

Soutenance en vue de l'obtention de l'Habilitation à Diriger des Recherches

16 janvier 2014

Reasoning between programming languages and architectures

Francesco Zappa Nardelli

<http://www.di.ens.fr/~zappa/readings/hdr>



A scenic mountain landscape with a white rectangular overlay containing text. The background shows a vast mountain range under a blue sky with scattered white clouds. The foreground features a rocky, brownish mountain slope. A white rectangular box is centered on the page, containing the text "Illustrate my approach to research" in a black, serif font.

Illustrate my approach to research

III Explain one research project

III Explain Show some code project arch

1. A Better World



Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```


Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```


Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 1 returns without modifying b

Thread 2

```
b = 42;  
printf("%d\n", b);
```


Shared memory

```
int a = 1;
```

```
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Thread 1 returns without modifying `b`

Thread 2 is not affected by Thread 1 and vice-versa
(this program is *data-race free*)

Shared memory

```
int a = 1;
```

```
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Thread 1 returns without modifying `b`

Thread 2 is not affected by Thread 1 and vice-versa
(this program is *data-race free*)

We expect this program to print 42.

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b<=26; ++b)  
        ;  
}
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```



...sometimes we get 0 on the screen


```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

gcc 4.7 -O2



```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

```
movl    a(%rip), %edx    # load a into edx  
movl    b(%rip), %eax    # load b into eax  
testl   %edx, %edx      # if a!=0  
jne     .L2              # jump to .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)    # store eax into b  
xorl    %eax, %eax      # store 0 into eax  
ret                                           # return
```

gcc 4.7 -O2



```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

The outer loop can be (and is) optimised away

```
movl    a(%rip), %edx    # load a into edx  
movl    b(%rip), %eax    # load b into eax  
testl   %edx, %edx      # if a!=0  
jne     .L2              # jump to .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)    # store eax into b  
xorl    %eax, %eax      # store 0 into eax  
ret                                           # return
```

gcc 4.7 -O2



```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

```
movl    a(%rip), %edx    # load a into edx  
movl    b(%rip), %eax    # load b into eax  
testl   %edx, %edx      # if a!=0  
jne     .L2              # jump to .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)    # store eax into b  
xorl    %eax, %eax      # store 0 into eax  
ret                                           # return
```

gcc 4.7 -O2



```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

```
movl    a(%rip), %edx    # load a into edx  
movl    b(%rip), %eax    # load b into eax  
testl   %edx, %edx      # if a!=0  
jne     .L2              # jump to .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)    # store eax into b  
xorl    %eax, %eax      # store 0 into eax  
ret                                           # return
```


gcc 4.7 -O2



```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

```
movl    a(%rip), %edx    # load a into edx  
movl    b(%rip), %eax    # load b into eax  
testl   %edx, %edx      # if a!=0  
jne     .L2              # jump to .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)    # store eax into b  
xorl    %eax, %eax      # store 0 into eax  
ret                                           # return
```

gcc 4.7 -O2



```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

```
movl    a(%rip), %edx    # load a into edx  
movl    b(%rip), %eax    # load b into eax  
testl   %edx, %edx      # if a!=0  
jne     .L2              # jump to .L2  
movl    $0, b(%rip)  
ret
```

.L2:

```
movl    %eax, b(%rip)    # store eax into b  
xorl    %eax, %eax      # store 0 into eax  
ret
```

gcc 4.7 -O2




```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

```
movl    a(%rip), %edx    # load a into edx  
movl    b(%rip), %eax    # load b into eax  
testl   %edx, %edx      # if a!=0  
jne     .L2              # jump to .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)    # store eax into b  
xorl    %eax, %eax      # store 0 into eax  
ret     # return
```

