UNIVERSITY OF TWENTE.

VERCORS: VERIFICATION OF CONCURRENT SYSTEMS

MARIEKE HUISMAN

UNIVERSITY OF TWENTE, NETHERLANDS

SEVENTH FRAMEWORK PROGRAMME

erc European Research Council

Established by the European Commission

OUTLINE OF THIS LECTURE

- How to ensure software quality?
- Classical program logic
- Separation logic
- The next challenge: concurrent software
- Permission-based separation logic
- Functional properties of concurrent programs
- Reasoning about GPU kernels

SOFTWARE QUALITY

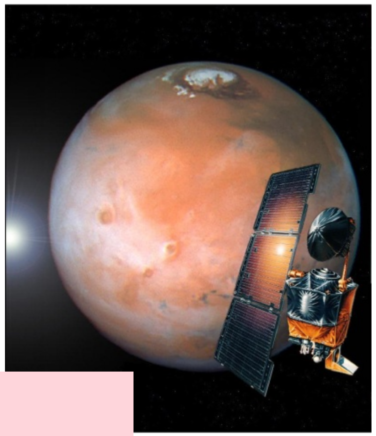
Peter Naur
1968
Working on the *Software crisis* report

SOFTWARE QUALITY IS A CHALLENGE

ICT problems Dutch gouvernment

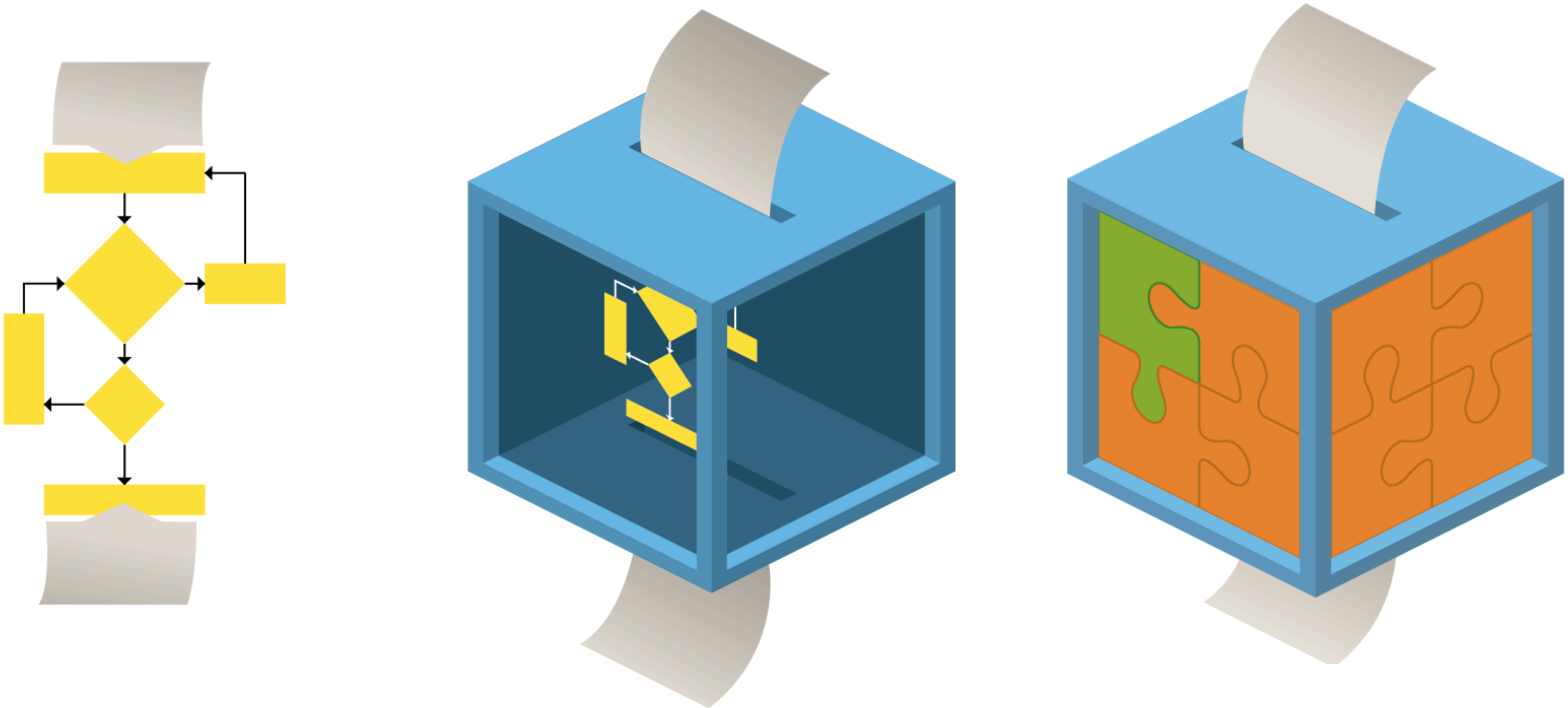
Toyata Prius: software errors due to lack of testing

Unreachable banks because of network problems



Mars Climate Orbiter: Crash due to different units

OUR APPROACH



Software Box it Check the components

SPECIFYING PROGRAM BEHAVIOUR

Use logic to describe behaviour of program components

- Precondition: what do you know in advance?

  Example: increaseBy(int n)

  requires n > 0

- Postcondition: what holds afterwards

  Example: increaseBy(int n)

  x increased by n

  ensures x == old(x) + n

Dates back to the 60-ies

Bob Floyd
(1936 – 2001)

Tony Hoare
(1934 -)

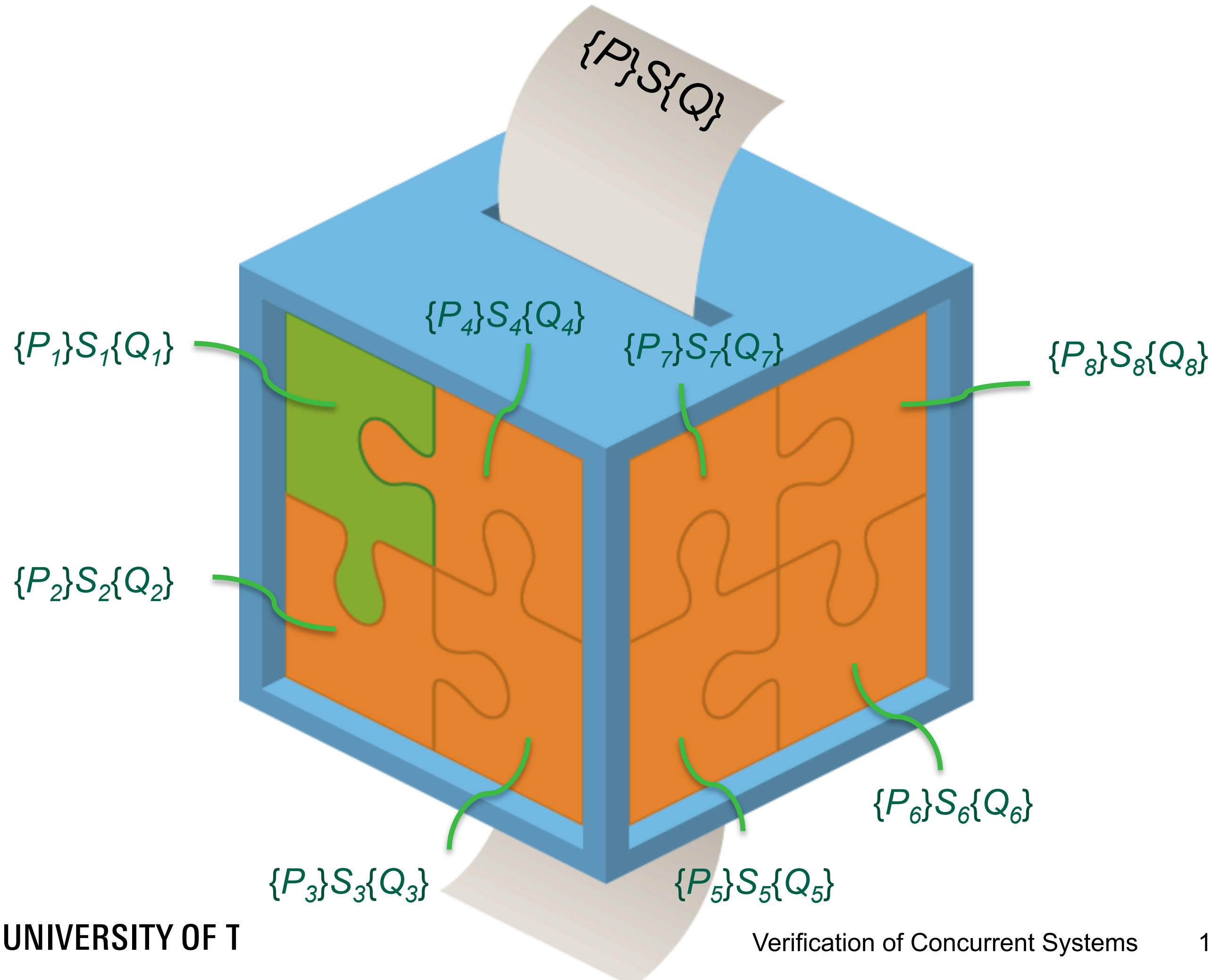
Hoare triples

Notation: {P}S{Q}

precondition

postcondition

HOARE TRIPLES FOR ALL COMPONENTS

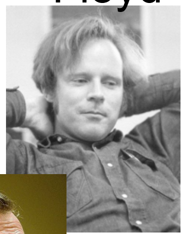


HISTORY OF PROGRAM VERIFICATION

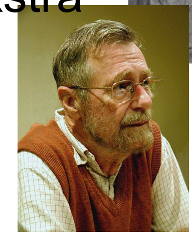
My thesis
(around 2000)

State-of-the-art

Floyd - Hoare



Dijkstra



LOOP FOREVER

KeY

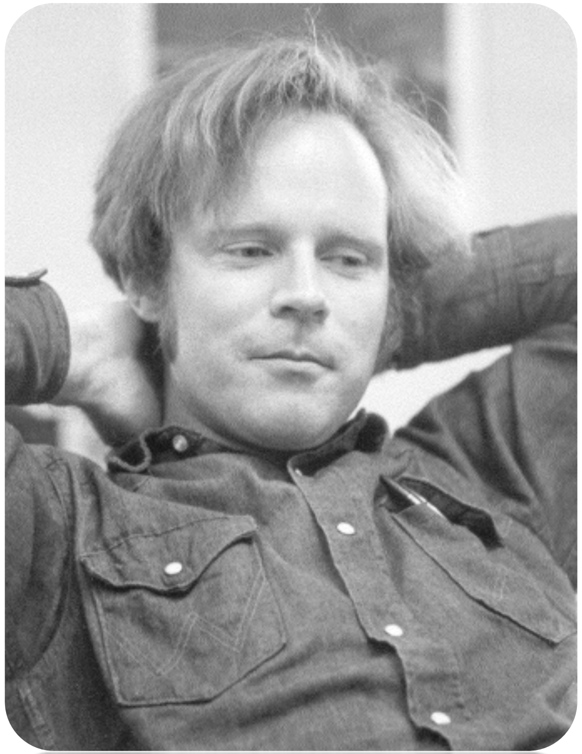
Krakatoa

JML

Code Contracts

Dafny

PROGRAM LOGIC



Bob Floyd
1936 - 2001

PRE- AND POSTCONDITIONS

- Precondition: property that should be satisfied when method is called – otherwise correct functioning of method is not guaranteed
- Postcondition: property that method establishes – caller can assume this upon return of method

- Method specification is contract between implementer and caller of method.
  - Caller promises to call method only in states in which precondition holds
  - Implementer guarantees postcondition will be established

HOARE TRIPLES

- *{P}S{Q}*

1934 -

- Due to Tony Hoare (1969)

- Meaning: if *P* holds in initial state *s*, and execution of *S* in *s* terminates in state *s'*, then *Q* holds in *s'*
- Formally:

  $\{P\}S\{Q\} = \forall s.P(s) \wedge (S,s) \rightarrow s' \Rightarrow Q(s')$

HOARE LOGIC

- Hoare triples: specify behaviour of methods
- How to guarantee that methods indeed respect this behaviour?

