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.

“What has formalism ever done for us?” (Mark Woodman, Professor in 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.
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

```plaintext
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 ...)

Structured Programming was a Bloody Good Idea, in stark contrast to Software Engineering (UML, anybody?).
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:
  - 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 proselytise 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.)
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 proselytise 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.)
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 proselytise 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 proselytise 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 proselytise 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

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 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.
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),
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.)
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 *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 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 *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!).

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.
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.
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

![Diagram of a binary tree with a node labeled $p$, and pointers to left and right subtrees.](image)
How to implement a binary tree
How to implement a binary tree

... and an alternative left subtree.
How to replace $L$ with $L'$?

$$p \leftarrow \begin{cases} \text{left} & \text{right} \\ L & R \end{cases}$$

```plaintext
What could be easier?
```

```plaintext
\text{temp} := p.\text{left};
\text{p.left} := l;
\text{dispose tree temp}
```

(basic first-year undergrad stuff!)
How to replace $L$ with $L'$?

What could be easier?
How to replace $L$ with $L'$?

What could be easier?

\[
\begin{align*}
temp & := p.\text{left}; \\
p.\text{left} & := l; \\
\text{disposetree} & \ temp
\end{align*}
\]
How to replace $L$ with $L'$?

What could be easier?

\[
\begin{align*}
temp & := p.left; \\
p.left & := l; \\
\text{disposetree } temp
\end{align*}
\]

(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.

\[
\text{tree Empty } p \quad \triangleq \quad p = \text{nil} \land \text{emp}
\]

\[
\text{tree Node}(\lambda, \rho) \; p \quad \triangleq \quad \exists \; l, r \cdot (p \mapsto l, r \ast \text{tree } \lambda l \ast \text{tree } \rho r)
\]

\[(p \mapsto l, r \text{ is a record, } A \ast B \text{ is heap separation})\]
Separation logic

Separation logic introduces a new way to reason about programs that manipulate memory. It is based on the idea of separating different parts of the heap, allowing us to reason about the state of the memory at different points in a program.

Let's consider a single-celled heap with address $E$ and content $F$. We denote this as $E \mapsto \rightarrow F$. This means that there is an address $E$ that points to a cell containing $F$.

A two-celled heap with addresses $E$, $F_0$, and $F_1$ can be denoted as $E \mapsto \rightarrow F_0 \mapsto \rightarrow F_1$. Similarly, a three-celled heap would have $E \mapsto \rightarrow F_0 \mapsto \rightarrow F_1 \mapsto \rightarrow F_2$.

$E$ and $F$ are 'pure' expressions that do not mention the heap, so we do not need to use $\mapsto \rightarrow$ in these cases.

$A \star B$ is separation of heaps, which means that $A \land B$, $A \lor B$, $\neg A$, $A \rightarrow B$, $\forall x \cdot P(x)$, and $\exists x \cdot P(x)$ are as normal.

$A \land B$ expresses coincidence of heaps. You do not need to know about $A - \star B$.

$E \mapsto \rightarrow F_0 \mapsto \rightarrow F_1$ is shorthand for $E \mapsto \rightarrow F_0 \star E + 1 \mapsto \rightarrow F_1$.
Separation logic

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

- $E \leftrightarrow F$ is a single-celled heap with address $E$ and content $F$.
- $E \leftrightarrow F_0, F_1$ is a two-celled heap; $E \leftrightarrow F_0, F_1, F_2$ 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; $A \land B, A \lor B, \neg A, A \rightarrow B, \forall x \cdot P(x), \exists x \cdot P(x)$ are as normal. $A \land B$ expresses coincidence of heaps; you don’t need to know about $A \rightarrow \star B$. 
Separation logic

- $E \leftrightarrow F$ is a single-celled heap with address $E$ and content $F$.
- $E \leftrightarrow F_0, F_1$ is a two-celled heap; $E \leftrightarrow F_0, F_1, F_2$ 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; $A \land B, A \lor B, \neg A, A \rightarrow B, \forall x \cdot P(x), \exists x \cdot P(x)$ are as normal. $A \land B$ expresses coincidence of heaps; you don’t need to know about $A \rightarrow B$.
