Why Separation Logic is the Bee's Knees, and why you should care

Richard Bornat
School of Computing, Middlesex University

8th December 2005 (version 3.0)

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.
- "*What has formalism ever done for us?*" (Mark Woodman, Professor in (sic) Information Technology, Middlesex University, 2003).

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.
- "*What has formalism ever done for us?*" (Mark Woodman, Professor in (sic) Information Technology, Middlesex University, 2003).
- Computing is a collision between formalism (mathematical logic) and calculation (arithmetic).

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.
- "*What has formalism ever done for us?*" (Mark Woodman, Professor in (sic) Information Technology, Middlesex University, 2003).
- Computing is a collision between formalism (mathematical logic) and calculation (arithmetic).
- Programs are utterly formal, completely meaningless texts.

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.
- "*What has formalism ever done for us?*" (Mark Woodman, Professor in (sic) Information Technology, Middlesex University, 2003).
- Computing is a collision between formalism (mathematical logic) and calculation (arithmetic).
- Programs are utterly formal, completely meaningless texts.
- Programs are hard to write, but once written easy to compile and to execute.

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.
- “*What has formalism ever done for us?*” (Mark Woodman, Professor in (sic) Information Technology, Middlesex University, 2003).
- Computing is a collision between formalism (mathematical logic) and calculation (arithmetic).
- Programs are utterly formal, completely meaningless texts.
- Programs are hard to write, but once written easy to compile and to execute.
- Proofs are hard to write, but once written easy to check.

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.
- "*What has formalism ever done for us?*" (Mark Woodman, Professor in (sic) Information Technology, Middlesex University, 2003).
- Computing is a collision between formalism (mathematical logic) and calculation (arithmetic).
- Programs are utterly formal, completely meaningless texts.
- Programs are hard to write, but once written easy to compile and to execute.
- Proofs are hard to write, but once written easy to check.
- Computing is everything you can do with formalism.

The dawn of computing

- In Bletchley (Turing, Flowers) in 1942, and in Germany (Zuse) about the same time.
- "*What has formalism ever done for us?*" (Mark Woodman, Professor in (sic) Information Technology, Middlesex University, 2003).
- Computing is a collision between formalism (mathematical logic) and calculation (arithmetic).
- Programs are utterly formal, completely meaningless texts.
- Programs are hard to write, but once written easy to compile and to execute.
- Proofs are hard to write, but once written easy to check.
- Computing is everything you can do with formalism.
- Advances in computing are advances in formalism, and vice-versa.

*Programming is **really** hard: only nine lines, no CAS, and you **still** can't understand it*

Programming is really hard: only nine lines, no CAS, and you still can't understand it

```
var   reading, latest : bit
      slot : array bit of bit
      data : array bit of array bit of datatype

procedure   write    (item : datatype);
            var      pair, index : bit;
            begin
                     pair := not(reading);
                     index := not(slot[pair]);
                     data[pair, index] := item;
                     slot[pair] := index;
                     latest := pair
            end;

procedure   read :   datatype;
            var      pair, index : bit;
            begin
                     pair := latest;
                     reading := pair;
                     index := slot[pair];
                     read := data[pair, index]
            end;
```

The dawn of high-level programming and compilers

- In Chicago in 1953 (Backus).

The dawn of high-level programming and compilers

- In Chicago in 1953 (Backus).
- Instead of writing programs, we were to write FORmulas which the compiler would TRANslate into a program.

The dawn of high-level programming and compilers

- In Chicago in 1953 (Backus).
- Instead of writing programs, we were to write FORmulas which the compiler would TRANslate into a program.
- John Reynolds thought this was magic: a computer writing a program!

The dawn of high-level programming and compilers

- In Chicago in 1953 (Backus).
- Instead of writing programs, we were to write FORmulas which the compiler would TRANslate into a program.
- John Reynolds thought this was magic: a computer writing a program!
- Despite the inventors' best intentions, programs got bigger, not better.

The dawn of high-level programming and compilers

- In Chicago in 1953 (Backus).
- Instead of writing programs, we were to write FORmulas which the compiler would TRANslate into a program.
- John Reynolds thought this was magic: a computer writing a program!
- Despite the inventors’ best intentions, programs got bigger, not better.
- (Programming is absolutely as hard as we dare make it, and always will be.)

The dawn of high-level programming and compilers

- In Chicago in 1953 (Backus).
- Instead of writing programs, we were to write FORmulas which the compiler would TRANslate into a program.
- John Reynolds thought this was magic: a computer writing a program!
- Despite the inventors' best intentions, programs got bigger, not better.
- (Programming is absolutely as hard as we dare make it, and always will be.)
- (Concurrent programming programs are small: it's no coincidence.)

The looming of concurrency

- (‘Looming’ is the light seen on the sea horizon when an island or land mass is just about to come into view. That’s not in the OAD!)

The looming of concurrency

- (‘Looming’ is the light seen on the sea horizon when an island or land mass is just about to come into view. That’s not in the OAD!)
- Atlas interrupts started it; time-sharing continued it.

The looming of concurrency

- (‘Looming’ is the light seen on the sea horizon when an island or land mass is just about to come into view. That’s not in the OAD!)
- Atlas interrupts started it; time-sharing continued it.
- It loomed in 1968, in Eindhoven, in the THE operating system (Dijkstra et al.).

The looming of concurrency

- ('Looming' is the light seen on the sea horizon when an island or land mass is just about to come into view. That's not in the OAD!)
- Atlas interrupts started it; time-sharing continued it.
- It loomed in 1968, in Eindhoven, in the THE operating system (Dijkstra et al.).
- "*We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other.*" (EWD)

The looming of concurrency

- ('Looming' is the light seen on the sea horizon when an island or land mass is just about to come into view. That's not in the OAD!)
- Atlas interrupts started it; time-sharing continued it.
- It loomed in 1968, in Eindhoven, in the THE operating system (Dijkstra et al.).
- "*We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other.*" (EWD)
- This is *programming methodology*, advice to the wise. It is *not* formal support.

The looming of concurrency

- (‘Looming’ is the light seen on the sea horizon when an island or land mass is just about to come into view. That’s not in the OAD!)
- Atlas interrupts started it; time-sharing continued it.
- It loomed in 1968, in Eindhoven, in the THE operating system (Dijkstra et al.).
- “*We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other.*” (EWD)
- This is *programming methodology*, advice to the wise. It is *not* formal support.
- Concurrency became possible, using semaphores and critical sections, but remained almost impossibly difficult.

Looming a little closer

- Semaphores are hard to think about. Not every semaphore program has critical sections.

Looming a little closer

- Semaphores are hard to think about. Not every semaphore program has critical sections.
- Hoare's CCRs were impossible to implement but easy to understand; Hoare / Brinch-Hansen's monitors were easy to implement and understand.

Looming a little closer

- Semaphores are hard to think about. Not every semaphore program has critical sections.
- Hoare's CCRs were impossible to implement but easy to understand; Hoare / Brinch-Hansen's monitors were easy to implement and understand.
- Concurrency became straightforward, until the invention of Java.

Looming a little closer

- Semaphores are hard to think about. Not every semaphore program has critical sections.
- Hoare's CCRs were impossible to implement but easy to understand; Hoare / Brinch-Hansen's monitors were easy to implement and understand.
- Concurrency became straightforward, until the invention of Java.

