n)} \\ * \text{token}(x, n) \\ * x \mapsto \_ * \dots * x+n-1 \mapsto \_ \end{array} \right\}_G$$

29

We can encode this fact using explicit stabilisation.

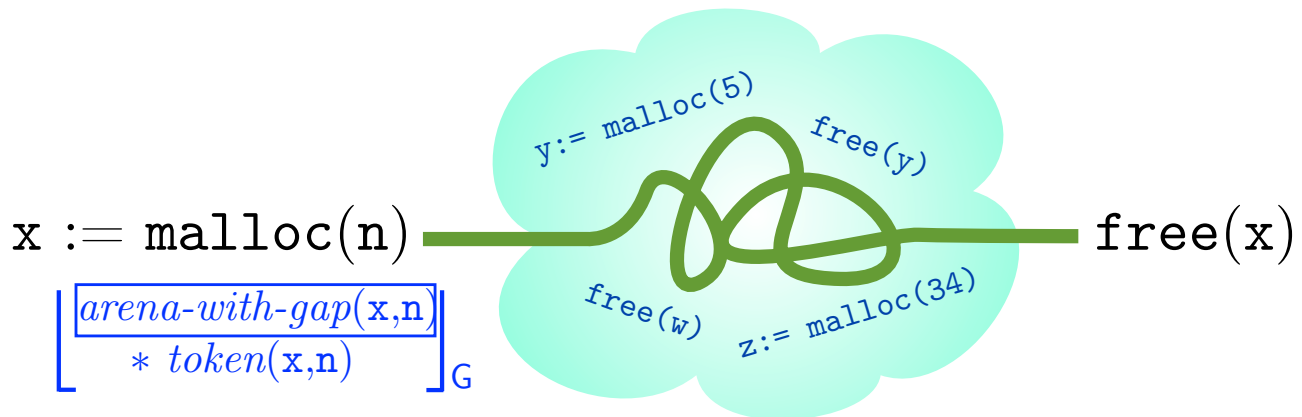
We wrap the assertion in the stabilisation brackets. We're identifying that part of the assertion that is crucial to the stability of the overall assertion.

We're using the strengthening stabiliser in the postcondition -- this is only sound when it has no effect, i.e. when its operand is already stable!

Note that the contents of the block lies outside the stabilisation, so the client can fiddle with that however it likes, but it shouldn't go inside the stabilisation brackets, otherwise it would have to recalculate stability.

Note that  $G$  can now be treated as an abstract relation, just like  $\text{arena}$  and  $\text{token}$  are abstract predicates. The client doesn't need to know its value; just that  $G$  represents the actions of  $\text{malloc}$  and  $\text{free}$ , and that this assertion is stable under  $G$ .

# Crux of the proof



So in answer to our earlier question – how to get the postcondition of malloc safely to reach the precondition of free – we need to partner the arena-with-gap predicate with a token, and then wrap it in the explicit stabilisation operator (which tells the client, don't touch inside here!), and it will be safe for the journey.

# Summary

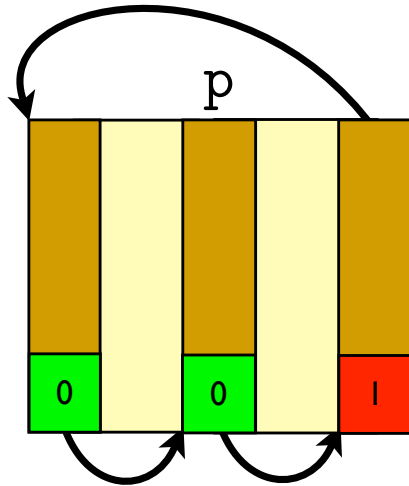
- ▶ We have extended RG reasoning with ‘explicit stabilisation’, which lets us...
  - ▶ Verify libraries without knowing clients’ environments
  - ▶ Verify a module’s clients without revealing the module’s ‘internal interference’

So: two forms of modularity, which we’ve teased apart and tackled separately. Perhaps in future work we might be able to blend the two solutions together.

# Extra slides

- ▶ The bug
- ▶ E.S. as fixed points / weakest pre
- ▶ The arena, graphically
- ▶ Definition of token
- ▶ Actions
- ▶ Deny-Guarantee

# The bug



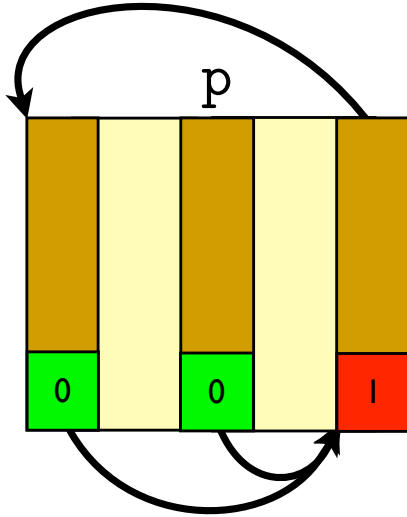
```
malloc(40287);
```

33

This bug was discovered as a direct consequence of the verification process – an invariant that we wanted didn't hold, and upon investigation as to why, we found this bug.