- $E \leftrightarrow F_0, F_1$ is just shorthand for $E \leftrightarrow F_0 \star E + 1 \leftrightarrow F_1$. 
A modified Hoare logic

The 'small axioms' of assignment are

\[
\{\text{emp}\} \ x := \ \text{new}() \ \Rightarrow \ \{x \mapsto \_\} \ \{E \mapsto \_\} \ \text{dispose} \ E \ \{\text{emp}\} \ \{R[E/x]\}
\]

\[
x := E \ \{R\} (\text{the Hoare axiom})
\]

\[
\{E \mapsto \_\} \ x := [E] \ \{x = F \land E \mapsto \_\} \ \{E \mapsto \_\} \ [E] = F \ \{E \mapsto \_\}
\]
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

\[
\begin{align*}
\{\text{emp}\} x := \text{new}() & \{x \mapsto \_\} \\
\{E \mapsto \_\} \text{ dispose } E & \{\text{emp}\} \\
\{R[E/x]\} x := E \{R\} & \quad \text{(the Hoare axiom)} \\
\{E \mapsto F\} x := [E] \{x = F \land E \mapsto F\} & \quad (x \text{ not free in } E, F) \\
\{E \mapsto \_\} [E] := F & \{E \mapsto F\}
\end{align*}
\]
Three inference rules

- The frame rule:

\[
\{ Q \} C \{ R \} \implies \{ P \star Q \} C \{ P \star R \} (\text{modifies } C \cap \text{free } P = \{\})
\]

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

\[
\{ Q_1 \} C_1 \{ R_1 \} \quad \ldots \quad \{ Q_n \} C_n \{ R_n \}
\]

\[
\{ Q_1 \star Q_2 \star \cdots \star Q_n \} C_1 \| C_2 \| \cdots \| C_n \{ R_1 \star R_2 \star \cdots \star R_n \}
\]

- The CCR rule (has atrocious side condition):

\[
\{ (Q \star I_b) \land G \} C \{ R \star I_b \} \{ Q \}
\]

with \( b \) when \( G \) do \( C \od \{ R \} \]
Three inference rules

- **The frame rule**: \[
\frac{\{Q\} C \{R\}}{\{P \ast Q\} C \{P \ast R\}} \quad \text{(modifies } C \cap \text{free } P = \{\})
\]

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

- ▶ The CCR rule (has atrocious side condition):
Three inference rules

- The **frame** rule:  
  \[
  \frac{\{Q\} \ C \ \{R\}}{\{P \star Q\} \ C \ \{P \star R\}} \quad \text{(modifies} \ C \cap \text{free} \ P = \{\})
  \]

- The **concurrency** rule (has horrid side-condition):
  
  \[
  \frac{\{Q_1\} \ C_1 \ \{R_1\} \quad \{Q_2\} \ C_2 \ \{R_2\} \quad \ldots \quad \{Q_n\} \ C_n \ \{R_n\}}{\{Q_1 \star Q_2 \star \cdots \star Q_n\} \ C_1 \ \| \ C_2 \ \| \ \cdots \ \| \ C_n \ \{R_1 \star R_2 \star \cdots \star R_n\}}
  \]

Three inference rules

- The frame rule: \[
\begin{align*}
\{Q\} C \{R\} \quad \{P \ast Q\} C \{P \ast R\}
\end{align*}
\] (modifies \(C \cap \text{free } P = \{\}\))

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

\[
\begin{align*}
\{Q_1\} C_1 \{R_1\} \quad \{Q_2\} C_2 \{R_2\} \quad \ldots \quad \{Q_n\} C_n \{R_n\}
\end{align*}
\]
\[
\{Q_1 \ast Q_2 \ast \cdots \ast Q_n\} C_1 \parallel C_2 \parallel \cdots \parallel C_n \{R_1 \ast R_2 \ast \cdots \ast R_n\}
\]

- The CCR rule (has atrocious side condition):

\[
\begin{align*}
\{(Q \ast I_b) \land G\} C \{R \ast I_b\}
\end{align*}
\]
\[
\{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 $\rightarrow$) to allow sharing of heap;
Recent simplifications (not explained here)

- Permissions (fractions of $\rightarrow$, counts of $\leftrightarrow$) to allow sharing of heap;
- Variable permissions, to allow variables to be resource;
Recent simplifications (not explained here)

- Permissions (fractions of $\rightarrow$, counts of $\rightarrow\rightarrow$) 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 $\hookrightarrow$) 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

*slot:*

```
0 1
```

*data:*

```
← wide →
```
Nine lines are now ten,
with added auxiliary proof-variables

write: with bundle when true do pair := \texttt{not}(reading); wuse := pair od;
index := \texttt{not}(slot[pair]);
data[pair, index] := \texttt{item};
with bundle when true do slot[pair] := index; wuse := \texttt{-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 := \texttt{-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, $data[pair, index] \mapsto \_ \text{ replaces } data + 2 \times pair + index \mapsto \_$. 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, $\text{data}[\text{pair}, \text{index}] \mapsto -$ replaces $\text{data} + 2 \times \text{pair} + \text{index} \mapsto -$.