- Collection of derivation rules to reason about Hoare triples
- Rules defined by induction on the program structure
- Proven sound w.r.t. program semantics

- Here: a very simple language, but exists for more complicated languages

AXIOMS

Skip $$\frac{}{\{P\}Skip\{P\}}$$

Ass. $$\frac{}{\{P[v := e]\}v := e\{P\}}$$

STATEMENT DECOMPOSITION

Seq $$\frac{\{P\}S1\{Q\} \quad \{Q\}S2\{R\}}{\{P\}S1;S2\{R\}}$$

If $$\frac{\{P \wedge b\}S1\{Q\} \quad \{P \wedge \neg b\}S2\{Q\}}{\{P\}\text{if } (b)\ S1 \text{ else } S2\ \{Q\}}$$

(*): precondition strengthening

EXAMPLE

$a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\ [z := 1] =$
$a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge 1 = 1$

Ass

$$\frac{}{\{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge 1 = 1\}\ z := 1\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}}$$

$a \geq 0 \wedge n \geq 0 \wedge k = 0 \Rightarrow a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge 1 = 1$

$a \geq 0 \wedge n \geq 0 \wedge k = 0\ [k := 0] =$
$a \geq 0 \wedge n \geq 0 \wedge 0 = 0$

Ass

$$\frac{}{\{a \geq 0 \wedge n \geq 0 \wedge 0 = 0\}\ k := 0\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0\}}$$

$$\frac{a \geq 0 \wedge n \geq 0 \Rightarrow a \geq 0 \wedge n \geq 0 \wedge 0 = 0}{\{a \geq 0 \wedge n \geq 0\}\ k := 0\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0\}}\ (*)$$

$$\frac{}{\{a \geq 0 \wedge n \geq 0 \wedge k = 0\}\ z := 1\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}}\ (*)$$

Seq

$$\frac{}{\{a \geq 0 \wedge n \geq 0\}\ k := 0;\ z := 1\ \{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}}$$

Seq

$$\frac{\{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}\ \text{while}\ (k < n)\ \{z := z * a;\ k := k + 1;\}\ \{z = a\hat{\ }n\}}{\{a \geq 0 \wedge n \geq 0\}\ k := 0;\ z := 1;\ \text{while}\ (k < n)\ \{z := z * a;\ k := k + 1;\}\ \{z = a\hat{\ }n\}}$$

RULES OF CONSEQUENCE

Pre. Str. $$\frac{P \Rightarrow P' \qquad \{P'\}S\{Q\}}{\{P\}S\{Q\}}$$

Post. Weak. $$\frac{\{P\}S\{Q\} \qquad Q \Rightarrow Q'}{\{P\}S\{Q'\}}$$

LOOPS

$$\text{Loop}\ \frac{\{I \wedge b\}S\{I\}}{\{I\}\text{while}\ (b)\ S\ \{I \wedge \neg b\}}$$

- I called loop invariant
- Preserved by every iteration of the loop
- Can in general not be found automatically

- Notation in our language
  invariant I;
  while (b) S

EXAMPLE: METHOD POWER

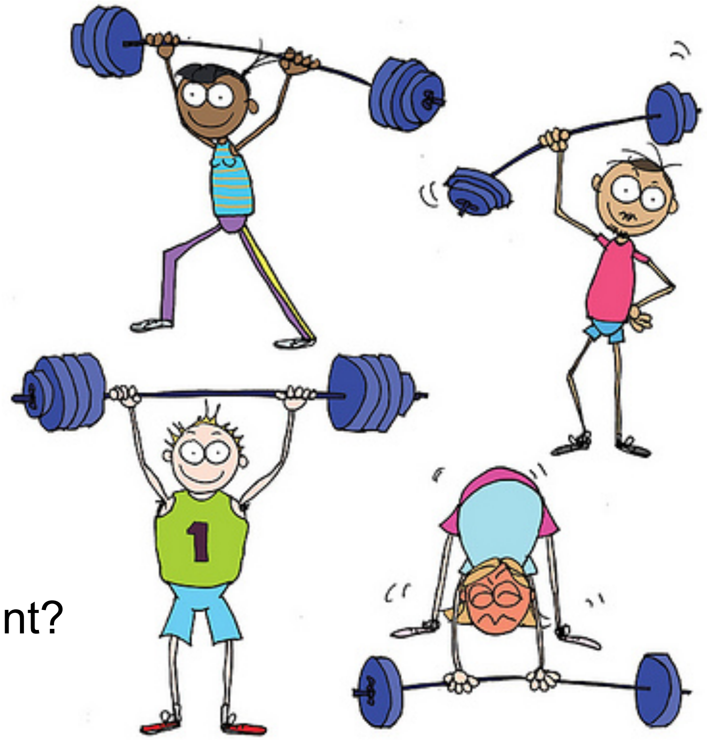
$\{ a \geq 0 \wedge n \geq 0 \}$

```
k := 0;
z := 1;
```

$\{ a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1 \}$

```
while (k < n)
   {  z := z * a;
      k := k + 1;
   }
```

$\{ z = a\hat{}n \}$



What should be the loop invariant?

$z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge k \geq 0$

EXAMPLE CONTINUED

Ass

$\{z*a = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}\ z := z * a\ \{z = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}$

$z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge !(k = n) \Rightarrow z*a = a\hat{}(k+1) \wedge a \geq 0 \wedge k + 1 \leq n$

Pre. Str.

Ass

$\{z = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}\ k := k + 1\ \{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge !(k = n)\}\ z := z * a\ \{z = a\hat{}(k+1) \wedge k + 1 \leq n \wedge a \geq 0\}$

Seq

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge !(k = n)\}\ z := z * a;\ k := k + 1\ \{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$

Loop

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$ while $(!(k = n))\ \{z := z * a;\ k := k + 1;\}\ \{z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge k = n\}$

$z = a\hat{}k \wedge k \leq n \wedge a \geq 0 \wedge k = n \Rightarrow z = a\hat{}n$

Post. Weak.

$\{z = a\hat{}k \wedge k \leq n \wedge a \geq 0\}$ while $(!(k = n))\ \{z := z * a;\ k := k + 1;\}\ \{z = a\hat{}n\}$

$a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1 \Rightarrow z = a\hat{}k \wedge k \leq n \wedge a \geq 0$

Pre. Str.

$\{a \geq 0 \wedge n \geq 0 \wedge k = 0 \wedge z = 1\}$ while $(!(k = n))\ \{z := z * a;\ k := k + 1;\}\ \{z = a\hat{}n\}$

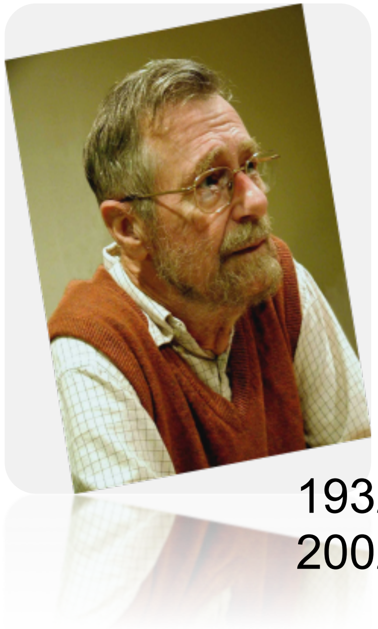
TOOL SUPPORT FOR PROGRAM VERIFICATION

Rustan Leino

A CALCULATIONAL APPROACH

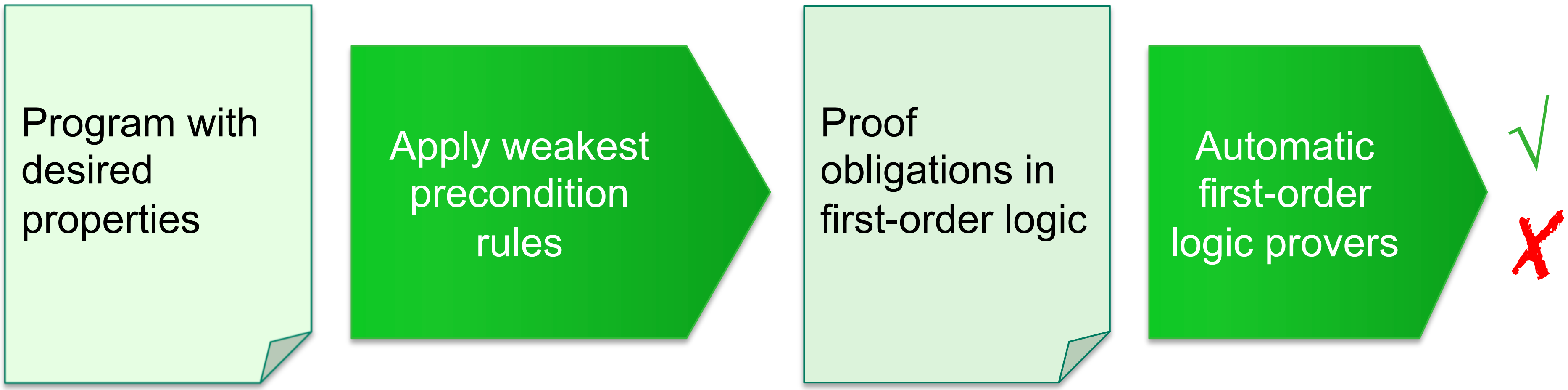
Many intermediate predicates can be computed

- Weakest liberal precondition wp(S,Q)
- The weakest predicate such that {wp(S,Q)}S{Q}
- Due to Edsger Dijkstra (1975)
- Calculus allows to compute weakest preconditions of sequential code
- Proof obligations: preconditions imply weakest liberal preconditions
- Loop invariants still given explicitly