The compiled code saves and restores b

Correct in a sequential setting, but...



```
movl  a(%rip), %edx    # load a into edx
movl  b(%rip), %eax    # load b into eax
testl %edx, %edx      # if a!=0
jne   .L2             # jump to .L2
movl  $0, b(%rip)
ret
.L2:
movl  %eax, b(%rip)   # store eax into b
xorl  %eax, %eax      # store 0 into eax
ret                    # return
```

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
movl    a(%rip),%edx  
movl    b(%rip),%eax  
testl   %edx, %edx  
jne     .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)  
xorl    %eax, %eax  
ret
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```


Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
movl a(%rip),%edx  
movl b(%rip),%eax  
testl %edx, %edx  
jne .L2  
movl $0, b(%rip)  
ret  
.L2:  
movl %eax, b(%rip)  
xorl %eax, %eax  
ret
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

- Read a (1) into edx

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
movl    a(%rip),%edx  
movl    b(%rip),%eax  
testl   %edx, %edx  
jne     .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)  
xorl    %eax, %eax  
ret
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

- Read **a** (1) into **edx**
- Read **b** (0) into **eax**

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
movl    a(%rip),%edx  
movl    b(%rip),%eax  
testl   %edx, %edx  
jne     .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)  
xorl    %eax, %eax  
ret
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

- Read **a** (1) into **edx**
- Read **b** (0) into **eax**
- Store 42 into **b**

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
movl    a(%rip),%edx  
movl    b(%rip),%eax  
testl   %edx, %edx  
jne     .L2  
movl    $0, b(%rip)  
ret
```

.L2:

```
movl    %eax, b(%rip)  
xorl    %eax, %eax  
ret
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

- Read **a** (1) into **edx**
- Read **b** (0) into **eax**
- Store 42 into **b**
- Store **eax** (0) into **b**

Shared memory

```
int a = 1;  
int b = 0;
```

Thread 1

```
movl    a(%rip),%edx  
movl    b(%rip),%eax  
testl   %edx, %edx  
jne     .L2  
movl    $0, b(%rip)  
ret  
.L2:  
movl    %eax, b(%rip)  
xorl    %eax, %eax  
ret
```

Thread 2

```
b = 42;  
printf("%d\n", b);
```

- Read **a** (1) into **edx**
- Read **b** (0) into **eax**
- Store 42 into **b**
- Store **eax** (0) into **b**
- Print **b**: 0 is printed

Shared memory

```
int a = 1;  
int b = 0;
```

...gives unexpected results
in some concurrent contexts!

```
movl    %eax, b(%rip)  
xorl    %eax, %eax  
ret
```

- Read **b** (0) into **eax**
- Store 42 into **b**
- Store **eax** (0) into **b**
- Print **b**: 0 is printed



ORIGINAL
NEVER

What? Can our program print 0?



What? Can our program print 0?

No, C11 states that printing 42
is the only correct output

This is a *compiler bug*

What? Can our program print 0?