There is no change in meaning.)

\[
\text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, wuse_{0.5}, \text{pair}, \text{index} \\
\models \left( \begin{array}{c} 
\text{slot}[0] \mapsto - \star \text{slot}[1] \mapsto - \star \\
\text{if } wuse \geq 0 \text{ then } \text{data}[\text{pair}, \text{index}] \mapsto - \text{ else emp fi} 
\end{array} \right)
\]
What the reader owns

\[ \text{reading}_{0.5}, ruse_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \]

\[ \models \text{if } ruse \geq 0 \text{ then } data[pair, index] \mapsto \_ \text{ else } \text{emp} \text{ fi} \]
The bundle owns the rest

\[ \exists s \cdot \begin{cases} \text{slot}[0] \mapsto 0.5 \ast \text{slot}[1] \mapsto 0.5 \ast \text{s(0)} \ast \text{s(1)} \ast \\
\text{if } \text{wuse} \geq 0 \land \text{ruse} \geq 0 \text{ then} \\
\quad \text{data[reading, not(ruse)]} \mapsto _* \text{data[wuse, s(wuse)]} \mapsto _* \\
\quad \text{elsf } \text{wuse} \geq 0 \text{ then} \\
\quad \quad \text{data[wuse, s(wuse)]} \mapsto _* \ast \\
\quad \quad \text{data[not(wuse), s(not(wuse))]} \mapsto _* \ast \text{data[not(wuse), not(s(not(wuse)))]} \mapsto _* \\
\quad \text{elsf } \text{ruse} \geq 0 \text{ then} \\
\quad \quad \text{data[reading, not(ruse)]} \mapsto _* \ast \\
\quad \quad \text{data[not(reading), s(not(reading))]} \mapsto _* \ast \text{data[not(reading), not(s(not(reading)))]} \mapsto _* \\
\text{else} \\
\quad \text{data} \mapsto _, _, _, _ \\
\text{fi} \end{cases} \]
The writer

\[
\{ \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \xrightarrow{0.5} - \star \text{slot}[1] \xrightarrow{0.5} - \}
\]

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

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

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

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

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

\[
\{ \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \xrightarrow{0.5} - \star \text{slot}[1] \xrightarrow{0.5} - \}
\]
\[
\begin{aligned}
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \overset{0.5}{\rightarrow} - \star \text{slot}[1] \overset{0.5}{\rightarrow} - \}
\end{aligned}
\]

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

\[
\begin{aligned}
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \text{wuse} = \text{pair} \land \exists i \cdot \left( \begin{array}{c} \text{slot}[\text{pair}] \overset{0.5}{\rightarrow} i \star \text{slot[not(pair)]} \overset{0.5}{\rightarrow} - \star \\ \text{data}[\text{pair}, \text{not(i)}] \leftrightarrow - \end{array} \right) \}
\end{aligned}
\]

\[
\text{index} := \text{not(slot[\text{pair}]})
\]

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

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

with \textit{bundle} when true do \text{latest := pair} od

\[
\begin{aligned}
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \overset{0.5}{\rightarrow} - \star \text{slot}[1] \overset{0.5}{\rightarrow} - \}
\end{aligned}
\]
The writer

\[
\begin{align*}
\text{with bundle when true do } \& \text{ pair := not(reading); } wuse := \text{ pair } \text{ od;}
\end{align*}
\]

\[
\begin{align*}
l latest, slot, data, wuse, pair, index \models wuse = -1 \land \text{slot}[0] \overset{0.5}{\rightarrow} \_ \land \text{slot}[1] \overset{0.5}{\rightarrow} \_ \\
\text{index := not(slot[pair]);}
\end{align*}
\]

\[
\begin{align*}