1932 - 2002

AUTOMATION



Preferably also counter example: why does program not have desired behaviour

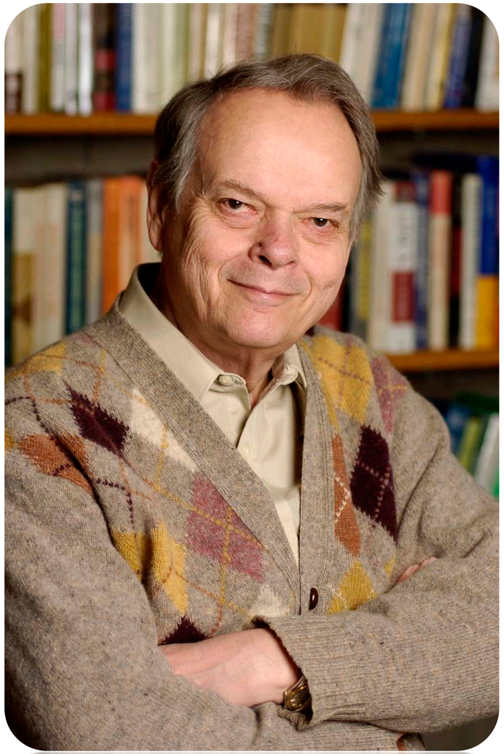
LIMITATIONS OF CLASSICAL PROGRAM LOGIC

- Idealised language
- No side-effects in conditions
- No pointers
- No multi-threading

Separation logic

- Reasoning about pointers
- Natural extension to multi-threading

SEPARATION LOGIC



John Reynolds
1935 - 2013

THE CHALLENGE OF POINTER PROGRAMS

```
class C {

  D f;
  D g;
}

class D {
  int x := 0;
}
```

```
ensures c.g.x = 0;
method m() {
  c := new C;
  d := new D;
  c.f := d;
  c.g := d;
  update_x(c.f, 3);
}

ensures d.x = v;
method update_x(d, v) {
  d.x := v;
}
```

This should not be verified!

SEPARATION LOGIC

- State distinguishes heap and store
- Heap contains dynamically allocated data that exists during run-time of program

  (Object-oriented program: the objects are stored on the heap)
- Store (or call stack) contains data related to method call (parameters, local variables)
- Heap accessed by pointers
- Locations on heap can be aliased
- Main idea: assertions about state can be decomposed into assertions about disjoint substates

INTUITIONISTIC SEPARATION LOGIC

Syntax extension of predicate logic:

$$\varphi ::= e.f \rightarrow e' \mid \varphi \ast \varphi \mid \varphi -\!\!\ast\ \varphi \mid \ldots$$

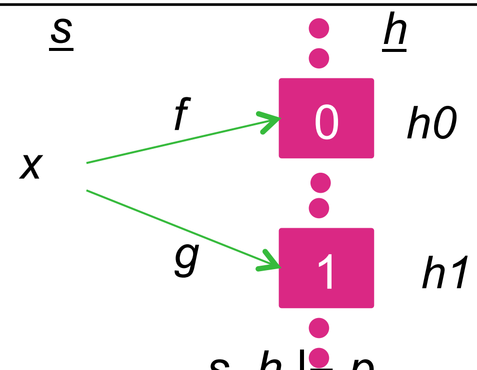
where e is an expression, and f a field

Meaning:

- $e.f \rightarrow e'$ – heap contains location pointed to by $e.f$, containing the value given by the meaning e'
- $\varphi 1 \ast \varphi 2$ – heap can be split in disjoint parts, satisfying φ1 and φ2, respectively
- $\varphi 1 -\!\!\ast\ \varphi 2$ – if heap extended with part that satisfies φ1, composition satisfies φ2

Monotone w.r.t. extensions of the heap

EXAMPLES INTUITIONISTIC SEPARATION LOGIC



Suppose x is an object in the store, with fields f and g

$h0 = [\ s(x.f) : 0]$ $\qquad$ $h1 = [\ s(x.g) : 1]$ $\qquad$ x

p	$s, h \models p$	p	$s, h \models p$
$x.f \to 0$	$h0 \subseteq h$	$x.f \to 0 \ast (x.f \to 0 \vee x.g \to 1)$	$h0 \ast h1 \subseteq h$
$x.g \to 1$	$h1 \subseteq h$	$(x.f \to 0 \vee x.g \to 1) \ast (x.f \to 0 \vee x.g \to 1)$	$h0 \ast h1 \subseteq h$
$x.f \to 0 \ast x.g \to 1$	$h0 \ast h1 \subseteq h$	$x.f \to 0 \ast x.g \to 1 \ast (x.f \to 0 \vee x.g \to 1)$	false
$x.f \to 0 \ast x.f \to 0$	false	$x.f \to 0 \ast$ true	$h0 \subseteq h$
$x.f \to 0 \vee x.g \to 1$	$h0 \subseteq h$ or $h1 \subseteq h$		

EXAMPLE: CLASS BOX

```
class Box {

int cnts;

requires this.cnts → _;
ensures this.cnts → o;
void set (int o) {
    this.cnts = o;
    return null;
}
```

requires *P;*
ensures Q;
void m(..) { ... }
alternative notation for
{*P*} method m() {*Q*}

```
requires this.cnts → X;
ensures this.cnts → X ∧  result = X;
int get() {
 return this.cnts;
}

}
```

Compare with specifications in classical Hoare logic
requires true;
ensures this.cnts == o;

ADVANTAGES OF SEPARATION LOGIC

- Reasoning about programs with pointers
- Two interpretations $e.f \rightarrow v$
  - Field $e.f$ contains value v
  - Permission to access field $e.f$

    A field can only be accessed or written if $e.f \rightarrow$ _ holds!
- Implicit disjointness of parts of the heap allows reasoning about (absence) of aliasing

  $x.f \rightarrow$ _ $\ast$ $y.f \rightarrow$ _ implicitly says that x and y are not aliases
- Local reasoning
  - only reason about heap that is actually accessed by code fragment
  - rest of heap is implicitly unaffected: frame rule

UPDATES AND LOOKUP OF THE HEAP

$$\frac{}{\{e.f \rightarrow \_\}\ e.f := v\ \{e.f \rightarrow v\}}$$

$$\frac{}{\{X = e \wedge X.f \rightarrow Y\} v := e.f\ \{X.f \rightarrow Y \wedge v = Y\}}$$

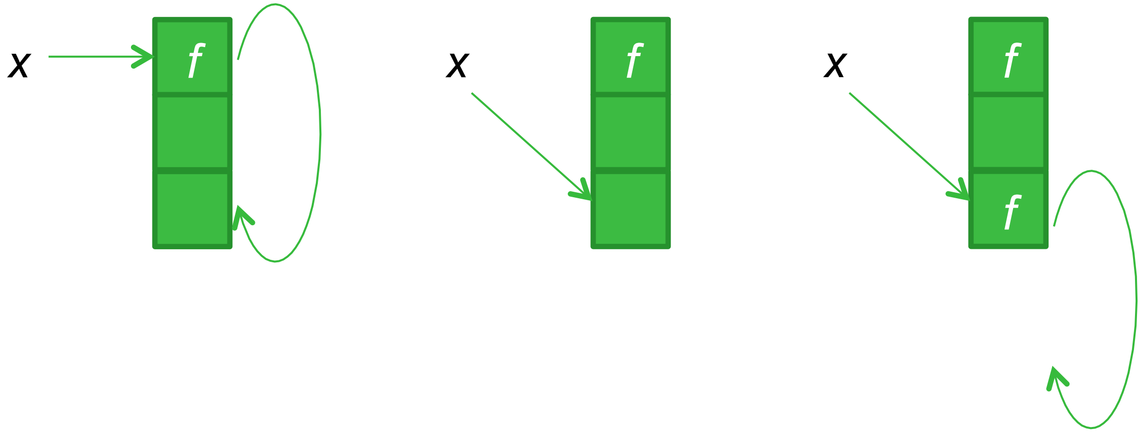
where X and Y are logical variables

- For simplicity v is typically assumed to be a simple (unqualified) expression
- Any assignment $e.f := e'.g$ can be split up in $x := e'.g;\ e.f := x$

Logical variables needed to handle $x := x.f$

WHY IS THE LOGICAL VARIABLE NEEDED?

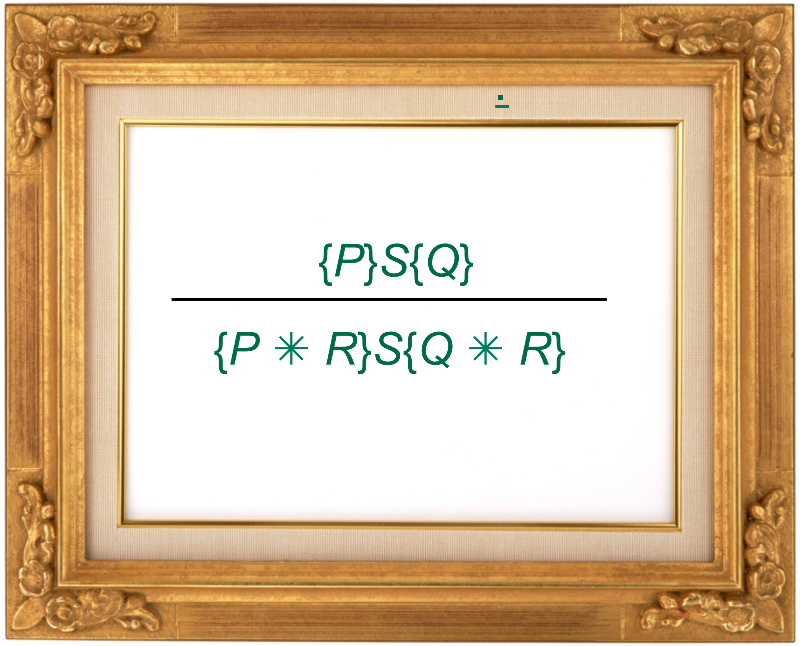
$\{x.f \rightarrow Y\} x := x.f \{x.f \rightarrow Y \wedge x = Y\}$ is not correct!



But this is:

$\{X = e \wedge X.f \rightarrow Y\} v := e.f \{X.f \rightarrow Y \wedge v = Y\}$

FRAME RULE



$$\frac{\{P\}S\{Q\}}{\{P \ast R\}S\{Q \ast R\}}$$

where R does not contain any variable that is modified by S.