No, C11 states that printing 42
is the only correct output

This is a *concurrency compiler bug*



We reported it
it was promptly fixed

World is a better place



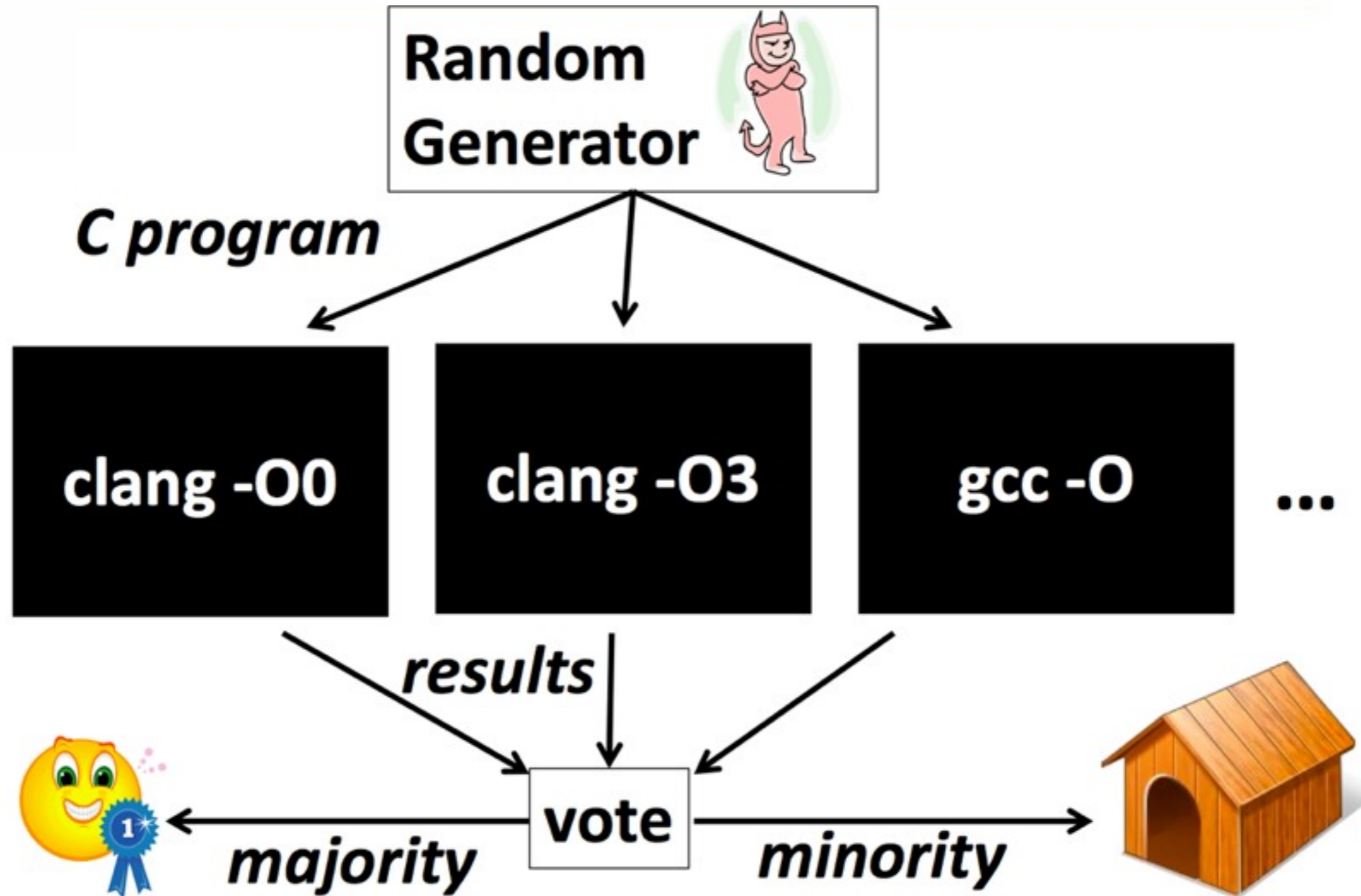


World is a better place

Can we catch more similar bugs
and make the world even better?

Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

Random



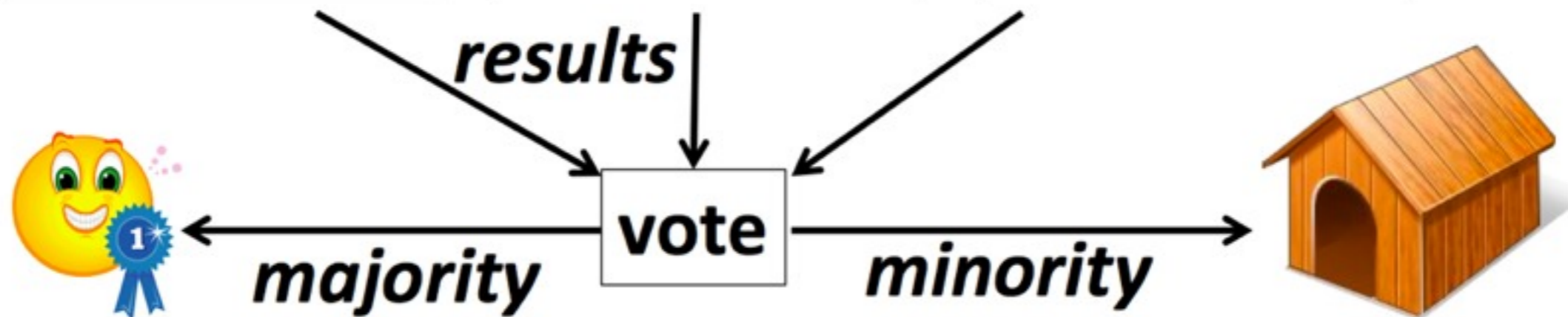
Reported hundreds of bugs
on various versions of gcc, clang and other compilers