To illustrate, consider a very small arena comprising two free blocks. Then attempt a malloc that fails. The free blocks are coalesced, but the victim pointer, *p*, is neglected.

# The bug

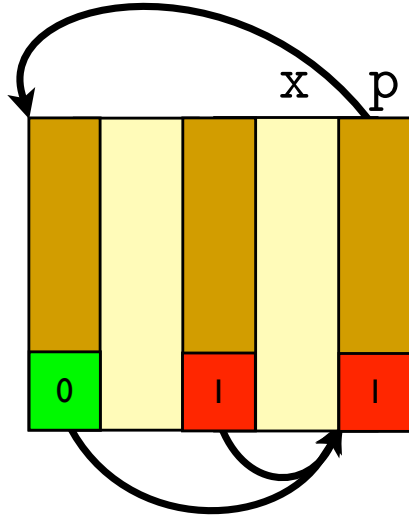


```
x := malloc(1);
```

34

Then do a successful malloc, which corrects the victim pointer.

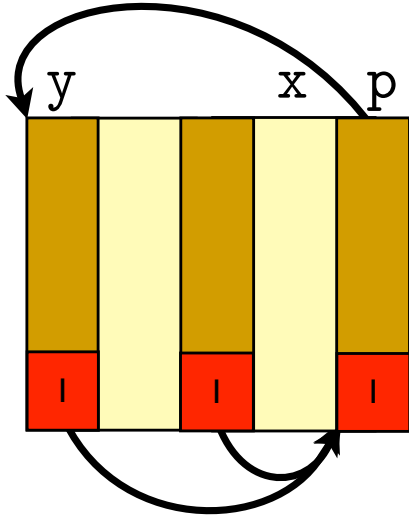
# The bug



```
y := malloc(3);
```

Do another successful malloc.

# The bug



36

Now the block at *y* includes the pointer of the block at *x*. So you can modify that pointer, then free *x*, and you'll get a segfault.

# Explicit Stabilisation

- ▶ Can be thought of as fixed-points:

$$\lfloor p \rfloor_R = \bigvee \{q \mid q \Rightarrow p \wedge q \text{ stab } R\}$$

$$\lceil p \rceil_R = \bigwedge \{q \mid p \Rightarrow q \wedge q \text{ stab } R\}$$

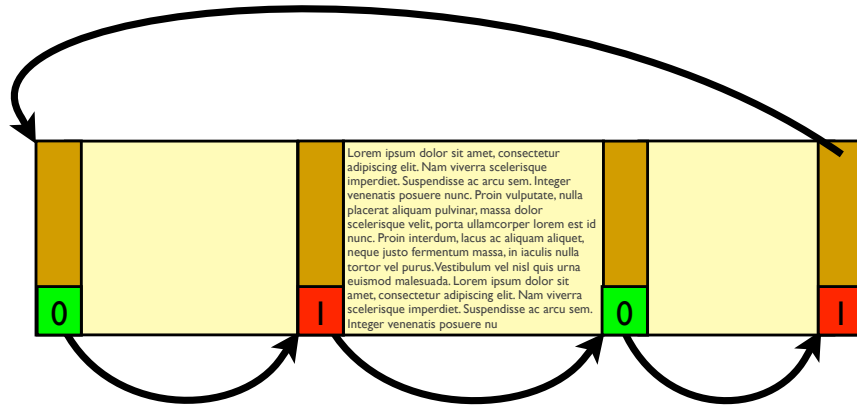
- ▶ Or as predicate transformers:

$$\{p\} R^* \{\lceil p \rceil_R\} \quad \{\lfloor p \rfloor_R\} R^* \{p\}$$

Floor is big OR-ing of all the stronger, stable assertions.

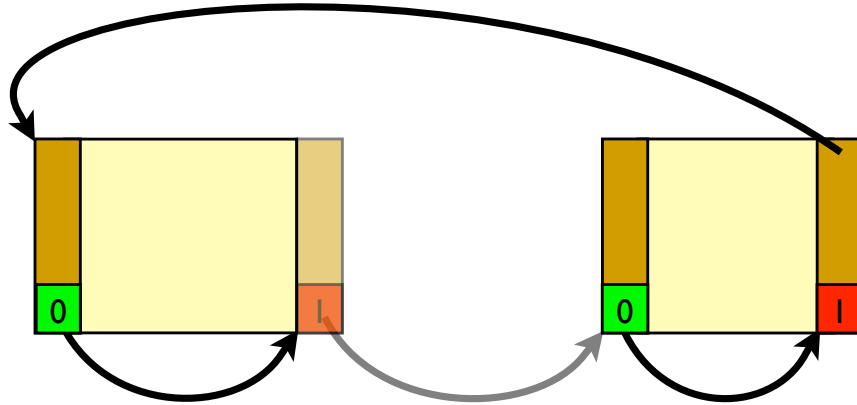
Ceiling is big AND-ing of all the weaker, stable assertions.