- Milner's CCS and Hoare's CSP were attempts to re-engineer concurrency in terms of message passing and identifiable processes.

Looming a little closer

- Semaphores are hard to think about. Not every semaphore program has critical sections.
- Hoare's CCRs were impossible to implement but easy to understand; Hoare / Brinch-Hansen's monitors were easy to implement and understand.
- Concurrency became straightforward, until the invention of Java.

- Milner's CCS and Hoare's CSP were attempts to re-engineer concurrency in terms of message passing and identifiable processes.
- They were both impossible to use. They both rumble on in PhD theses, and will do so for ever.

Closer still

- CSP led to occam, a programming language (May et al., 1982?), and to Pascal-m, another programming language (Bornat & Abramsky, 1982-ish).

Closer still

- CSP led to occam, a programming language (May et al., 1982?), and to Pascal-m, another programming language (Bornat & Abramsky, 1982-ish).
- Message-passing concurrency is pretty easy, but you still get deadlocks and have to read dumps.

Closer still

- CSP led to occam, a programming language (May et al., 1982?), and to Pascal-m, another programming language (Bornat & Abramsky, 1982-ish).
- Message-passing concurrency is pretty easy, but you still get deadlocks and have to read dumps.
- And, because of Hoare, you couldn't use pointers.

Closer still

- CSP led to occam, a programming language (May et al., 1982?), and to Pascal-m, another programming language (Bornat & Abramsky, 1982-ish).
- Message-passing concurrency is pretty easy, but you still get deadlocks and have to read dumps.
- And, because of Hoare, you couldn't use pointers.
- This was in the Golden Age of programming languages (1958-85) when compilers found more than one error and syntax didn't make you ill. Then came the scourge of C and its bastard child Java, and darkness fell. But even the Java Wolf shall not eat the world for ever ...

Closer still

- CSP led to occam, a programming language (May et al., 1982?), and to Pascal-m, another programming language (Bornat & Abramsky, 1982-ish).
- Message-passing concurrency is pretty easy, but you still get deadlocks and have to read dumps.
- And, because of Hoare, you couldn't use pointers.
- This was in the Golden Age of programming languages (1958-85) when compilers found more than one error and syntax didn't make you ill. Then came the scourge of C and its bastard child Java, and darkness fell. But even the Java Wolf shall not eat the world for ever ...
- Steve Brookes has said sorry for failure semantics, and pointed out that if you use asynchronous message-passing and sort-of-infinite buffers, it all gets easier still. And I now know how to fix Pascal-m.

The dawn of Structured Programming

The dawn of Structured Programming

- Software Engineering was invented in 1968, and was an almost immediate complete bloody disaster.

The dawn of Structured Programming

- Software Engineering was invented in 1968, and was an almost immediate complete bloody disaster.
- In about 1974, Hoare and Dijkstra tried to repair the damage, by inventing Structured Programming.

The dawn of Structured Programming

- Software Engineering was invented in 1968, and was an almost immediate complete bloody disaster.
- In about 1974, Hoare and Dijkstra tried to repair the damage, by inventing Structured Programming.
- Only use program constructs which we understand and you can reason about: assignment (:=), sequence (;), condition (if-then-else-fi), iteration (while-do-od) and procedure call.

The dawn of Structured Programming

- Software Engineering was invented in 1968, and was an almost immediate complete bloody disaster.
- In about 1974, Hoare and Dijkstra tried to repair the damage, by inventing Structured Programming.
- Only use program constructs which we understand and you can reason about: assignment (:=), sequence (;), condition (if-then-else-fi), iteration (while-do-od) and procedure call.
- Do *not* use goto; do *not* commit aliasing.

The dawn of Structured Programming

- Software Engineering was invented in 1968, and was an almost immediate complete bloody disaster.
- In about 1974, Hoare and Dijkstra tried to repair the damage, by inventing Structured Programming.
- Only use program constructs which we understand and you can reason about: assignment (:=), sequence (;), condition (if-then-else-fi), iteration (while-do-od) and procedure call.
- Do *not* use goto; do *not* commit aliasing.
- Implicitly, use high-level language.

The dawn of Structured Programming

- Software Engineering was invented in 1968, and was an almost immediate complete bloody disaster.
- In about 1974, Hoare and Dijkstra tried to repair the damage, by inventing Structured Programming.
- Only use program constructs which we understand and you can reason about: assignment (:=), sequence (;), condition (if-then-else-fi), iteration (while-do-od) and procedure call.
- Do *not* use goto; do *not* commit aliasing.
- Implicitly, use high-level language.
- It was vehemently opposed, but slowly achieved total victory. Nobody now writes in anything else. (Except in god-awful Java, and unspeakable C++, and in C#, and ...)

The dawn of Structured Programming

- Software Engineering was invented in 1968, and was an almost immediate complete bloody disaster.
- In about 1974, Hoare and Dijkstra tried to repair the damage, by inventing Structured Programming.
- Only use program constructs which we understand and you can reason about: assignment (:=), sequence (;), condition (if-then-else-fi), iteration (while-do-od) and procedure call.
- Do *not* use goto; do *not* commit aliasing.
- Implicitly, use high-level language.
- It was vehemently opposed, but slowly achieved total victory. Nobody now writes in anything else. (Except in god-awful Java, and unspeakable C++, and in C#, and ...)
- Structured Programming was a Bloody Good Idea, in stark contrast to Software Engineering (UML, anybody?).

After Structured Programming

- Programs got bigger. Of course!

After Structured Programming

- Programs got bigger. Of course!
- (Except for the concurrent ones, of course.)

The dawn of types

The dawn of types

- Types came to us via two routes:

The dawn of types

- Types came to us via two routes:
  - from Russell (type hierarchy, a solution to the paradox with which he kneecapped poor Frege);

The dawn of types

- Types came to us via two routes:
  - from Russell (type hierarchy, a solution to the paradox with which he kneecapped poor Frege);
  - from FORTRAN via Algol 60: INT means use the integer accumulator; REAL means use that floating-point thingy instead.

The dawn of types

- Types came to us via two routes:
  - from Russell (type hierarchy, a solution to the paradox with which he kneecapped poor Frege);
  - from FORTRAN via Algol 60: INT means use the integer accumulator; REAL means use that floating-point thingy instead.
- About 1972, in Burstall's Hope, and a bit later, in Milner's ML, the routes converged. Type *inference* became possible.

The dawn of types

- Types came to us via two routes:
  - from Russell (type hierarchy, a solution to the paradox with which he kneecapped poor Frege);
  - from FORTRAN via Algol 60: INT means use the integer accumulator; REAL means use that floating-point thingy instead.
- About 1972, in Burstall's Hope, and a bit later, in Milner's ML, the routes converged. Type *inference* became possible.
- Hoare and others began to proseltyse types as a means of avoiding mistakes. Another Bloody Good Idea.

The dawn of types

- Types came to us via two routes:
  - from Russell (type hierarchy, a solution to the paradox with which he kneecapped poor Frege);
  - from FORTRAN via Algol 60: INT means use the integer accumulator; REAL means use that floating-point thingy instead.
- About 1972, in Burstall's Hope, and a bit later, in Milner's ML, the routes converged. Type *inference* became possible.
- Hoare and others began to proseltyse types as a means of avoiding mistakes. Another Bloody Good Idea.
- Types won when they reached C, because they helped people to program more safely with C pointers and procedure calls (though C syntax did its best to stop them).