THE CHALLENGE OF POINTER PROGRAMS

```
class C {

  D f;
  D g;
}

class D {
  int x := 0;
}
```

```
method m() {
  c := new C;
  d := new D;
  c.f := d;
  c.g := d;
  update_x(c.f, 3);
}
```

$c.f \rightarrow \_ \ast c.g \rightarrow \_$
does not hold

Empty frame

```
ensures d.x = v;
method update_x(d, v) {
  d.x := v;
}
```

ABSTRACT PREDICATES

Matthew Parkinson

SPECIFYING DATA STRUCTURES

- Abstract predicates represent and encapsulate state, with appropriate operations
- Abstract predicates are scoped
  - Code verified in scope can use name and body
  - Code verified out of scope can only use name
- Explicit open/close axiom to open definition of abstract predicate, provided it is in scope

$$\alpha(x1, .., xn) = P \text{ in scope} \mid\!- \alpha(e1, .., en) \Leftrightarrow P[x1 := e1, .. xn := en]$$

ABSTRACT PREDICATES ON LIST

```
class Node {
        int val;
        Node next;
}
```

- Predicate list
  - pred list (i)= (i = null) $\vee$ $\exists$ Node j, int a. i.val $\rightarrow a \ast i$.next $\rightarrow j \ast$ list j

  recognises list structure
- Predicate list:
  - pred list (ϵ, i) = (i = null)
  - pred list ((a.α), i) = $\exists$Node j. i.val $\rightarrow a \ast i$.next $\rightarrow j \ast$ list α j

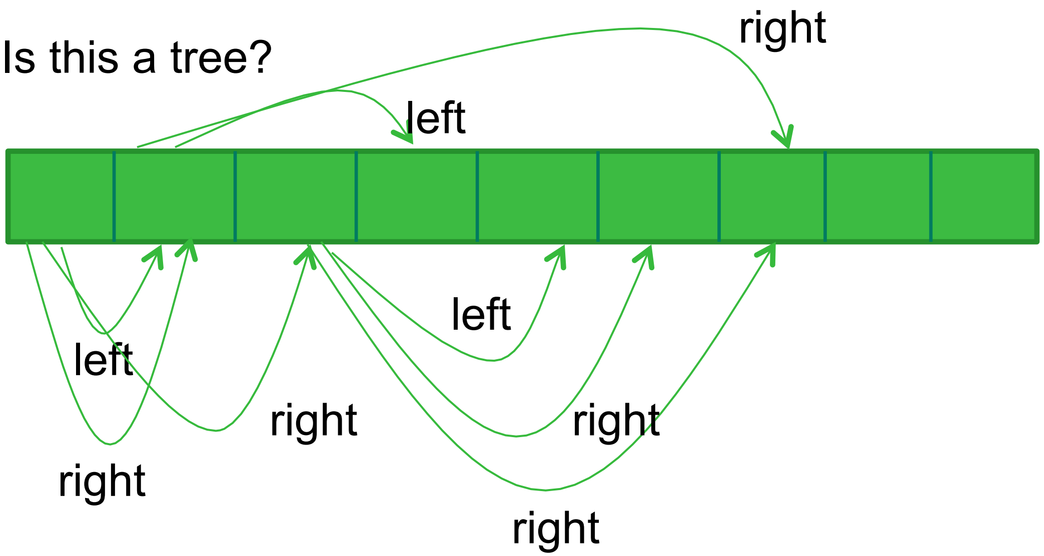
  relates list content with abstract list value
- Operations like append and reverse in specifications can be defined on abstract type

ABSTRACT PREDICATE ON TREES

- tree i = (i = null) $\vee$ $\exists$ Node $j, k.$ i.left $\rightarrow j \ast i$.right $\rightarrow k \ast$ tree $j \ast$ tree k

  recognises tree structure

  Is this a tree?



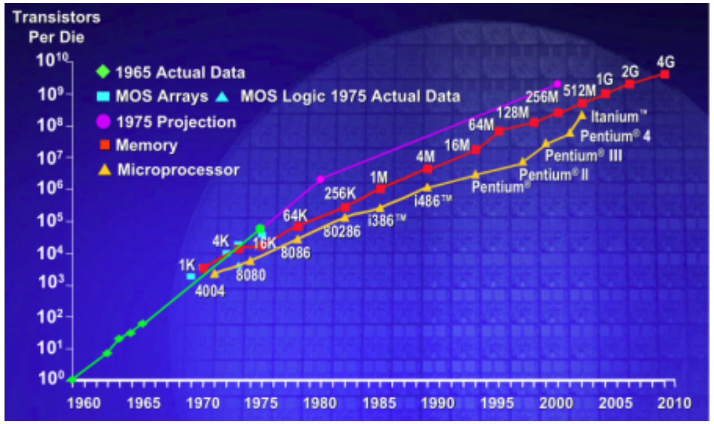
NO

CONCURRENCY: THE NEXT CHALLENGE

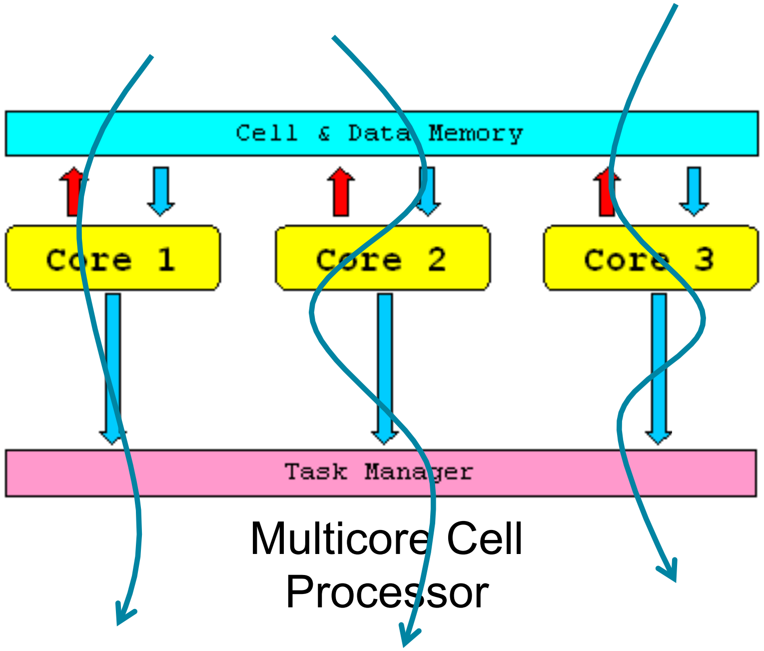
Doug Lea

THE FUTURE OF COMPUTING IS MULTICORE

Single core processors:
The end of Moore's law



Solution:
Multi-core processors

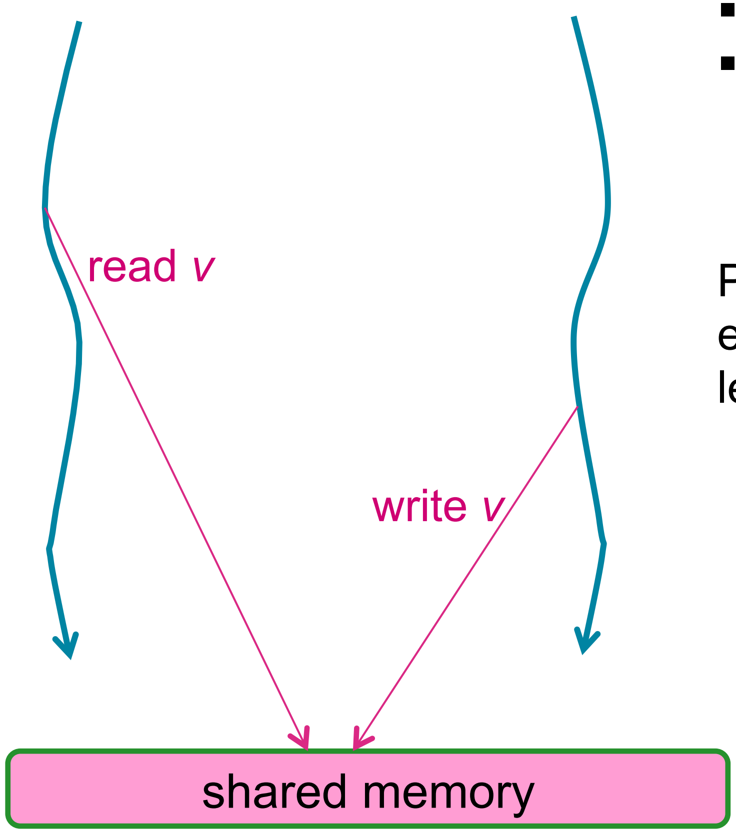


Multicore Cell Processor

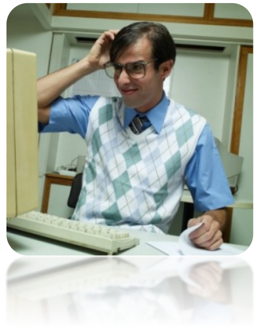
Multiple threads of execution

Coordination problem shifts from hardware to software

MULTIPLE THREADS CAUSE PROBLEMS

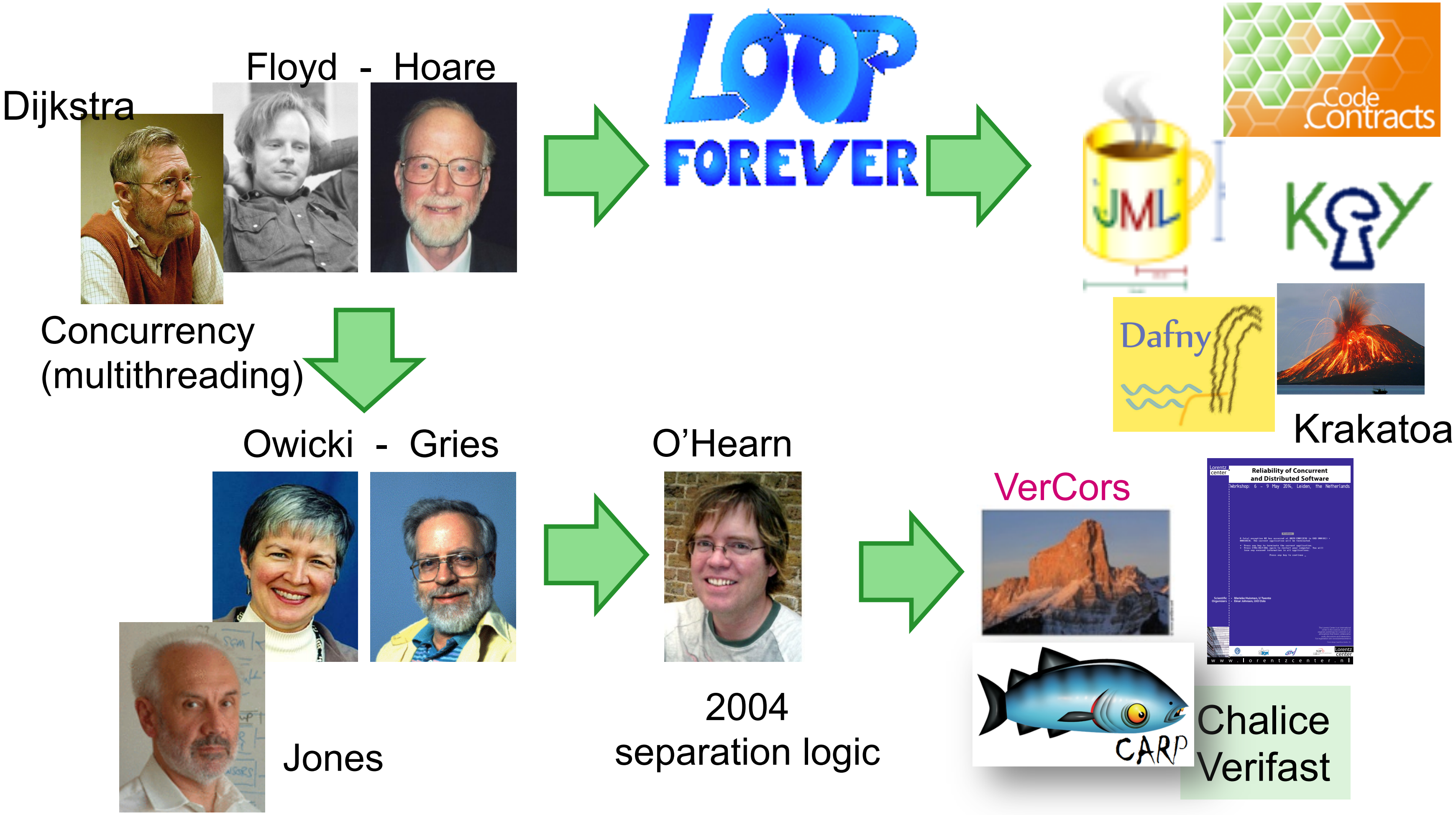


- Order?
- More threads?