# The arena



Let's have a look at it pictorially: here's the arena: a contiguous linked-list. The lowest bits of the pointers are used as status bits: 0 means the next block is free and 1 means it's allocated.

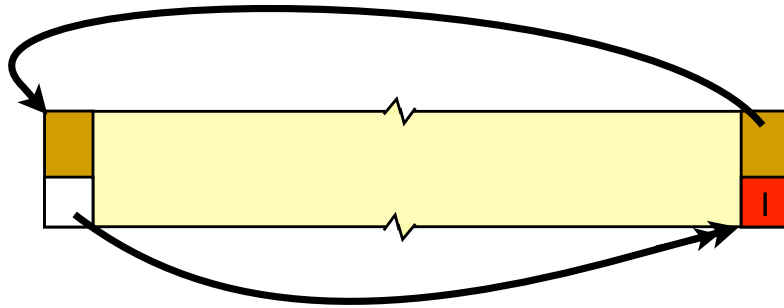
# The arena



$$token(x, n) = (x - 1) \overset{.5}{\mapsto} (x + n)$$

We can implement the token as half-permission on a block's pointer (the memory manager needs to keep the other half for later traversals of the list). This implementation is not exposed to the client.

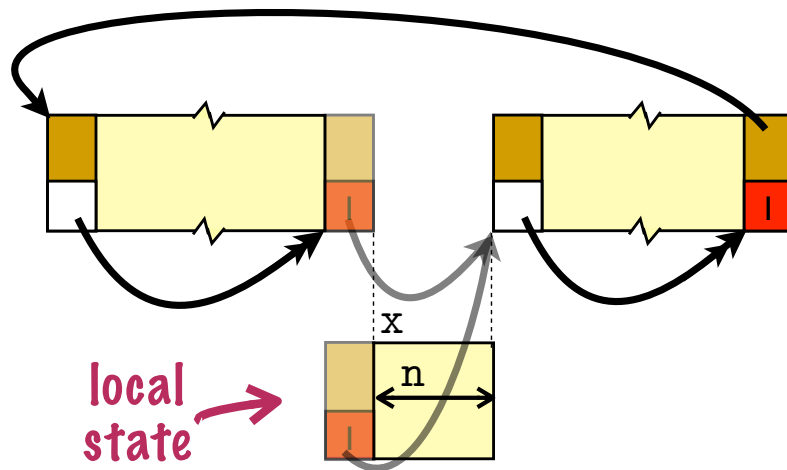
# Malloc precondition



40

Here's malloc's precondition, pictorially. The double-headed arrow means a sequence of pointers, and the zigzag abbreviates a sequence of blocks.

# Malloc postcondition



41

Afterwards, one block has been transferred into the client's local state. We also transfer half permission on the block's pointer (as the token). We leave behind an "arena-with-gap" predicate, which we need to show to be stable under the actions of other calls to malloc and free.

# Actions

Coalesce:



AllocatePart:



AllocateWhole:



Free:



42

These actions are:

1. Coalescing adjacent free blocks.
2. Allocating part or the whole of a block.
3. Freeing a block.

Problem: the arena-with-gap predicate is not stable under the Free action, which is potentially able to fill in the gap.

# Deny-Guarantee

- ▶ Dodds et al. (ESOP 2009)
- ▶ Fork rule, originally:

$$\frac{\begin{array}{l} \vdash \{p_{\text{give}}\} C \{q\} \\ p_{\text{keep}} \Rightarrow \text{canwrite}(x) \\ \text{thread}(x, q) * p_{\text{keep}} \Rightarrow q' \end{array}}{\vdash \{p_{\text{give}} * p_{\text{keep}}\} x := \text{fork}(C) \{q'\}}$$

43

In DG, assertions both describe the state and contain permissions to perform reads and writes. I have  $p_{\text{give}}$  and  $p_{\text{keep}}$ . I spawn off a thread, and give it  $p_{\text{give}}$ . I write its thread identifier into  $x$  (provided  $p_{\text{keep}}$  contains sufficient permission to do so). Later I'll call  $\text{join}(x)$ , and get back the assertion  $q$ . The  $q'$  assertion is simply a weaker, stable assertion (stability is an implicit side condition on all assertions). We could use the ceiling operator to neaten this up...

# Deny-Guarantee

- ▶ Fork rule with explicit stabilisation:

$$\frac{\begin{array}{c} \vdash \{p_{\text{give}}\} C \{q\} \\ p_{\text{keep}} \Rightarrow \text{canwrite}(x) \end{array}}{\vdash \{p_{\text{give}} * p_{\text{keep}}\} x := \text{fork}(C) \{ \lceil \text{thread}(x, q) * p_{\text{keep}} \rceil \}}$$

... like so.