The dawn of types

- Types came to us via two routes:
  - from Russell (type hierarchy, a solution to the paradox with which he kneecapped poor Frege);
  - from FORTRAN via Algol 60: INT means use the integer accumulator; REAL means use that floating-point thingy instead.
- About 1972, in Burstall's Hope, and a bit later, in Milner's ML, the routes converged. Type *inference* became possible.
- Hoare and others began to proseltyse types as a means of avoiding mistakes. Another Bloody Good Idea.
- Types won when they reached C, because they helped people to program more safely with C pointers and procedure calls (though C syntax did its best to stop them).
- Bertrand Meyer (Eifell) thinks that OOP is based on the idea of types. Would that it were so! (The road to Hell is paved with good intentions.)

An after-lunch fiasco

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),
- and then to prove that the program corresponded to its specification.

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),
- and then to prove that the program corresponded to its specification.
- It was a fiasco. (Fiasco: sounds like fiesta. Fun, but still a fiasco.)

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),
- and then to prove that the program corresponded to its specification.
- It was a fiasco. (Fiasco: sounds like fiesta. Fun, but still a fiasco.)

- The specifications were to be more precise than the English which spawned them.

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),
- and then to prove that the program corresponded to its specification.
- It was a fiasco. (Fiasco: sounds like fiesta. Fun, but still a fiasco.)

- The specifications were to be more precise than the English which spawned them.
- They were more precise but also more *obscure*, and very very very very *very* hard to think up.

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),
- and then to prove that the program corresponded to its specification.
- It was a fiasco. (Fiasco: sounds like fiesta. Fun, but still a fiasco.)

- The specifications were to be more precise than the English which spawned them.
- They were more precise but also more *obscure*, and very very very very *very* hard to think up.
- Programs which used arrays were hard to deal with.

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),
- and then to prove that the program corresponded to its specification.
- It was a fiasco. (Fiasco: sounds like fiesta. Fun, but still a fiasco.)

- The specifications were to be more precise than the English which spawned them.
- They were more precise but also more *obscure*, and very very very very *very* hard to think up.
- Programs which used arrays were hard to deal with.
- Programs which involved loops were harder still.

An after-lunch fiasco

- In 1977, Hoare began a software engineering project.
- The idea was write *small* specifications in classical logic of *large* programs in a high-level language (not C),
- and then to prove that the program corresponded to its specification.
- It was a fiasco. (Fiasco: sounds like fiesta. Fun, but still a fiasco.)

- The specifications were to be more precise than the English which spawned them.
- They were more precise but also more *obscure*, and very very very very *very* hard to think up.
- Programs which used arrays were hard to deal with.
- Programs which involved loops were harder still.
- Pointers were *right out*, and probably anathema.

Don’t go swimming straight after lunch

- Or, when in a hole, stop digging. They didn’t.

Don't go swimming straight after lunch

- Or, when in a hole, stop digging. They didn't.
- They invented refinement, which is *even harder* than verification.

Don’t go swimming straight after lunch

- Or, when in a hole, stop digging. They didn’t.
- They invented refinement, which is *even harder* than verification.

- They invented Z (Abrial & Sufrin), in an attempt to allow bozos to specify things they couldn’t prove.

Don’t go swimming straight after lunch

- Or, when in a hole, stop digging. They didn’t.
- They invented refinement, which is *even harder* than verification.

- They invented Z (Abrial & Sufrin), in an attempt to allow bozos to specify things they couldn’t prove.
- Abrial ran away from Z, and made a tool called B – which works – but there are only two people in the world who can use it.

Don't go swimming straight after lunch

- Or, when in a hole, stop digging. They didn't.
- They invented refinement, which is *even harder* than verification.

- They invented Z (Abrial & Sufrin), in an attempt to allow bozos to specify things they couldn't prove.
- Abrial ran away from Z, and made a tool called B – which works – but there are only two people in the world who can use it.
- One of them is Abrial, and he used it to make the safety software for the Paris Métro Ligne 14 (driverless: have a go!).

Don’t go swimming straight after lunch

- Or, when in a hole, stop digging. They didn’t.
- They invented refinement, which is *even harder* than verification.

- They invented Z (Abrial & Sufrin), in an attempt to allow bozos to specify things they couldn’t prove.
- Abrial ran away from Z, and made a tool called B – which works – but there are only two people in the world who can use it.
- One of them is Abrial, and he used it to make the safety software for the Paris Métro Ligne 14 (driverless: have a go!).
- So: not a complete bloody disaster, but really quite a train wreck, and highly entertaining if you weren’t on the train. I was.

Don't go swimming straight after lunch

- Or, when in a hole, stop digging. They didn't.
- They invented refinement, which is *even harder* than verification.

- They invented Z (Abrial & Sufrin), in an attempt to allow bozos to specify things they couldn't prove.
- Abrial ran away from Z, and made a tool called B – which works – but there are only two people in the world who can use it.
- One of them is Abrial, and he used it to make the safety software for the Paris Métro Ligne 14 (driverless: have a go!).
- So: not a complete bloody disaster, but really quite a train wreck, and highly entertaining if you weren't on the train. I was.
- That train wreck haunts us still: half of you are here to laugh at my idiocy in still trying to ride the rails.

This evening

This evening

- All those suns – high-level languages, structured programming, types – have passed their zenith and sunk below the horizon. It is now dark.

This evening

- All those suns – high-level languages, structured programming, types – have passed their zenith and sunk below the horizon. It is now dark.
- The concurrency-sun never even dawned. The formal proof hoo-hah was all hoo-hah.

This evening

- All those suns – high-level languages, structured programming, types – have passed their zenith and sunk below the horizon. It is now dark.
- The concurrency-sun never even dawned. The formal proof hoo-hah was all hoo-hah.
- In mid-afternoon, OOP started a forest fire, and nobody could see anything in the smoke.

This evening

- All those suns – high-level languages, structured programming, types – have passed their zenith and sunk below the horizon. It is now dark.
- The concurrency-sun never even dawned. The formal proof hoo-hah was all hoo-hah.
- In mid-afternoon, OOP started a forest fire, and nobody could see anything in the smoke.
- In late afternoon, Java started burning. The smoke of its god-awful stupid bloody concurrency mechanisms was so thick that most people thought the types-sun had already gone down, and nobody saw it set.

This evening

- All those suns – high-level languages, structured programming, types – have passed their zenith and sunk below the horizon. It is now dark.
- The concurrency-sun never even dawned. The formal proof hoo-hah was all hoo-hah.
- In mid-afternoon, OOP started a forest fire, and nobody could see anything in the smoke.
- In late afternoon, Java started burning. The smoke of its god-awful stupid bloody concurrency mechanisms was so thick that most people thought the types-sun had already gone down, and nobody saw it set.
- Here we are around our campfire, telling stories and wondering if the smoke will have gone before the dawn. You’re all pretty demoralised.

The morning

The morning

- I’m here to tell you that the dawn of concurrency is at hand.

The morning

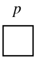
- I'm here to tell you that the dawn of concurrency is at hand.
- *At last* we have a workable formal treatment of concurrency. With its help, we'll be able to see through the Java smoke to the new land around us.