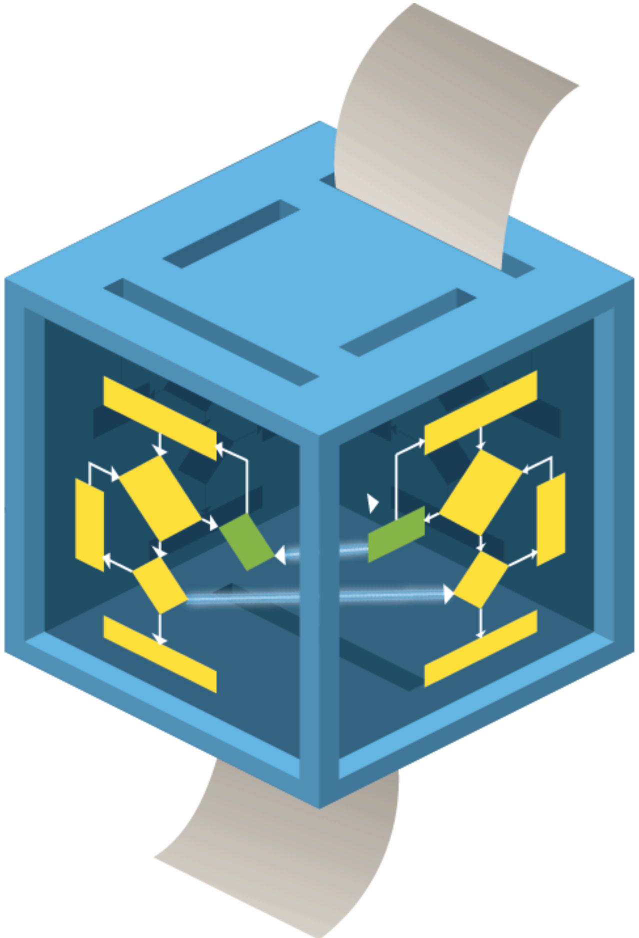


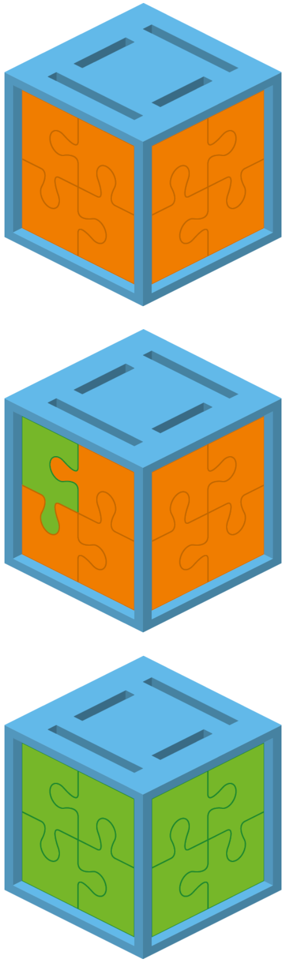
Possible consequences:
errors such as data races caused lethal bugs as in Therac-25

VERIFICATION OF MULTITHREADED PROGRAMS



OUR APPROACH





SPECIFICATIONS IN A CONCURRENT SETTING

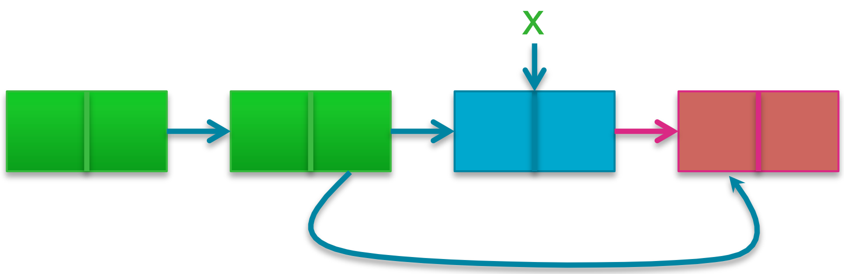
requires true

ensures x is the last element in the list

```
void addToList(Elem x) {
    // code
}
```

Any other thread might invalidate this!

‘x is in the list’ cannot even be guaranteed!



Except when no other thread can update the list

SOME HISTORY: REASONING ABOUT THREADS

Susan Owicki

OWICKI-GRIES METHOD (1975)

- For each thread: give a complete proof outline
- Verify each thread w.r.t. the proof outline
- For each annotation in the proof outline, show that it cannot be invalidated by any other thread: interference freedom

David Gries

EXAMPLE OWICKI-GRIES

$\{x = 0 \wedge y = 0\} x := x + 1; x := x + 1 \mid\mid y := y + 1; y := y + 1 \{x = 2 \wedge y = 2\}$

Proven correct by proving correctness of following:

- proof outlines
  - $\{x = 0\}\ x := x + 1\ \{x = 1\}\ x := x + 1\ \{x = 2\}$
  - $\{y = 0\}\ y := y + 1\ \{y = 1\}\ y := y + 1\ \{y = 2\}$
- interference freedom 2 x 2 x 3 proof obligations!!
  - $\{x = i \wedge y = j\} y := y + 1\ \{x = i\}$ (for $i = 0, 1, 2$, $j = 0,1$)
  - $\{x = j \wedge y = i\} x := x + 1\ \{y = i\}$ (for $i = 0, 1, 2$, $j = 0,1$)

DRAWBACKS OWICKI-GRIES

- Number of proof obligations easily blows up
- Non-compositional
- Proof outlines need to be complete: annotations after each atomic step
- Sometimes weakening of annotations necessary to be able to prove interference freeness

EXAMPLE WEAKENING OF ASSERTIONS

How to prove correctness of

$\{x = 0\}\ x := x + 1 \mid\mid x := x + 2\ \{x = 3\}$

(assuming complete assignments are atomic)

Following proof outlines need to be proven

correct and free of interference

- $\{x = 0 \vee x = 2\}\ x := x + 1\ \{x = 1 \vee x = 3\}$
- $\{x = 0 \vee x = 1\}\ x := x + 2\ \{x = 2 \vee x = 3\}$

ALTERNATIVE APPROACH: WITH GHOST CODE

```
              {x = a + b & a == 0 & b == 0}
{x == a + b & a == 0}        || {x == a + b & b == 0}
<x := x + 1;>                || <x := x + 2;>
<a := 1;> // ghost           || <b :=2;> //ghost
{x == a + b & a == 1}        || {x == a + b & b == 2}
              {x == a + b & a == 1 & b == 2}
                        {x == 3}
```

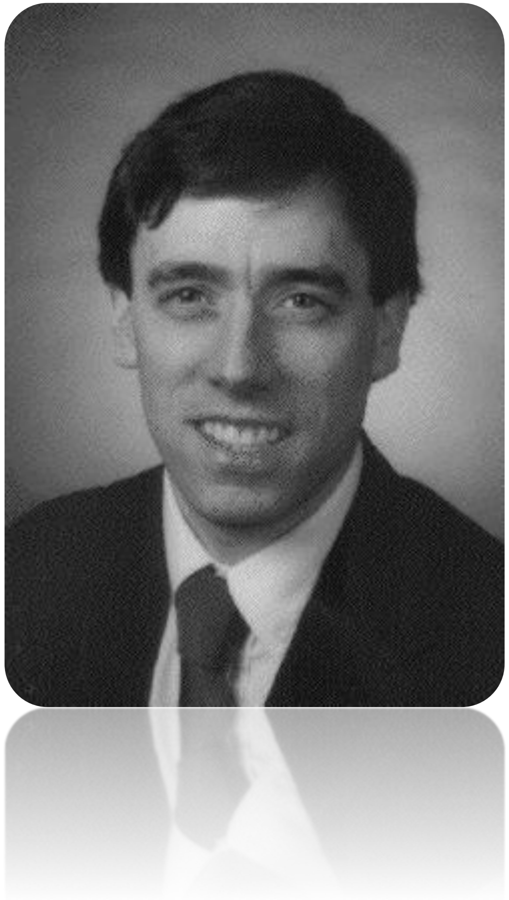
RELY-GUARANTEE METHOD

- Jones (1980)
- Compositional
- For each thread, specify
  - what it assumes from other threads
  - what it guarantees to other threads

Rely: what transitions may other threads make
Guarantee: what transitions may current thread make

$$\frac{\begin{array}{c} rely \vee guar1 \Rightarrow rely2 \\ rely \vee guar2 \Rightarrow rely1 \\ guar1 \vee guar2 \Rightarrow guar \\ \langle relyi, guari \rangle : \{Pi\}\ Si\ \{Qi\},\ i = 1,2 \end{array}}{\langle rely, guar \rangle : \{P\}\ S1 \parallel S2\ \{Q\}}$$

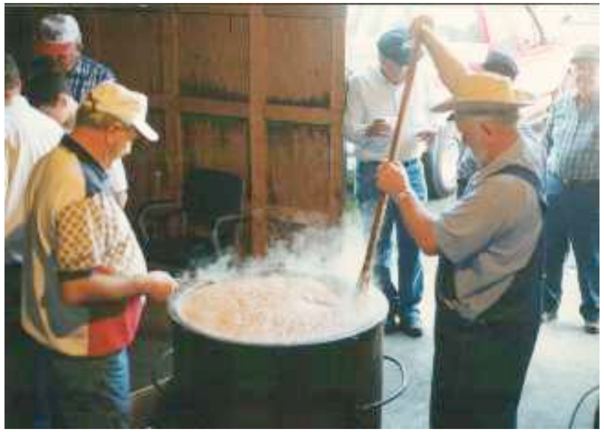
AVOIDING DATA RACES



John Boyland

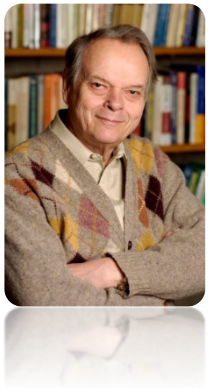
RECIPE FOR REASONING ABOUT JAVA

- Separation logic for sequential Java (Parkinson)
- Concurrent Separation Logic (O'Hearn)
- Permissions (Boyland)



Permission-based Separation Logic for Java

JOHN REYNOLDS'S 70TH BIRTHDAY PRESENT



$$\frac{\{P1\}S1\{Q1\} \quad \ldots\ldots\ldots\ldots \{Pn\}Sn\{Qn\}}{\{P1 \ast \ldots \ast Pn\}\ S1 \parallel \ldots \parallel Sn\ \{Q1 \ast \ldots \ast Qn\}}$$

where no variable free in *Pi* or *Qi* is changed in *Sj* (if $i \neq j$)

EXAMPLE

$$\dfrac{\dfrac{\vdots}{\{x = 0\}x := x + 1; x := x + 1\{x = 2\}} \qquad \dfrac{\vdots}{\{y = 0\}\ y := y + 1; y := y + 1\ \{y = 2\}}}{\{x = 0 \ast y = 0\}x := x + 1; x := x + 1 \parallel y := y + 1; y := y + 1\ \{x = 2 \ast y = 2\}}$$

No interference between the threads

WHY IS THIS NOT SUFFICIENT?