l latest, slot, data, wuse, pair, index \models wuse = \text{pair} \land \exists i \cdot (\text{slot}[\text{pair}] \overset{0.5}{\rightarrow} i \land \text{slot}[\text{not(pair)}] \overset{0.5}{\rightarrow} \_ ) \\
data[\text{pair}, \text{not(i)}] \mapsto \_ \\
data[\text{pair}, \text{index}] \mapsto \item;
\end{align*}
\]

\[
\begin{align*}
\text{with bundle when true do } \text{slot}[\text{pair}] := \text{index}; \ wuse := -1 \text{ od;}
\end{align*}
\]

\[
\begin{align*}
\text{with bundle when true do } \text{latest := pair } \text{ od}
\end{align*}
\]

\[
\begin{align*}
l latest, slot, data, wuse, pair, index \models wuse = -1 \land \text{slot}[0] \overset{0.5}{\rightarrow} \_ \land \text{slot}[1] \overset{0.5}{\rightarrow} \_ \\
\end{align*}
\]
The writer

\[
\begin{align*}
\{ \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \} & \models \text{wuse} = -1 \land \text{slot}[0] \overset{0.5}{\rightarrow} - \ast \text{slot}[1] \overset{0.5}{\rightarrow} - \\
\text{with bundle when true do } & \text{pair} := \text{not}(\text{reading}); \text{wuse} := \text{pair} \text{ od;}
\end{align*}
\]

\[
\begin{align*}
\{ \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \} & \models \text{wuse} = \text{pair} \land \exists i \cdot \\
& \left( \text{slot}[\text{pair}] \overset{0.5}{\rightarrow} i \ast \text{slot}[\text{not}(\text{pair})] \overset{0.5}{\rightarrow} - \ast \right) \\
\text{index} & := \text{not}(\text{slot}[\text{pair}]);
\end{align*}
\]

\[
\begin{align*}
\{ \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \} & \models \text{wuse} = \text{pair} \land \\
& \left( \text{slot}[\text{pair}] \overset{0.5}{\rightarrow} \text{not}(\text{index}) \ast \text{slot}[\text{not}(\text{pair})] \overset{0.5}{\rightarrow} - \ast \right) \\
\text{data}[\text{pair}, \text{index}] & := \text{item};
\end{align*}
\]

\[
\begin{align*}
\{ \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \} & \models \text{wuse} = \text{pair} \land \\
& \left( \text{slot}[\text{pair}] \overset{0.5}{\rightarrow} \text{not}(\text{index}) \ast \text{slot}[\text{not}(\text{pair})] \overset{0.5}{\rightarrow} - \ast \right) \\
\text{with bundle when true do } & \text{slot}[\text{pair}] := \text{index}; \text{wuse} := -1 \text{ od;}
\end{align*}
\]

\[
\begin{align*}
\text{with bundle when true do } \text{latest} & := \text{pair} \text{ od}
\end{align*}
\]

\[
\begin{align*}
\{ \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \} & \models \text{wuse} = -1 \land \text{slot}[0] \overset{0.5}{\rightarrow} - \ast \text{slot}[1] \overset{0.5}{\rightarrow} -
\end{align*}
\]
The writer

\[
\begin{aligned}
\{ & \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \xrightarrow{0.5} - \star \text{slot}[1] \xrightarrow{0.5} - \\
\text{with bundle when true do pair := not(reading); wuse := pair od;}
\end{aligned}
\]

\[
\{ & \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \\
\models & \text{wuse} = \text{pair} \land \exists i . \left( \text{slot}[\text{pair}] \xrightarrow{0.5} i \star \text{slot}[\text{not}(\text{pair})] \xrightarrow{0.5} - \star \text{data}[\text{pair}, \text{not}(i)] \leftrightarrow - \right) \\
\text{index} & := \text{not}(\text{slot}[\text{pair}]);
\end{aligned}
\]

\[
\{ & \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \\
\models & \text{wuse} = \text{pair} \land \left( \text{slot}[\text{pair}] \xrightarrow{0.5} \text{not}(\text{index}) \star \text{slot}[\text{not}(\text{pair})] \xrightarrow{0.5} - \star \text{data}[\text{pair}, \text{index}] \leftrightarrow - \right) \\