The morning

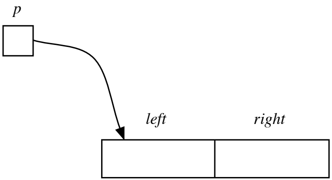
- I’m here to tell you that the dawn of concurrency is at hand.
- *At last* we have a workable formal treatment of concurrency. With its help, we’ll be able to see through the Java smoke to the new land around us.
- This time, the hoo-hah is going to work for real.

How to implement a binary tree

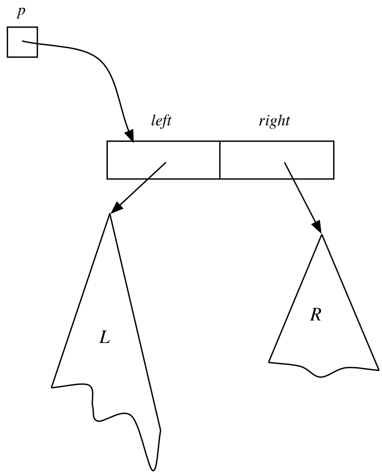
How to implement a binary tree



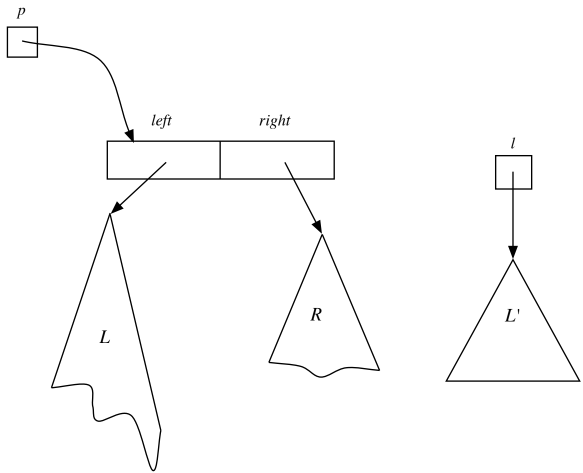
How to implement a binary tree



How to implement a binary tree

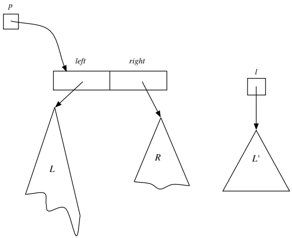


How to implement a binary tree

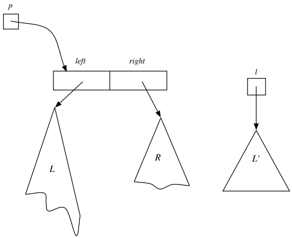


... and an alternative left subtree.

How to replace L with L'?

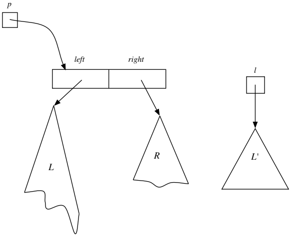


How to replace L with L'?



What could be easier?

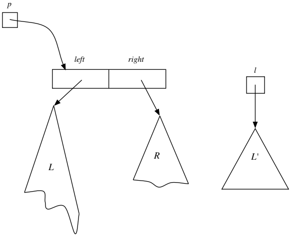
How to replace L with L'?



What could be easier?

$$temp := p.left;$$
$$p.left := l;$$
$$\text{disposetree } temp$$

How to replace L with L'?



What could be easier?

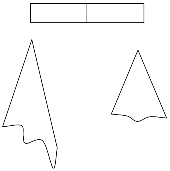
$$temp := p.left;$$
$$p.left := l;$$
$$\text{disposetree } temp$$

(basic first-year undergrad stuff!)

How to describe a tree (Reynolds)

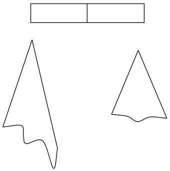
How to describe a tree (Reynolds)

Trees come apart, into three separate sections.



How to describe a tree (Reynolds)

Trees come apart, into three separate sections.



$$
\begin{aligned}
\mathsf{tree}\,\mathsf{Empty}\;p &\;\hat{=}\; p = \mathrm{nil} \wedge \mathbf{emp} \\
\mathsf{tree}\,\mathsf{Node}(\lambda, \rho)\;p &\;\hat{=}\; \exists l, r \cdot (p \mapsto l, r \star \mathrm{tree}\;\lambda\;l \star \mathrm{tree}\;\rho\;r)
\end{aligned}
$$

($p \mapsto l, r$ is a record, $A \star B$ is heap separation)

Separation logic

Separation logic

- $E \mapsto F$ is a single-celled heap with address E and content F.

Separation logic

- $E \mapsto F$ is a single-celled heap with address E and content F.
- $E \mapsto F_0, F_1$ is a two-celled heap; $E \mapsto F_0, F_1, F_2$ is three cells; and so on for four-, five-, ... celled heaps.

Separation logic

- $E \mapsto F$ is a single-celled heap with address E and content F.
- $E \mapsto F_0, F_1$ is a two-celled heap; $E \mapsto F_0, F_1, F_2$ is three cells; and so on for four-, five-, ... celled heaps.
- E and F must be 'pure' expressions that don't mention the heap (don't use $\mapsto$).

Separation logic

- $E \mapsto F$ is a single-celled heap with address E and content F.
- $E \mapsto F_0, F_1$ is a two-celled heap; $E \mapsto F_0, F_1, F_2$ is three cells; and so on for four-, five-, ... celled heaps.
- E and F must be 'pure' expressions that don't mention the heap (don't use $\mapsto$).
- $A \star B$ is separation of heaps; $A \wedge B$, $A \vee B$, $\neg A$, $A \rightarrow B$, $\forall x \cdot P(x)$, $\exists x \cdot P(x)$ are as normal. $A \wedge B$ expresses coincidence of heaps; you don't need to know about $A \mathrel{-\!\!\star} B$.

Separation logic

- $E \mapsto F$ is a single-celled heap with address E and content F.
- $E \mapsto F_0, F_1$ is a two-celled heap; $E \mapsto F_0, F_1, F_2$ is three cells; and so on for four-, five-, ... celled heaps.
- E and F must be 'pure' expressions that don't mention the heap (don't use $\mapsto$).
- $A \star B$ is separation of heaps; $A \wedge B$, $A \vee B$, $\neg A$, $A \rightarrow B$, $\forall x \cdot P(x)$, $\exists x \cdot P(x)$ are as normal. $A \wedge B$ expresses coincidence of heaps; you don't need to know about $A \mathrel{-\!\!\star} B$.
- $E \mapsto F_0, F_1$ is just shorthand for $E \mapsto F_0 \star E + 1 \mapsto F_1$.

A modified Hoare logic

A modified Hoare logic

- $\{Q\}\ C\ \{R\}$ is a resourced and partial correctness assertion. C will not go wrong (exceed its allocated resources) if started with resource Q, and will, if it terminates, deliver resource R.