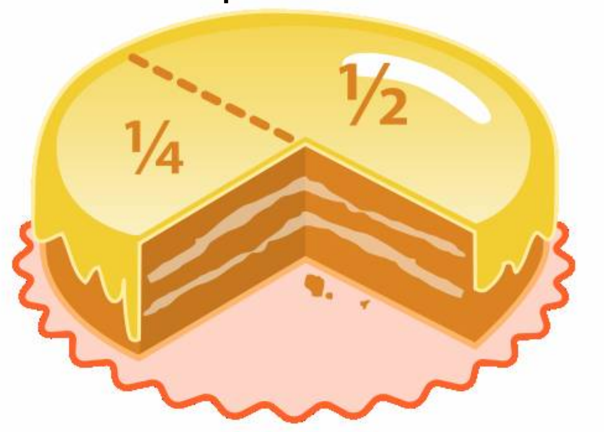
- Simultaneous reads not allowed

  1. Distinguish between read and write accesses

- Number of parallel threads is fixed

PERMISSIONS

- Permission to access a variable
- Value between 0 and 1
- Full permission 1 allows to change the variable
- Fractional permission in (0, 1) allows to inspect a variable
- Points-to predicate decorated with a permission
- Global invariant: for each variable, the sum of all the permissions in the system is never more than 1
- Permissions can be split and combined



PERMISSION-BASED SEPARATION LOGIC

Syntax extension of predicate logic:

$\varphi ::= e.f \xrightarrow{\pi} v \mid \varphi \ast \varphi \mid \varphi -\!\ast\ \varphi \mid \ldots$

Meaning:

- $e.f \xrightarrow{\pi} v$ – *e.f* contains value *v* and thread has access right π on *e.f*
- $\varphi 1 \ast \varphi 2$ – heap can be split in disjoint parts, satisfying φ1 and φ2, respectively
- $\varphi 1 -\!\ast\ \varphi 2$ – if heap extended with part that satisfies φ1, composition satisfies φ2

Notation:

$e.f \xrightarrow{\pi} v$	PointsTo($e.f$, π, v)
$\exists v.\ e.f \xrightarrow{\pi} v$	Perm($e.f$, π)

EXAMPLE

Permissions on n equally distributed over threads

$$\frac{\dfrac{\begin{array}{c}\vdots\\ \hline \{PointsTo(x,1,0) \ast Perm(n, ½)\}\\ x := x + n;\ x := x + n\\ \{PointsTo(x,1,2^*n) \ast Perm(n, ½)\}\end{array} \qquad \begin{array}{c}\vdots\\ \hline \{PointsTo(y,1,0) \ast Perm(n, ½)\}\\ y := y + n;\ y := y + n\\ \{PointsTo(y,1,2^*n) \ast Perm(n, ½)\}\end{array}}{}}{}$$

$$\begin{array}{c}\{PointsTo(x,1,0) \ast PointsTo(y,1,0) \ast Perm(n,1)\}\\ x := x + n;\ x := x + n \ \|\ y := y + n;\ y := y + n\\ \{PointsTo(x,1,2^*n) \ast PointsTo(y,1,2^*n) \ast Perm(n,1)\}\}\end{array}$$

Perm(x,1) = Perm(x, ½) ✳ Perm(x, ½)

Shared variable is only read
No interference between the threads

WHY IS THIS NOT SUFFICIENT?

- Simultaneous reads not allowed

1. Distinguish between read and write accesses

- Number of parallel threads is fixed

2. Dynamic thread creation

Thread specifications indicate how permissions should be distributed

EXAMPLE

```
class List {
      int val; List next;
       ...
}

class T {
      List y;
      void run() { ... }
}
```

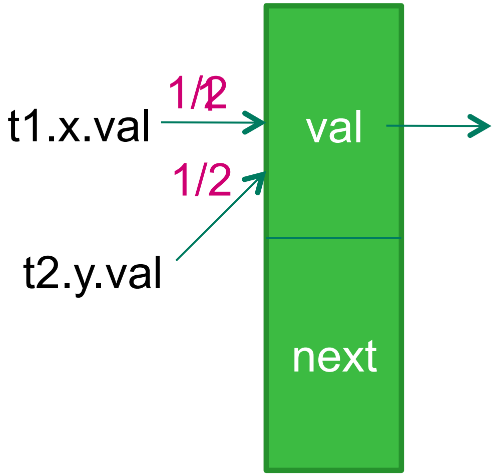
t1

```
x := new List;
x.val := ...;
t2 := new T;
t2.y := x;
fork t2;
read x.val;
...


join t2;
x.val := ...;
```

t2

```
run(){
...
read y.val
...
}
```



SPECIFICATION FOR RUN METHOD IN T2

requires y.val $\xrightarrow{1/2}$ _ ;
ensures y.val $\xrightarrow{1/2}$ _ ;
void run() {....}

- Forking thread has to give up required permissions
- Joining thread gains back ensured permissions

What happens if run is specified as follows:

requires y.val $\xrightarrow{1}$ _ ;
ensures y.val $\xrightarrow{1}$ _ ;
void run() {....}

EXAMPLE

```
class List {
      int val; List next;
       ...
}

class T {
      List y;
      void run() { ... }
}
```

t1

```
x := new List;
x.val := ...;
t2 := new T;
t2.y := x;
fork t2();
read x.val;
...
```

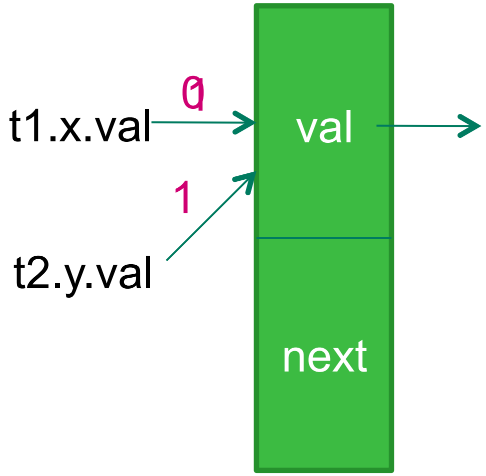
NOT ALLOWED!

```
join t2;
read x.val;
x.val := ...;
```

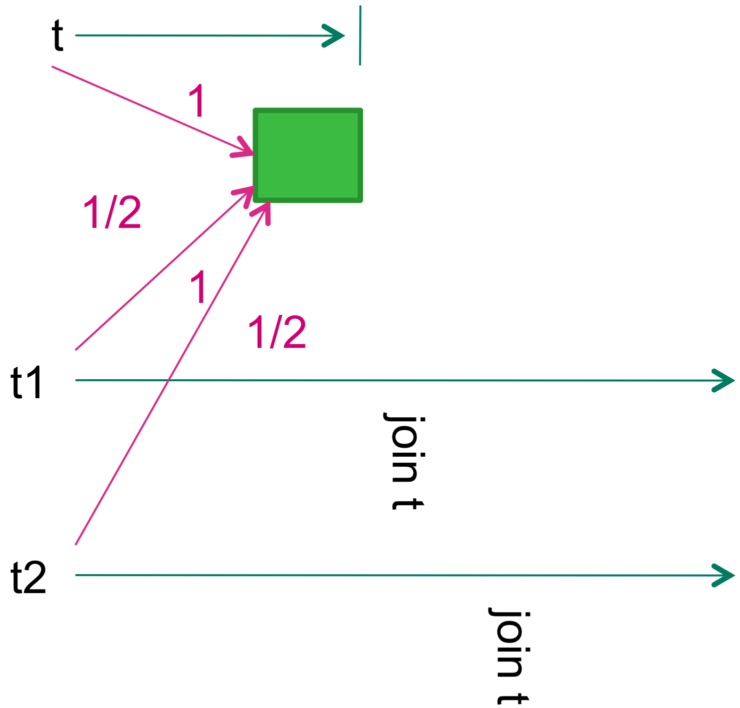
Now the permissions are back

t2

```
run(){
...
read y.val
...
}
```



THREAD TERMINATION



JOIN TOKEN

- Extension of property language Join(e, π)
- Permission to pick up fraction π after thread e has terminated
- Thread that creates thread t obtains Join-permission Join(t, 1)
- Join-permission treated as any other permission: can be transferred and split

  Join(e, π) $\ast\!\!-\!\!\ast$ Join(e, $\pi/2$) $\ast$ Join(e, $\pi/2$)

RULES FOR FORK AND JOIN

- Precondition fork = precondition run
  - Which permissions are transferred from creating to the newly created thread
- Postcondition run = postcondition join
  - Which permissions are released by the terminating thread, and can be reclaimed by another thread
  - Join only terminates when run has terminated
- Specification for run final, it can only be changed by extending definition of predicates preFork and postJoin

FORK, JOIN AND THREAD

```
class Thread {
    pred preFork = true;
    group postJoin<perm p> = true;


    requires preFork;
    ensures postJoin<1>;
    void run() {
        return null
    }
}
```

{t.preFork} fork t {join(t, 1)}

{join(t, π)} join t {t.postJoin(π)}

EXAMPLE: CLASS FIB

```
class Fib {
int number;

void init(n) {
    this.number := n;
    }

void run() {
    ..
    }
}
```

Leonardo di Pisa/
Fibonacci

FIB'S RUN METHOD

pred preFork = number $\xrightarrow{1}$ _;

group postJoin<perm p> = number $\xrightarrow{p}$ _;