\text{data}[\text{pair}, \text{index}] & := \text{item};
\end{aligned}
\]

\[
\{ & \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \\
\models & \text{wuse} = \text{pair} \land \left( \text{slot}[\text{pair}] \xrightarrow{0.5} \text{not}(\text{index}) \star \text{slot}[\text{not}(\text{pair})] \xrightarrow{0.5} - \star \text{data}[\text{pair}, \text{index}] \leftrightarrow \text{item} \right) \\
\text{with bundle when true do slot}[\text{pair}] := \text{index}; \text{wuse} := -1 \text{ od;}
\end{aligned}
\]

\[
\{ & \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \xrightarrow{0.5} - \star \text{slot}[1] \xrightarrow{0.5} - \\
\text{with bundle when true do latest := pair od}
\end{aligned}
\]

\[
\{ & \text{latest}_0.5, \text{slot}_0.5, \text{data}_{0.33}, \text{wuse}_0.5, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \xrightarrow{0.5} - \star \text{slot}[1] \xrightarrow{0.5} - 
\}
\]
Details of the first writer step

\[
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \xrightarrow{0.5} \star \text{slot}[1] \xrightarrow{0.5} - \}
\]

with \textit{bundle} when true do

\[
\text{pair} := \text{not}(\text{reading}) ;
\]

\[
\text{wuse} := \text{pair}
\]

\[
\text{od;}
\]

\[
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \}
\models \text{wuse} = \text{pair} \land \exists i \cdot \left( \text{slot}[\text{pair}] \xrightarrow{0.5} i \star \text{slot}[^{\text{not}(\text{pair})}] \xrightarrow{0.5} \star \text{data}[\text{pair}, \text{not}(i)] \xrightarrow{-} \right)
\]
Details of the first writer step

\[
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \begin{align*}
\text{wuse} &= -1 \land \text{slot} \mapsto s(0), s(1) \\
\text{data}[\neg \text{reading}, s(\neg \text{reading})] &\mapsto _\ast \text{data}[\neg \text{reading}, \neg s(\neg \text{reading})] \mapsto _\ast \\
\text{if } \text{ruse} \geq 0 \text{ then } \text{data}[\text{reading}, \neg \text{ruse}] &\mapsto _\ast \\
\text{else } \text{data}[\text{reading}, s(\text{reading})] &\mapsto _\ast \text{data}[\text{reading}, \neg s(\text{reading})] \mapsto _\ast \\
\text{fi} \\
\text{pair} &:= \neg \text{reading};
&\end{align*}
\}
\]

\[
\text{wuse} := \text{pair}
\]

od;

\[
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \begin{align*}
\text{wuse} &= \text{pair} \land \exists i \cdot \left( \text{slot}[\text{pair}] \mapsto _{0.5} i \ast \text{slot}[\neg \text{pair}] \mapsto _{0.5} _\ast \text{data}[\text{pair}, \neg s(i)] \mapsto _\ast \right) 
\end{align*}
\}
\]
Details of the first writer step

\[
\left\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \models \text{wuse} = -1 \land \text{slot}[0] \xrightarrow{0.5} \text{wuse} \xrightarrow{0.5} \ast \text{slot}[1] \right. \\
\text{with bundle when true do} \\
\left\{ \text{latest}, \text{reading}_{0.5}, \text{slot}, \text{data}_{0.66}, \text{wuse}, \text{pair}, \text{index} \\
\quad \text{wuse} = -1 \land \text{slot} \mapsto s(0), s(1) \ast \\
\quad \text{data}[\text{not}(\text{reading}), s(\text{not}(\text{reading}))] \mapsto \_ \ast \text{data}[\text{not}(\text{reading}), \text{not}(s(\text{not}(\text{reading})))] \mapsto \_ \\
\quad \text{if } ruse \geq 0 \text{ then } \text{data}[\text{reading}, s(\text{not}(ruse))] \mapsto \_ \\
\quad \text{else } \text{data}[\text{reading}, s(\text{reading})] \mapsto \_ \ast \text{data}[\text{reading}, s(\text{reading})] \mapsto \_ \\
\quad \text{fi} \\
\right. \\
\text{pair} := \text{not}(\text{reading}); \\