A modified Hoare logic

- $\{Q\}\, C\, \{R\}$ is a resourced and partial correctness assertion. C will not go wrong (exceed its allocated resources) if started with resource Q, and will, if it terminates, deliver resource R.
- The ‘small axioms’ of assignment are

$$
\begin{array}{ll}
\{\mathbf{emp}\}\, x := \mathrm{new}()\, \{x \mapsto \_\} & \\
\{E \mapsto \_\}\ \mathrm{dispose}\, E\, \{\mathbf{emp}\} & \\
\{R[E/x]\}\, x := E\, \{R\} & (\text{the Hoare axiom}) \\
\{E \mapsto F\}\, x := [E]\, \{x = F \wedge E \mapsto F\} & (x \text{ not free in } E, F) \\
\{E \mapsto \_\}\ [E] := F\, \{E \mapsto F\} &
\end{array}
$$

Three inference rules

Three inference rules

- The frame rule: $\dfrac{\{Q\}\, C\, \{R\}}{\{P \star Q\}\, C\, \{P \star R\}}$ (modifies $C \bigcap$ free $P = \{\}$)

Three inference rules

- The frame rule: $\dfrac{\{Q\}\, C\, \{R\}}{\{P \star Q\}\, C\, \{P \star R\}}$ $(\text{modifies } C \bigcap \text{free } P = \{\})$

- The concurrency rule (has horrid side-condition):

$$\frac{\{Q_1\}\, C_1\, \{R_1\} \quad \{Q_2\}\, C_2\, \{R_2\} \quad \ldots \quad \{Q_n\}\, C_n\, \{R_n\}}{\{Q_1 \star Q_2 \star \cdots \star Q_n\}\, C_1 \mid\mid C_2 \mid\mid \cdots \mid\mid C_n\, \{R_1 \star R_2 \star \cdots \star R_n\}}$$

Three inference rules

- The frame rule: $\dfrac{\{Q\}\ C\ \{R\}}{\{P \star Q\}\ C\ \{P \star R\}}$ (modifies $C \bigcap$ free $P = \{\}$)

- The concurrency rule (has horrid side-condition):

$$\frac{\{Q_1\}\ C_1\ \{R_1\} \qquad \{Q_2\}\ C_2\ \{R_2\} \qquad \ldots \qquad \{Q_n\}\ C_n\ \{R_n\}}{\{Q_1 \star Q_2 \star \cdots \star Q_n\}\ C_1 \parallel C_2 \parallel \cdots \parallel C_n\ \{R_1 \star R_2 \star \cdots \star R_n\}}$$

- The CCR rule (has *atrocious* side condition):

$$\frac{\{(Q \star I_b) \wedge G\}\ C\ \{R \star I_b\}}{\{Q\}\ \text{with } b \text{ when } G \text{ do } C \text{ od}\ \{R\}}$$

Recent simplifications (not explained here)

Recent simplifications (not explained here)

- Permissions (fractions of $\mapsto$, counts of $\rightarrowtail$) to allow sharing of heap;

Recent simplifications (not explained here)

- Permissions (fractions of $\mapsto$, counts of $\rightarrowtail$) to allow sharing of heap;
- Variable permissions, to allow variables to be resource;

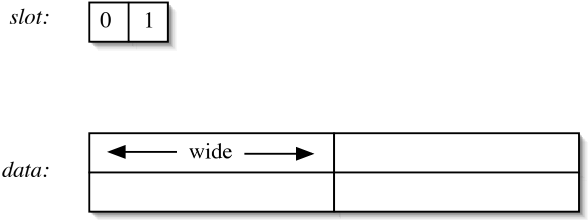
Recent simplifications (not explained here)

- Permissions (fractions of $\mapsto$, counts of $\rightarrowtail$) to allow sharing of heap;
- Variable permissions, to allow variables to be resource;
- Trivial side conditions;

Recent simplifications (not explained here)

- Permissions (fractions of $\mapsto$, counts of $\rightarrowtail$) to allow sharing of heap;
- Variable permissions, to allow variables to be resource;
- Trivial side conditions;
- No side conditions at all (very new, this!).

Data structures: a bit array and a wide data array



Nine lines are now ten, with added auxiliary proof-variables

write: with $bundle$ when true do $pair :=$ not$(reading)$; $wuse := pair$ od;
$index :=$ not$(slot[pair])$;
$data[pair, index] := item$;
with $bundle$ when true do $slot[pair] := index$; $wuse := -1$ od;
with $bundle$ when true do $latest := pair$ od

read: with $bundle$ when true do $pair := latest$ od;
with $bundle$ when true do $reading := pair$ od;
with $bundle$ when true do $index := slot[pair]$; $ruse := index$ od;
$read := data[pair, index]$;
with $bundle$ when true do $ruse := -1$ od

What the writer owns

(A point of notation: I've used a special form of $\mapsto$ to describe a heap, just to make the slides easy to read.

For example, $data[pair, index] \mapsto \_$ replaces $data + 2 \star pair + index \mapsto \_$.

There is no change in meaning.)

What the writer owns

(A point of notation: I've used a special form of $\mapsto$ to describe a heap, just to make the slides easy to read.

For example, $data[pair, index] \mapsto \_$ replaces $data + 2 \star pair + index \mapsto \_$.

There is no change in meaning.)

$$latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index$$

$$\models \left(\begin{array}{l} slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \star \\ \text{if } wuse \geq 0 \text{ then } data[pair, index] \mapsto \_ \text{ else } \mathbf{emp} \text{ fi} \end{array} \right)$$

What the reader owns

$$reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index$$
$$\models \text{if } ruse \geq 0 \text{ then } data[pair, index] \mapsto \_ \text{ else } \mathbf{emp} \text{ fi}$$

The bundle owns the rest

$$latest_{0.5}, reading_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, ruse_{0.5}$$

$$\models \exists s \cdot \left(\begin{array}{l} slot[0] \underset{0.5}{\longmapsto} s(0) \star slot[1] \underset{0.5}{\longmapsto} s(1) \star \\ \text{if } wuse \geq 0 \wedge ruse \geq 0 \text{ then} \\ \quad data[reading, \text{not}(ruse)] \mapsto \_ \star data[wuse, s(wuse)] \mapsto \_ \\ \text{elsf } wuse \geq 0 \text{ then} \\ \quad data[wuse, s(wuse)] \mapsto \_ \star \\ \quad data[\text{not}(wuse), s(\text{not}(wuse))] \mapsto \_ \star data[\text{not}(wuse), \text{not}(s(\text{not}(wuse)))] \mapsto \_ \\ \text{elsf } ruse \geq 0 \text{ then} \\ \quad data[reading, \text{not}(ruse)] \mapsto \_ \star \\ \quad data[\text{not}(reading), s(\text{not}(reading))] \mapsto \_ \star data[\text{not}(reading), \text{not}(s(\text{not}(reading)))] \mapsto \_) \\ \text{else} \\ \quad data \mapsto \_, \_, \_, \_ \\ \text{fi} \end{array} \right)$$

The writer

$$\left\{\; latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \vDash wuse = -1 \wedge slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \;\right\}$$

with $bundle$ when true do $pair :=$ not$(reading)$; $wuse := pair$ od;

$index :=$ not$(slot[pair])$;

$data[pair, index] := item$;

with $bundle$ when true do $slot[pair] := index$; $wuse := -1$ od;

with $bundle$ when true do $latest := pair$ od

$$\left\{\; latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \vDash wuse = -1 \wedge slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \;\right\}$$