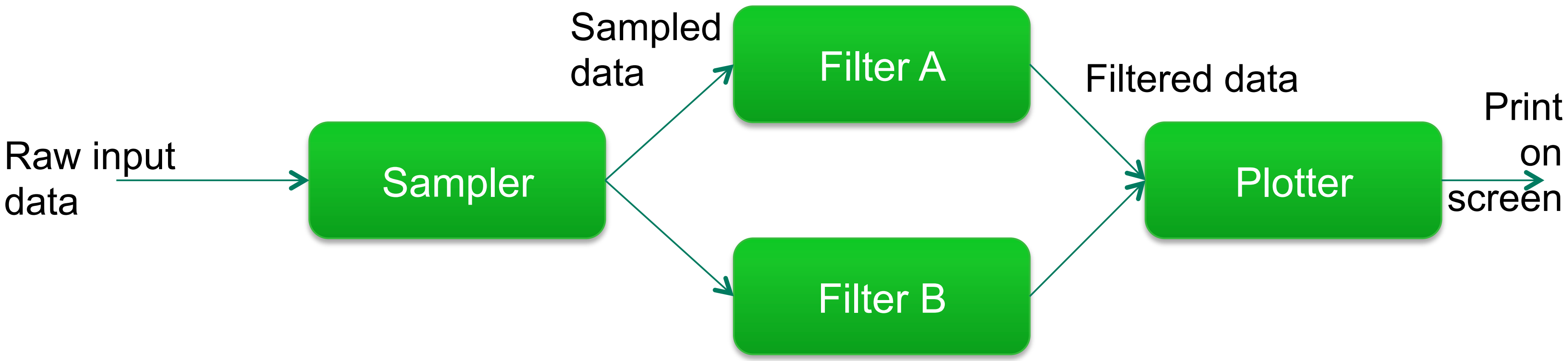
```
requires preFork;
ensures postJoin<1>;
void run() {
    if (! (this.number < 2))
    {  f1 = new Fib; f1.init(number -1);
       f2 = new Fib; f2.init(number - 2);
       fork f1; fork f2; join f1; join f2;
       this.number := f1.number + f2.number }
    else this.number := 1;
}
```

PROOF OUTLINE

pred preFork = number $\xrightarrow{1}$ _;
group postJoin<perm p> = number $\xrightarrow{p}$ _;

```
requires preFork;
void run() {
    if (! (this.number < 2))
    {   f1 = new Fib; f1.init(number -1); f2 = new Fib; f2.init(number - 2);
        {Perm(f1.number, 1) ✳ Perm(f2.number, 1) ✳ Perm(number, 1)}
        [fold preFork (2x)]
        {f1.preFork ✳ f2.preFork ✳ Perm(number, 1)}
        fork f1;
        {join(f1, 1) ✳ f2.preFork ✳ Perm(number, 1)}
        fork f2;
        {join(f1, 1) ✳ join(f2, 1) ✳ Perm(number, 1)}
        join f1; join f2;
        {f1.postJoin ✳ f2.postJoin ✳ Perm(number, 1)}
        [unfold postJoin (2x)]
        {Perm(f1.number, 1) ✳ Perm(f2.number, 1) ✳ Perm(number, 1)}
        this.number := f1.number + f2.number
        [close postJoin]
        {this.PostJoin}}
    else this.number := 1;
}
ensures postJoin(1);
```

MULTIPLE JOINS: PLOTTER



Filter A and Filter B both join Sampler

Plotter joins Filter A and Filter B

MAIN METHOD OF PLOTTER APPLICATION

```
requires … ensures …
void main(MVList lst) {
{S*A*B*P} [abbreviates preFork – joinToken for Sampler, Filter A/B,  Plotter]
Sampler<len> smp = new Sampler; smp.init(data); smp.fork();
{ Join(smp,1) * A * B * P }
AFilter<len> af = new AFilter; af.init(data, smp); af.fork();
{ Join(smp,1/2) * Join(af,1) * B * P }
BFilter<len> bf = new BFilter; bf.init(data, smp); bf.fork();
{ Join(af,1) * Join(bf,1) * P }
Plotter<len> plt = new Plotter; plt.init(data,af,bf); plt.fork();
{ Join(plt,1) }
plt.join();
{ plt.postJoin<1> }
} }
```

REASONING ABOUT LOCKS

Clément Hurlin

RESOURCE INVARIANT – CLASSICAL APPROACH

- Lock *x* acquired and released with lock *x* and unlock *x*
- Each lock has associated resource invariant
- Lock acquired ⟹ resource invariant lend to thread
- Lock released ⟹ resource invariant taken back from thread
- Class Object contains predicate

  pred inv = true;
- In rules: if *I* is resource invariant of *x*

  {true} lock *x* {*I*}

  {*I*}unlock *x*{true}
- This is sound only for single-entrant locks

{true}
lock *x*;
{*I*}
lock *x*;
{*I* ✳ *I*}
...

Resource *I* has been duplicated!

EXTRA PREDICATES

- Add extra predicates to logic
- $\varphi ::= e.f \xrightarrow{\pi} v \mid \varphi \ast \varphi \mid \varphi -\!\!\ast\ \varphi \mid$

  $\text{Lockset}(S) \mid S \text{ contains } e$
- Lockset (S) - S is the multiset of locks held by current thread
- S contains E - multiset S contains e

Multiset: set where you count the number of occurrences of each element
For multiset S: $x.x.S \neq x.S$

RULES FOR LOCKING

$$\frac{\{\mathrm{Lockset}(S) \ast \neg(S \text{ contains } u) \ast u.\mathrm{initialized}\}\ \ \mathrm{lock}\ u\ \ \{\mathrm{Lockset}(u.S) \ast u.\mathrm{inv}\}}{\{\mathrm{Lockset}(u.S)\}\mathrm{lock}\ u\{\mathrm{Lockset}(u.u.S)\}}$$

Will be explained

RULES FOR UNLOCKING

$$\frac{}{\{\text{Lockset}(u.S) \ast u.\text{inv}\}\text{unlock } u\{\text{Lockset}(S)\}}$$

$$\frac{}{\{\text{Lockset}(u.u.S)\}\text{unlock } u\{\text{Lockset}(u.S)\}}$$

$e.unlocked(e') = \text{Lockset}(e') \ast \neg\ (e' \text{ contains } e)$

EXAMPLE

class Account {

int balance;

pred inv = this.balance $\xrightarrow{1}$ _ ;

requires initialized $\ast$ unlocked(S); ensures Lockset(S);

void deposit(int x) {

{initialized $\ast$ unlocked(S)}

lock this;

{Lockset($S \cdot$ this) $\ast$ inv}

this.balance := this.balance + x;

{Lockset($S \cdot$ this) $\ast$ inv}

unlock this;

{Lockset(S)}

}

open and close of predicate

NEW THREADS HAVE EMPTY LOCKSET

Specification for method run becomes:

```
requires preFork ✳ Lockset(nil);
ensures postJoin<1>;
    method run() {
        return null;
    }
```

Empty multiset

SPECIFICATIONS FOR WAIT AND NOTIFY

requires Lockset(S) $\ast$ S contains this $\ast$ inv;
ensures Lockset(S) $\ast$ inv;
void wait();

requires Lockset(S) $\ast$ S contains this;
ensures Lockset(S);
void notify();

LOCK INITIALISATION

- Locks created dynamically
- Initialisation of resource invariant necessary
- Object can only be used as lock when its resource invariant has been initialised
- Special annotation command commit to mark that resource invariant is initialised
- Default position for commit: end of constructor

LOCK INITIALISATION EXAMPLE

- Class ThreadPool contains Vector v to store threads
- Construction of ThreadPool gives Perm(v, 1)
- Resource invariant inv = Perm(v, 1)
- Constructor body:
  - Initialise v to empty Vector
  - commit: Perm(v, 1) stored inside lock
- Now lock on ThreadPool can be acquired and released by threads, to add and remove threads to threadpool
- Only when thread has lock on ThreadPool, does it have permission to access *v*

EXTRA PREDICATES

- Add extra predicates to logic
- $\varphi ::= e.f \xrightarrow{\pi} v \mid \varphi \ast \varphi \mid \varphi -\!\!\ast\ \varphi \mid$

  $\text{Lockset}(S) \mid S \text{ contains } e \mid e.\text{fresh} \mid e.\text{initialized}$
- Lockset (*S*) - *S* is the multiset of locks held by current thread
- *S* contains *e* - multiset *S* contains *e*
- *e*.fresh - *e*'s resource invariant not yet initialized
- *e*.initialized - *e*'s resource invariant initialized

Some of these atomic propositions can be freely duplicated, some cannot

COPYABLE VERSUS NON-COPYABLE

- Copyable properties: persistent state properties

  Once established, they hold forever
- Non-copyable properties: transient state properties

  Properties that hold temporarily
- Axiom for copyable properties (to use in proofs):

  $(G \wedge F) -\!\!\ast (G \ast F)$
- This implies

  $F -\!\!\ast (F \ast F)$

  i.e., formula can be duplicated freely

COPYABLE VERSUS NON-COPYABLE

Lockset (*S*)	-	non-copyable
S contains *e*	-	copyable
e.fresh	-	non-copyable
e.initialized	-	copyable

RULES FOR LOCK CREATION AND COMMIT

Now we can formulate the rules for new and commit

$$\frac{}{\{\text{true}\}\ v := \text{new}\ C\ \{\exists X.v \to X \ast X.f1 \to \text{null} \ast \ldots \ast X.fn \to \ast\ v.\text{fresh}\}}$$

$$\frac{}{\{\text{Lockset}(S) \ast u.\text{inv} \ast u.\text{fresh}\}\ \text{commit}\ u\ \{\text{Lockset}(S) \ast \neg(S\ \text{contains}\ u) \ast u.\text{initialized}\}}$$

FUNCTIONAL VERIFICATION OF CONCURRENT PROGRAMS

WORK IN PROGRESS

Marina Zaharieva – Stojanovski

EXAMPLE: PARALLEL INCREASE

How to prove:

Ghost code solution:

```
               {x = a + b & a == 0 & b == 0}
{x == a + b & a == 0}          || {x == a + b & b == 0}
<x := x + 1;>                  || <x := x + 1;>
<a := 1;> // ghost             || <b :=1;> //ghost
{x == a + b & a == 1}          || {x == a + b & b == 1}
               {x == a + b & a == 1 & b == 1}
                          {x == 2}
```

Problem:

```
{x == 0}
< x := x + 1;>
{x == 1}
```

Our approach:
Maintain abstract history of updates

unstable: assertions can be made invalid by other threads

AS A JAVA-LIKE PROGRAM

```
class Counter{
    int data;
    Lock l;
    resource_inv = exists v. PointsTo(data, 1, v);

requires true;
ensures  true;
void increase(){
    l.lock();         // obtain PointsTo(data, 1, v);
        data ++;
    l.unlock();       // loose PointsTo(data, 1, v + 1);
    // now we don't know anything about data anymore
  }
}
```

Client:

```
c = new Counter(0);
fork t1;   //t1 calls c.increase();
fork t2;   //t2 calls c.increase();
join t1;
join t2;

// Is  c.data  == 2 ?
```

Permission to read and update data

Needed:
A specification of increase that records the update

SEPARATE PERMISSION AND VALUE

[Separation Rule]
PointsTo(x, 1, v) *-* Perm(x, 1) * Init(x, {v}) * Hist(x, 1, {});