\left\{ \text{latest}, \text{reading}_{0.5}, \text{slot}, \text{data}_{0.66}, \text{wuse}, \text{pair}, \text{index} \\
\quad \text{wuse} = -1 \land \text{pair} = \text{not}(\text{reading}) \land \text{slot} \mapsto s(0), s(1) \ast \\
\quad \text{data}[\text{not}(\text{reading}), s(\text{not}(\text{reading}))] \mapsto \_ \ast \text{data}[\text{not}(\text{reading}), \text{not}(s(\text{not}(\text{reading})))] \mapsto \_ \\
\quad \text{if } ruse \geq 0 \text{ then } \text{data}[\text{reading}, s(\text{not}(ruse))] \mapsto \_ \\
\quad \text{else } \text{data}[\text{reading}, s(\text{reading})] \mapsto \_ \ast \text{data}[\text{reading}, s(\text{reading})] \mapsto \_ \\
\quad \text{fi} \\
\right. \\
\text{wuse} := \text{pair} \\
\right. \\
\text{od;} \\
\left\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \\
\quad \models \text{wuse} = \text{pair} \land \exists i \cdot \left( \text{slot}[\text{pair}] \xrightarrow{0.5} i \ast \text{slot}[\text{not}(\text{pair})] \xrightarrow{0.5} \_ \ast \text{data}[\text{pair}, \text{not}(i)] \mapsto \_ \right) \right. \}
Details of the first writer step

\[
\begin{align*}
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \} \models \text{wuse} = -1 \land \text{slot}[0] & \xrightarrow{0.5} - \ast \text{slot}[1] \xrightarrow{0.5} - \\
\text{with bundle when true do} \\
\{ \text{latest}, \text{reading}_{0.5}, \text{slot}, \text{data}_{0.66}, \text{wuse}, \text{pair}, \text{index} \} \\
& \models \text{wuse} = -1 \land \text{slot} \rightarrow s(0), s(1) \ast \\
& \models \exists s \cdot \\
& \quad \text{if } \text{ruse} \geq 0 \quad \text{then } \text{data}[\text{reading}, \text{not}(\text{ruse})] \rightarrow - \\
& \quad \quad \text{else } \text{data}[\text{reading}, s(\text{reading})] \rightarrow - \ast \text{data}[\text{reading}, \text{not}(s(\text{reading}))] \rightarrow - \\
\text{pair} := \text{not}(\text{reading}) \text{; } \\
\{ \text{latest}, \text{reading}_{0.5}, \text{slot}, \text{data}_{0.66}, \text{wuse}, \text{pair}, \text{index} \} \\
& \models \text{wuse} = -1 \land \text{pair} = \text{not}(\text{reading}) \land \text{slot} \rightarrow s(0), s(1) \ast \\
& \models \exists s \cdot \\
& \quad \text{if } \text{ruse} \geq 0 \quad \text{then } \text{data}[\text{reading}, \text{not}(\text{ruse})] \rightarrow - \\
& \quad \quad \text{else } \text{data}[\text{reading}, s(\text{reading})] \rightarrow - \ast \text{data}[\text{reading}, \text{not}(s(\text{reading}))] \rightarrow - \\
\text{wuse} := \text{pair} \text{; } \\
\{ \text{latest}, \text{reading}_{0.5}, \text{slot}, \text{data}_{0.66}, \text{wuse}, \text{pair}, \text{index} \} \\
& \models \text{wuse} = \text{pair} \land \text{pair} = \text{not}(\text{reading}) \land \text{slot} \rightarrow s(0), s(1) \ast \\
& \models \exists s \cdot \\
& \quad \text{if } \text{ruse} \geq 0 \quad \text{then } \text{data}[\text{reading}, \text{not}(\text{ruse})] \rightarrow - \\
& \quad \quad \text{else } \text{data}[\text{reading}, s(\text{reading})] \rightarrow - \ast \text{data}[\text{reading}, \text{not}(s(\text{reading}))] \rightarrow - \\
\text{od;} \\
\{ \text{latest}_{0.5}, \text{slot}_{0.5}, \text{data}_{0.33}, \text{wuse}_{0.5}, \text{pair}, \text{index} \} \\
& \models \text{wuse} = \text{pair} \land \exists i \cdot \left( \text{slot}[\text{pair}] \xrightarrow{0.5} i \ast \text{slot}[\text{not}(\text{pair})] \xrightarrow{0.5} - \ast \text{data}[\text{pair}, \text{not}(i)] \rightarrow - \right)
\end{align*}
\]
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;