The writer

$$\left\{ \mathit{latest}_{0.5}, \mathit{slot}_{0.5}, \mathit{data}_{0.33}, \mathit{wuse}_{0.5}, \mathit{pair}, \mathit{index} \vDash \mathit{wuse} = -1 \wedge \mathit{slot}[0] \xmapsto[0.5]{} \_ \star \mathit{slot}[1] \xmapsto[0.5]{} \_ \right\}$$

with bundle when true do $\mathit{pair} := \mathrm{not}(\mathit{reading})$; $\mathit{wuse} := \mathit{pair}$ od;

$$\left\{ \begin{array}{l} \mathit{latest}_{0.5}, \mathit{slot}_{0.5}, \mathit{data}_{0.33}, \mathit{wuse}_{0.5}, \mathit{pair}, \mathit{index} \\ \quad \vDash \mathit{wuse} = \mathit{pair} \wedge \exists i \cdot \left(\begin{array}{l} \mathit{slot}[\mathit{pair}] \xmapsto[0.5]{} i \star \mathit{slot}[\mathrm{not}(\mathit{pair})] \xmapsto[0.5]{} \_ \star \\ \mathit{data}[\mathit{pair}, \mathrm{not}(i)] \mapsto \_ \end{array} \right) \end{array} \right\}$$

$\mathit{index} := \mathrm{not}(\mathit{slot}[\mathit{pair}])$;

$\mathit{data}[\mathit{pair}, \mathit{index}] := \mathit{item}$;

with bundle when true do $\mathit{slot}[\mathit{pair}] := \mathit{index}$; $\mathit{wuse} := -1$ od;

with bundle when true do $\mathit{latest} := \mathit{pair}$ od

$$\left\{ \mathit{latest}_{0.5}, \mathit{slot}_{0.5}, \mathit{data}_{0.33}, \mathit{wuse}_{0.5}, \mathit{pair}, \mathit{index} \vDash \mathit{wuse} = -1 \wedge \mathit{slot}[0] \xmapsto[0.5]{} \_ \star \mathit{slot}[1] \xmapsto[0.5]{} \_ \right\}$$

The writer

$$\left\{ latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \land slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \right\}$$

with *bundle* when true do *pair* := not(*reading*); *wuse* := *pair* od;

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \land \exists i \cdot \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} i \star slot[\mathrm{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, \mathrm{not}(i)] \mapsto \_ \end{array} \right) \end{array} \right\}$$

index := not(*slot*[*pair*]);

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \land \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} \mathrm{not}(index) \star slot[\mathrm{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, index] \mapsto \_ \end{array} \right) \end{array} \right\}$$

data[*pair*, *index*] := *item*;

with *bundle* when true do *slot*[*pair*] := *index*; *wuse* := −1 od;

with *bundle* when true do *latest* := *pair* od

$$\left\{ latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \land slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \right\}$$

The writer

$$\left\{\ latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \land slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_\ \right\}$$

with $bundle$ when true do $pair :=$ not$(reading)$; $wuse := pair$ od;

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \land \exists i \cdot \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} i \star slot[\text{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, \text{not}(i)] \mapsto \_ \end{array} \right) \end{array} \right\}$$

$index :=$ not$(slot[pair])$;

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \land \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} \text{not}(index) \star slot[\text{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, index] \mapsto \_ \end{array} \right) \end{array} \right\}$$

$data[pair, index] := item$;

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \land \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} \text{not}(index) \star slot[\text{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, index] \mapsto item \end{array} \right) \end{array} \right\}$$

with $bundle$ when true do $slot[pair] := index$; $wuse := -1$ od;

with $bundle$ when true do $latest := pair$ od

$$\left\{\ latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \land slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_\ \right\}$$

The writer

$$\left\{\; latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \wedge slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \;\right\}$$

with $bundle$ when true do $pair := \mathrm{not}(reading)$; $wuse := pair$ od;

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \wedge \exists i \cdot \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} i \star slot[\mathrm{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, \mathrm{not}(i)] \mapsto \_ \end{array}\right) \end{array}\right\}$$

$index := \mathrm{not}(slot[pair])$;

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \wedge \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} \mathrm{not}(index) \star slot[\mathrm{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, index] \mapsto \_ \end{array}\right) \end{array}\right\}$$

$data[pair, index] := item$;

$$\left\{\begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \wedge \left(\begin{array}{l} slot[pair] \underset{0.5}{\longmapsto} \mathrm{not}(index) \star slot[\mathrm{not}(pair)] \underset{0.5}{\longmapsto} \_ \star \\ data[pair, index] \mapsto item \end{array}\right) \end{array}\right\}$$

with $bundle$ when true do $slot[pair] := index$; $wuse := -1$ od;

$$\left\{\; latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \wedge slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \;\right\}$$

with $bundle$ when true do $latest := pair$ od

$$\left\{\; latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \wedge slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \;\right\}$$

Details of the first writer step

$$\left\{\ latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \vDash wuse = -1 \wedge slot[0] \xmapsto[0.5]{} \_ \star slot[1] \xmapsto[0.5]{} \_\ \right\}$$

with $bundle$ when true do

$pair :=$ not$(reading)$;

$wuse := pair$

od;

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \vDash wuse = pair \wedge \exists i \cdot \left(slot[pair] \xmapsto[0.5]{} i \star slot[\text{not}(pair)] \xmapsto[0.5]{} \_ \star data[pair, \text{not}(i)] \mapsto \_ \right) \end{array} \right\}$$

Details of the first writer step

$$\left\{\; latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \wedge slot[0] \underset{0.5}{\longmapsto} \_ \star slot[1] \underset{0.5}{\longmapsto} \_ \;\right\}$$

with *bundle* when true do

$$\left\{ \begin{array}{l} latest, reading_{0.5}, slot, data_{0.66}, wuse, pair, index \\ \models \exists s \cdot \left(\begin{array}{l} wuse = -1 \wedge slot \mapsto s(0), s(1) \star \\ data[\mathsf{not}(reading), s(\mathsf{not}(reading))] \mapsto \_ \star data[\mathsf{not}(reading), \mathsf{not}(s(\mathsf{not}(reading)))] \mapsto \_ \star \\ \text{ if } ruse \geq 0 \;\; \text{then } data[reading, \mathsf{not}(ruse)] \mapsto \_ \\ \qquad\qquad\quad\;\; \text{else } data[reading, s(reading)] \mapsto \_ \star data[reading, \mathsf{not}(s(reading))] \mapsto \_ \\ \text{ fi} \end{array} \right) \end{array} \right\}$$

$pair := \mathsf{not}(reading);$

$wuse := pair$

od;

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \quad \models wuse = pair \wedge \exists i \cdot \left(slot[pair] \underset{0.5}{\longmapsto} i \star slot[\mathsf{not}(pair)] \underset{0.5}{\longmapsto} \_ \star data[pair, \mathsf{not}(i)] \mapsto \_ \right) \end{array} \right\}$$