- Perm(x,1) - permission to access x
- Init(x, {v}) - initial value of x
- Hist(x, 1, H) - history of all updates/actions to x

A HISTORY OF ACTIONS

History H is process algebra term composed of user-defined actions (use ACP)

Examples

action **a**<int x>(int k) = \old(x) + k;

action **b**<list l>(int e) = cons(\old(l), e);

action **c**<int k>(int w) = w;

COUNTER SPECIFICATION

```
class Counter{
   int data;
   Lock l;
    //resource_inv = Perm(data, 1);

    //action a<int x> () = \old(x) + 1;

requires Hist(data, p, H);
ensures Hist(data, p, H.a);
void increase(){
    l.lock();  /* start a */ data ++; /* record a */ l.unlock();
    }
}
```

Record LOCAL changes in the history

HISTORY MANIPULATION

[SplitHist Rule]

Hist(x, p, H) *-* Hist(x, p/2, H_1) * Hist(x, p/2, H_2);

$H = H_1 \parallel H_2$

- Forking a thread: mark with special synchronisation action (s, $\overline{s}$)
  $H = H.s \parallel \overline{s}$
  - Current thread: H.s
  - New thread: $\overline{s}$
- Joining a thread: continue with the parallel composition of the local histories

CLIENT-SIDE REASONING

To reason about the value in data**,** we need:

- Init(data, V) predicate
- Full Hist(data, **1,** H) token

After the client of the Counter joins both threads, we can

reinitialize the History:

$$\mathrm{Init}(data, \{0\}) * \mathrm{Hist}(data, 1, s_{t1}.\, s_{t2} \,||\, \overline{s}_{t1}.\, a() \,||\, \overline{s}_{t2}.\, a()) \; {*}\!\!-\!\!{*}$$

$$\mathrm{Init}(data, \{2\}) * \mathrm{Hist}(data, 1, \{\})$$

The only possible value for data is 2

NON-DETERMINISTIC BEHAVIOUR

```
class Counter{
//action a<int x> (int n) = \old(x) + n;
//action b<int x> (int n) = \old(x) * n;
requires Hist(data, p, H);
ensures  Hist(data, p, H.a(n));
void increase(int n){
   l.lock(); data = data + n; l.unlock();
  }
requires Hist(data, p, H);
ensures  Hist(data, p, H.b(m));
void increase(int m){
   l.lock(); data = data * m; l.unlock();
  }
}
```

Client:

```
c = new Counter(0);
fork t1;   //t1: c.increase(4);
fork t2;   //t2: c.multiply(4);
join t1;
join t2;

// What is  c.data?
```

COMPUTING POSSIBLE VALUES

Reinitialisation of the History:

$$\text{Init(data, \{0\}) * Hist(data, 1, } s_{t1}.\, s_{t2} \,\|\, s_{t1}.\, a(4) \,\|\, s_{t2}.\, b(4)) \text{ *-*}$$

$$\text{Init(data, \{0\}) * Hist(data, 1, a(4).b(4) + b(4).a(4)) *-*}$$

$$\text{Init(data, \{4,16\}) * Hist(data, 1, \{\})}$$

Extensions

- Histories for multiple variables
- Data structures

CLASS INVARIANTS IN CONCURRENT SETTING

- Class invariant: property about reachable object state
- Typical: relation between object's fields
- In sequential setting: breaking allowed within method boundaries
- In concurrent setting: breaking allowed when violation cannot be observed
- Explicit pack and unpack operations

$$\{holds(v.I, 1)\}\ unpack(v.I)\{unpacked(v.I, 1) * v.I\}$$

$$\{v \neq null * PointsTo(v.f,1,w) * \backslash forall{*}\ (I \in inv(V), v.f \in fp(v.I).\ unpacked(v.I,\pi))\}$$
$$v.f = w;$$
$$\{PointsTo(v.f,1,w) * \backslash forall{*}\ (I \in inv(V), v.f \in fp(v.I).\ unpacked(v.I,\pi))\}$$

$$\{unpacked(v.I, 1) * v.I\}\ pack(v.I)\ \{holds(v.I, 1)\}$$

Usage:

$$\frac{\{holds(v.I, \pi) * v.I\}\ c\ \{F\}}{\{holds(v.I, \pi)\}\ c\ \{F\}}$$

REASONING ABOUT GPU PROGRAMS

GPU KERNELS

- Originally for graphics
- More and more used also for other applications
- Single-Instruction-Multiple-Thread model (similar to Vector machines)

- Host (typically CPU) invokes kernel on separate device
- Kernel:
  - Many threads
  - Execute all same code
  - But on different data
- OpenCL: extended subset of C, platform-independent

VECTOR ADDITION AS OPENCL KERNEL

```
_kernel void square( _global float* input,
                     _global float* output) {
   int i = get_global_id(0);
   output[i] = input[i] * input[i];
}
```

__global
Where are the arrays stored

SYNCHRONISATION WITHIN A KERNEL

- Barrier: all threads block until all threads have reached (the same) barrier
- This is the only moment where you can make an assumption about the state of another thread
- Main problem: barrier divergence
- Example of possible barrier divergence:

```
if b
  BARRIER(…);
else
  BARRIER(…);
```

Only okay if
all threads satisfy
b or not b

BARRIER EXAMPLE

```
_kernel void square( _global float* input, _global float* output) {
    int i = get_global_id(0);
    output[i] = input[i] * input[i];
    barrier(CLK_GLOBAL_MEM_FENCE);
    output[(i+1)%wg_size]=output[(i+1)%wg_size] * input[i];
}
```

REASONING ABOUT KERNELS

- What happens when host invokes a kernel?
- Relation between kernel and thread
- What happens at the barrier?

- What should be specified?
- What should be verified?

EXAMPLE SPECIFICATION

```
_kernel void square( _global float* input,
                     _global float* output) {
   int i = get_global_id(0);
   output[i] = input[i] * input[i];
   barrier(CLK_GLOBAL_MEM_FENCE);
   output[(i+1)%wg_size]=
          output[(i+1)%wg_size] * input[i];
}
```

Provided by host

Kernel Specification:

- Global Memory Resources:
  - Write permission on all entries of output
  - Read permission on all entries of input
- Shared Memory Resources: -

Thread Specification:

- Resources:
  - Perm(output[i], 1) ★ Perm(input[i], π)
- Precondition: -
- Postcondition:
  - output[(i + 1) % wg_size] = input[i] * input[(i + 1) % wg_size]^2

Global proof obligation:
All threads together use no more resources than available in the kernel

EXAMPLE
BARRIER SPECIFICATION

```
_kernel void square( _global float* input,
                     _global float* output) {
   int i = get_global_id(0);
   output[i] = input[i] * input[i];
   barrier(CLK_GLOBAL_MEM_FENCE);
   output[(i+1)%wg_size]=
           output[(i+1)%wg_size] * input[i];
}
```

Kernel Specification:

- Global Memory Resources:
  - Write permission on all entries of output
  - Read permission on all entries of output
- Shared Memory Resources: -

Global proof obligation:
All permissions available in kernel

Barrier Specification:

- Resources:
  - Exchange write permission on output [i] for
    write permission on output[(i+1) % wg_size]
  - Keep read permission on input[i]
- Precondition: output[i] = input[i] * input[i]
- Postcondition: output[(i + 1)%wg_size] = input[(i + 1)%wg_size]^2

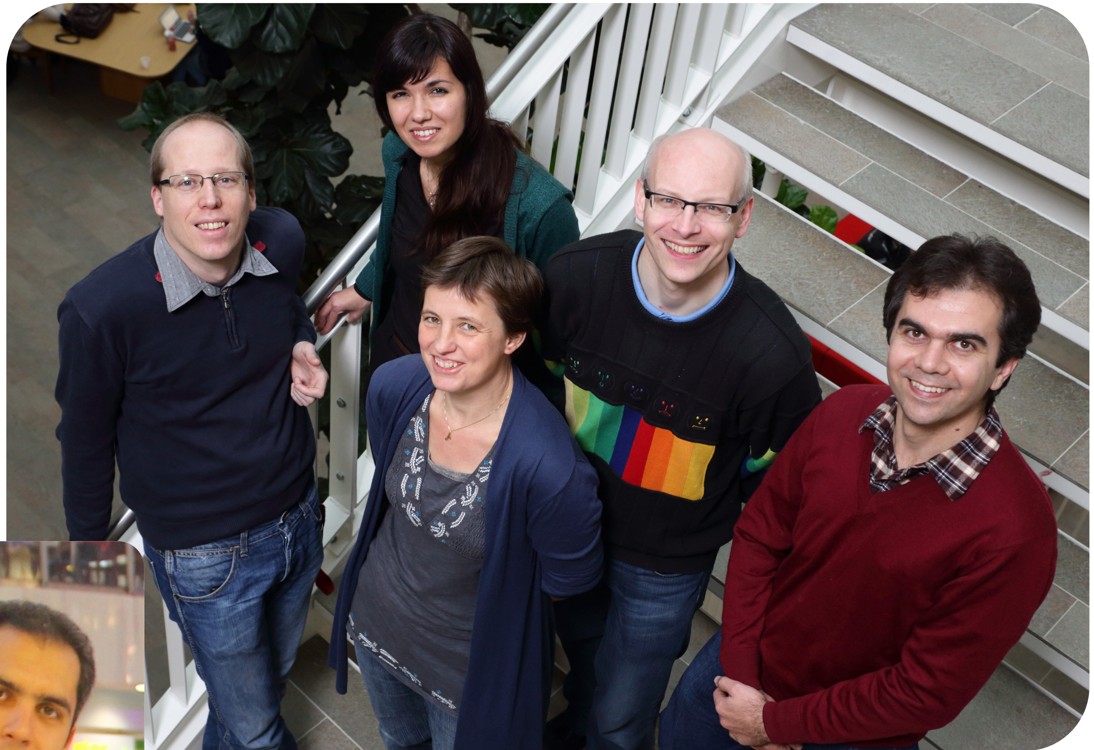
Global proof obligation:
Barriers correctly transfer knowledge about state

PROOF OBLIGATIONS

- Threads respects their thread specification
- Kernel resources are sufficient to provide each thread necessary global resources
- Local resources are properly distributed over threads
- Kernel precondition implies universal quantification of thread precondition
- Barriers only redistribute permissions that are in the kernel
- Universal quantification of barrier precondition implies universal quantification of barrier postcondition
- Universal quantification of thread postcondition implies kernel postcondition

Extra layer:
workinggroup specifications

ACKNOWLEDGEMENTS



Saeed Darabi, Wojciech Mostowski,
Marina Zaharieva-Stojanovski,
Stefan Blom, Afshin Amighi

SUMMARY

- Software quality remains a challenge
- Classical Hoare logic-based techniques are becoming more and more powerful
- Next challenge: verification of concurrent software
- Separation logic and permissions
- Permission transfer whenever threads synchronise
- Verification of functional properties
- Also applicable to other concurrent programming paradigms

More information?
Want to try it out yourself?
Go to: http://www.utwente.nl/vercors