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!

\{
    \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1
\}

with \textit{bundle} when true do \text{pair} := \text{latest} od;

\{
    \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1
\}

with \textit{bundle} when true do \text{reading} := \text{pair} od;

with \textit{bundle} when true do \text{index} := \text{slot}[	ext{pair}]; \text{ruse} := \text{index} od;

\text{read} := \text{data}[	ext{pair}, \text{index}];

\text{with} \textit{bundle} \text{ when true do} \text{ruse} := -1 \text{ od}

\{
    \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{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 \land 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 \models ruse = -1 \}
The reader is even easier than the writer!

\[ \{ \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \} \]

with bundle when true do \( \text{pair} := \text{latest} \) od;

\[ \{ \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \} \]

with bundle when true do \( \text{reading} := \text{pair} \) od;

\[ \{ \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \land \text{reading} = \text{pair} \} \]

with bundle when true do \( \text{index} := \text{slot}[\text{pair}]; \text{ruse} := \text{index} \) od;

\[ \{ \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} \geq 0 \land \text{reading} = \text{pair} \land \text{data}[\text{pair}, \text{index}] \mapsto \_ \} \]

\( \text{read} := \text{data}[\text{pair}, \text{index}] \);

\[ \text{with bundle when true do ruse} := -1 \text{ od} \]

\[ \{ \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \} \]
The reader is even easier than the writer!

\[
\begin{align*}
\{ & \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \} \\
& \text{with bundle when true do } \text{pair} := \text{latest} \text{ od;} \\
\{ & \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \} \\
& \text{with bundle when true do } \text{reading} := \text{pair} \text{ od;} \\
\{ & \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \land \text{reading} = \text{pair} \} \\
& \text{with bundle when true do } \text{index} := \text{slot}[\text{pair}]; \text{ruse} := \text{index} \text{ od;} \\
\{ & \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} \geq 0 \land \text{reading} = \text{pair} \land \text{data}[\text{pair}, \text{index}] \mapsto \_ \} \\
& \text{read} := \text{data}[\text{pair}, \text{index}]; \\
\{ & \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} \geq 0 \land \text{reading} = \text{pair} \land \exists i \cdot \text{data}[\text{pair}, \text{index}] \mapsto i \land \text{read} = i \} \\
& \text{with bundle when true do } \text{ruse} := -1 \text{ od} \\
\{ & \text{reading}_{0.5}, \text{ruse}_{0.5}, \text{data}_{0.33}, \text{pair}, \text{index} \models \text{ruse} = -1 \}
\end{align*}
\]
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 := not(reading); wuse := pair 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 very similar to
with bundle when true do pair := 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.
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.
Nice Proofs™

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

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.
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.
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).
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.
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.