Details of the first writer step

$$\left\{ latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \models wuse = -1 \land slot[0] \xmapsto[0.5]{} \_ \star slot[1] \xmapsto[0.5]{} \_ \right\}$$

with *bundle* when true do

$$\left\{ \begin{array}{l} latest, reading_{0.5}, slot, data_{0.66}, wuse, pair, index \\ \models \exists s \cdot \left(\begin{array}{l} wuse = -1 \land slot \mapsto s(0), s(1) \star \\ data[\mathsf{not}(reading), s(\mathsf{not}(reading))] \mapsto \_ \star data[\mathsf{not}(reading), \mathsf{not}(s(\mathsf{not}(reading)))] \mapsto \_ \star \\ \text{if } ruse \geq 0 \text{ then } data[reading, \mathsf{not}(ruse)] \mapsto \_ \\ \qquad\qquad\quad \text{else } data[reading, s(reading)] \mapsto \_ \star data[reading, \mathsf{not}(s(reading))] \mapsto \_ \\ \text{fi} \end{array} \right) \end{array} \right\}$$

$pair := \mathsf{not}(reading);$

$$\left\{ \begin{array}{l} latest, reading_{0.5}, slot, data_{0.66}, wuse, pair, index \\ \models \exists s \cdot \left(\begin{array}{l} wuse = -1 \land pair = \mathsf{not}(reading) \land slot \mapsto s(0), s(1) \star \\ data[\mathsf{not}(reading), s(\mathsf{not}(reading))] \mapsto \_ \star data[\mathsf{not}(reading), \mathsf{not}(s(\mathsf{not}(reading)))] \mapsto \_ \star \\ \text{if } ruse \geq 0 \text{ then } data[reading, \mathsf{not}(ruse)] \mapsto \_ \\ \qquad\qquad\quad \text{else } data[reading, s(reading)] \mapsto \_ \star data[reading, \mathsf{not}(s(reading))] \mapsto \_ \\ \text{fi} \end{array} \right) \end{array} \right\}$$

$wuse := pair$

od;

$$\left\{ \begin{array}{l} latest_{0.5}, slot_{0.5}, data_{0.33}, wuse_{0.5}, pair, index \\ \models wuse = pair \land \exists i \cdot \left(slot[pair] \xmapsto[0.5]{} i \star slot[\mathsf{not}(pair)] \xmapsto[0.5]{} \_ \star data[pair, \mathsf{not}(i)] \mapsto \_ \right) \end{array} \right\}$$

Details of the first writer step

$$\left\{ \mathit{latest}_{0.5}, \mathit{slot}_{0.5}, \mathit{data}_{0.33}, \mathit{wuse}_{0.5}, \mathit{pair}, \mathit{index} \models \mathit{wuse} = -1 \wedge \mathit{slot}[0] \underset{0.5}{\longmapsto} \_ \star \mathit{slot}[1] \underset{0.5}{\longmapsto} \_ \right\}$$

with *bundle* when true do

$$\left\{ \begin{array}{l} \mathit{latest}, \mathit{reading}_{0.5}, \mathit{slot}, \mathit{data}_{0.66}, \mathit{wuse}, \mathit{pair}, \mathit{index} \\ \models \exists s \cdot \left(\begin{array}{l} \mathit{wuse} = -1 \wedge \mathit{slot} \mapsto s(0), s(1) \star \\ \mathit{data}[\mathrm{not}(\mathit{reading}), s(\mathrm{not}(\mathit{reading}))] \mapsto \_ \star \mathit{data}[\mathrm{not}(\mathit{reading}), \mathrm{not}(s(\mathrm{not}(\mathit{reading})))] \mapsto \_ \star \\ \text{if } \mathit{ruse} \geq 0 \text{ then } \mathit{data}[\mathit{reading}, \mathrm{not}(\mathit{ruse})] \mapsto \_ \\ \qquad\qquad \text{else } \mathit{data}[\mathit{reading}, s(\mathit{reading})] \mapsto \_ \star \mathit{data}[\mathit{reading}, \mathrm{not}(s(\mathit{reading}))] \mapsto \_ \\ \text{fi} \end{array} \right) \end{array} \right\}$$

pair := not(*reading*);

$$\left\{ \begin{array}{l} \mathit{latest}, \mathit{reading}_{0.5}, \mathit{slot}, \mathit{data}_{0.66}, \mathit{wuse}, \mathit{pair}, \mathit{index} \\ \models \exists s \cdot \left(\begin{array}{l} \mathit{wuse} = -1 \wedge \mathit{pair} = \mathrm{not}(\mathit{reading}) \wedge \mathit{slot} \mapsto s(0), s(1) \star \\ \mathit{data}[\mathrm{not}(\mathit{reading}), s(\mathrm{not}(\mathit{reading}))] \mapsto \_ \star \mathit{data}[\mathrm{not}(\mathit{reading}), \mathrm{not}(s(\mathrm{not}(\mathit{reading})))] \mapsto \_ \star \\ \text{if } \mathit{ruse} \geq 0 \text{ then } \mathit{data}[\mathit{reading}, \mathrm{not}(\mathit{ruse})] \mapsto \_ \\ \qquad\qquad \text{else } \mathit{data}[\mathit{reading}, s(\mathit{reading})] \mapsto \_ \star \mathit{data}[\mathit{reading}, \mathrm{not}(s(\mathit{reading}))] \mapsto \_ \\ \text{fi} \end{array} \right) \end{array} \right\}$$

wuse := *pair*

$$\left\{ \begin{array}{l} \mathit{latest}, \mathit{reading}_{0.5}, \mathit{slot}, \mathit{data}_{0.66}, \mathit{wuse}, \mathit{pair}, \mathit{index} \\ \models \exists s \cdot \left(\begin{array}{l} \mathit{wuse} = \mathit{pair} \wedge \mathit{pair} = \mathrm{not}(\mathit{reading}) \wedge \mathit{slot} \mapsto s(0), s(1) \star \\ \mathit{data}[\mathrm{not}(\mathit{reading}), s(\mathrm{not}(\mathit{reading}))] \mapsto \_ \star \mathit{data}[\mathrm{not}(\mathit{reading}), \mathrm{not}(s(\mathrm{not}(\mathit{reading})))] \mapsto \_ \star \\ \text{if } \mathit{ruse} \geq 0 \text{ then } \mathit{data}[\mathit{reading}, \mathrm{not}(\mathit{ruse})] \mapsto \_ \\ \qquad\qquad \text{else } \mathit{data}[\mathit{reading}, s(\mathit{reading})] \mapsto \_ \star \mathit{data}[\mathit{reading}, \mathrm{not}(s(\mathit{reading}))] \mapsto \_ \\ \text{fi} \end{array} \right) \end{array} \right\}$$

od;

$$\left\{ \begin{array}{l} \mathit{latest}_{0.5}, \mathit{slot}_{0.5}, \mathit{data}_{0.33}, \mathit{wuse}_{0.5}, \mathit{pair}, \mathit{index} \\ \models \mathit{wuse} = \mathit{pair} \wedge \exists i \cdot \left(\mathit{slot}[\mathit{pair}] \underset{0.5}{\longmapsto} i \star \mathit{slot}[\mathrm{not}(\mathit{pair})] \underset{0.5}{\longmapsto} \_ \star \mathit{data}[\mathit{pair}, \mathrm{not}(i)] \mapsto \_ \right) \end{array} \right\}$$

The reader is even easier than the writer!

$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1\ \}$

with *bundle* when true do *pair* := *latest* od;

with *bundle* when true do *reading* := *pair* od;

with *bundle* when true do *index* := *slot*[*pair*]; *ruse* := *index* od;

$read := data[pair, index];$

with *bundle* when true do *ruse* := -1 od

$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1\ \}$

The reader is even easier than the writer!

$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse = -1\ \}$
with *bundle* when true do *pair* := *latest* od;
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse = -1\ \}$
with *bundle* when true do *reading* := *pair* od;

with *bundle* when true do *index* := *slot*[*pair*]; *ruse* := *index* od;

read := *data*[*pair*, *index*];

with *bundle* when true do *ruse* := -1 od
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse = -1\ \}$

The reader is even easier than the writer!

$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1\ \}$
with *bundle* when true do $pair := latest$ od;
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1\ \}$
with *bundle* when true do $reading := pair$ od;
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1 \wedge reading = pair\ \}$
with *bundle* when true do $index := slot[pair]$; $ruse := index$ od;

$read := data[pair, index]$;

with *bundle* when true do $ruse := -1$ od
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1\ \}$

The reader is even easier than the writer!

$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse = -1\ \}$
with $bundle$ when true do $pair := latest$ od;
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse = -1\ \}$
with $bundle$ when true do $reading := pair$ od;
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse = -1 \wedge reading = pair\ \}$
with $bundle$ when true do $index := slot[pair]; ruse := index$ od;
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse \geq 0 \wedge reading = pair \wedge data[pair, index] \mapsto \_\ \}$
$read := data[pair, index]$;

with $bundle$ when true do $ruse := -1$ od
$\{\ reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \models ruse = -1\ \}$

The reader is even easier than the writer!

$\{\; reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1 \;\}$

with $bundle$ when true do $pair := latest$ od;

$\{\; reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1 \;\}$

with $bundle$ when true do $reading := pair$ od;

$\{\; reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1 \wedge reading = pair \;\}$

with $bundle$ when true do $index := slot[pair]$; $ruse := index$ od;

$\{\; reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse \geq 0 \wedge reading = pair \wedge data[pair, index] \mapsto \_ \;\}$

$read := data[pair, index]$;

$$\left\{ \begin{array}{l} reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \\ \quad \vDash ruse \geq 0 \wedge reading = pair \wedge \exists i \cdot data[pair, index] \mapsto i \wedge read = i \end{array} \right\}$$

with $bundle$ when true do $ruse := -1$ od

$\{\; reading_{0.5}, ruse_{0.5}, data_{0.33}, pair, index \vDash ruse = -1 \;\}$

The rest of the reader is too easy to bother with

$$\text{with } bundle \text{ when true do } index := slot[pair]; \color{green}{ruse} := index$$

(in the reader) is very very *very* similar to

$$\text{with } bundle \text{ when true do } pair := \mathsf{not}(reading); \color{green}{wuse} := pair \text{ od}$$

(which I just showed you in detail from the writer),

so you don’t need to see it.

The rest of the reader is too easy to bother with

with $bundle$ when true do $index := slot[pair];\ ruse := index$

(in the reader) is very very *very* similar to

with $bundle$ when true do $pair := \mathrm{not}(reading);\ wuse := pair$ od

(which I just showed you in detail from the writer),

so you don't need to see it.

And the rest of the reader is trivial.

Nice Proofs™

Nice Proofs^{TM}

- Are proofs which you can read, understand and believe.

Nice Proofs™

- Are proofs which you can read, understand and believe.
- Proofs which don't fit on one slide are unbelievable.

Nice Proofs^{TM}

- Are proofs which you can read, understand and believe.
- Proofs which don't fit on one slide are unbelievable.
- Previous proofs of Simpson's 4-slot are Henderson (rely-guarantee, about 20 pages), Burton (refinement of atomicity, about 25 pages) and Burton's thesis (somehow, about 99 pages).

Nice Proofs™

- Are proofs which you can read, understand and believe.
- Proofs which don't fit on one slide are unbelievable.
- Previous proofs of Simpson's 4-slot are Henderson (rely-guarantee, about 20 pages), Burton (refinement of atomicity, about 25 pages) and Burton's thesis (somehow, about 99 pages).
- I ain't reading no 99-page proof.

Nice Proofs™

- Are proofs which you can read, understand and believe.
- Proofs which don’t fit on one slide are unbelievable.
- Previous proofs of Simpson’s 4-slot are Henderson (rely-guarantee, about 20 pages), Burton (refinement of atomicity, about 25 pages) and Burton’s thesis (somehow, about 99 pages).
- I ain’t reading no 99-page proof.
- My proofs fit on a slide with a bit of scaleboxing. You can read them. Given a day or so, you can understand them.

Nice Proofs™

- Are proofs which you can read, understand and believe.
- Proofs which don’t fit on one slide are unbelievable.
- Previous proofs of Simpson’s 4-slot are Henderson (rely-guarantee, about 20 pages), Burton (refinement of atomicity, about 25 pages) and Burton’s thesis (somehow, about 99 pages).
- I ain’t reading no 99-page proof.
- My proofs fit on a slide with a bit of scaleboxing. You can read them. Given a day or so, you can understand them.
- For the *very first time* we have nice proofs of a nine-line algorithm.

Nice Proofs™

- Are proofs which you can read, understand and believe.
- Proofs which don't fit on one slide are unbelievable.
- Previous proofs of Simpson's 4-slot are Henderson (rely-guarantee, about 20 pages), Burton (refinement of atomicity, about 25 pages) and Burton's thesis (somehow, about 99 pages).
- I ain't reading no 99-page proof.
- My proofs fit on a slide with a bit of scaleboxing. You can read them. Given a day or so, you can understand them.
- For the *very first time* we have nice proofs of a nine-line algorithm.
- I hope it has been worth the wait.

Summary

Summary

- O'Hearn and Reynolds invented separation logic to deal with lists and trees.

Summary

- O'Hearn and Reynolds invented separation logic to deal with lists and trees.
- I made it deal with graphs, by brute force and ignorance.

Summary

- O’Hearn and Reynolds invented separation logic to deal with lists and trees.
- I made it deal with graphs, by brute force and ignorance.
- O’Hearn showed me how to pass pointers in messages.

Summary

- O'Hearn and Reynolds invented separation logic to deal with lists and trees.
- I made it deal with graphs, by brute force and ignorance.
- O'Hearn showed me how to pass pointers in messages.
- By brute force, I made them all take notice of permissions.

Summary

- O'Hearn and Reynolds invented separation logic to deal with lists and trees.
- I made it deal with graphs, by brute force and ignorance.
- O'Hearn showed me how to pass pointers in messages.
- By brute force, I made them all take notice of permissions.
- Similarly, I made them take notice that variables are resource too.

Summary

- O'Hearn and Reynolds invented separation logic to deal with lists and trees.
- I made it deal with graphs, by brute force and ignorance.
- O'Hearn showed me how to pass pointers in messages.
- By brute force, I made them all take notice of permissions.
- Similarly, I made them take notice that variables are resource too.
- I did some proofs of some hoary old concurrency favourites.

Summary

- O'Hearn and Reynolds invented separation logic to deal with lists and trees.
- I made it deal with graphs, by brute force and ignorance.
- O'Hearn showed me how to pass pointers in messages.
- By brute force, I made them all take notice of permissions.
- Similarly, I made them take notice that variables are resource too.
- I did some proofs of some hoary old concurrency favourites.
- Matthew Parkinson, then Matthew Parkinson and I, did proofs of some old concurrency puzzlers.