clang -O0

clang -O3

gcc -O

...



Compiler testing: state of the art

Yang, Chen, Eide, Regehr - PLDI 2011

Random



Reported hundreds of bugs

*Cannot catch
concurrency compiler bugs*



majority

vote

minority



Hunting concurrency compiler bugs?

How to deal with non-determinism?

How to generate non-racy interesting programs?

How to capture all the behaviours of concurrent programs?

A compiler can optimise away behaviours:

how to test for correctness?

limit case: two compilers generate correct code with disjoint final states

Idea

C/C++ compilers support separate compilation
Functions can be called in arbitrary non-racy concurrent contexts



C/C++ compilers can only apply transformations sound
with respect to an arbitrary non-racy concurrent context

Hunt concurrency compiler bugs

=

search for transformations of sequential code
not sound in an arbitrary non-racy context

**Random
Generator**



**SEQUENTIAL
PROGRAM**

*optimising
compiler
under test*



EXECUTABLE



tracing

**MEMORY
TRACE**

reference
semantics



**REFERENCE
MEMORY
TRACE**



**Check: only transformations sound
in any concurrent non-racy context**

2. Soundness of compiler optimisations in the C11/C++11 memory model



World of optimisations

gcc 4.8.1 with -O2 option goes through 163 compilation passes

computed using `-fdump-tree-all` and `-fdump-rtl-all`

Sun Hotspot Server JVM has 18 high-level passes

each pass composed of one or more smaller passes

<http://www.azulsystems.com/blog/cliff-click/2009-04-14-odds-ends>

Example: loop invariant code motion

Compiler Writer



Semanticist



Example: loop invariant code motion

Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

Semanticist



Example: loop invariant code motion

Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

Semanticist



```
for (int i=0; i<2; i++) {  
    z = i;  
    x[i] += y+1 ;  
}
```

Example: loop invariant code motion

Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

Semanticist



```
tmp = y+1 ;  
for (int i=0; i<2; i++) {  
    z = i;  
    x[i] += tmp ;  
}
```

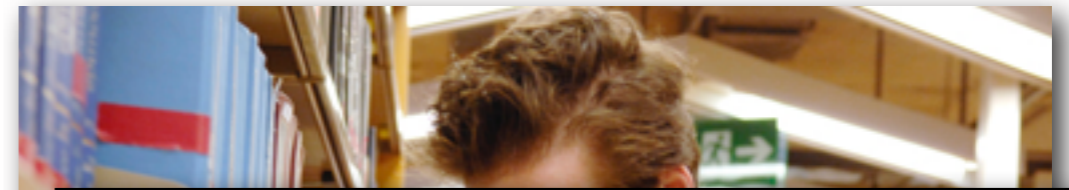
Example: loop invariant code motion

Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

Semanticist



Elimination of run-time events
Reordering of run-time events
Introduction of run-time events
Operations on sets of events

```
tmp = y+1 ;  
for (int i=0; i<2; i++) {  
    z = i;  
    x[i] += tmp ;  
}
```


Example: loop invariant code motion

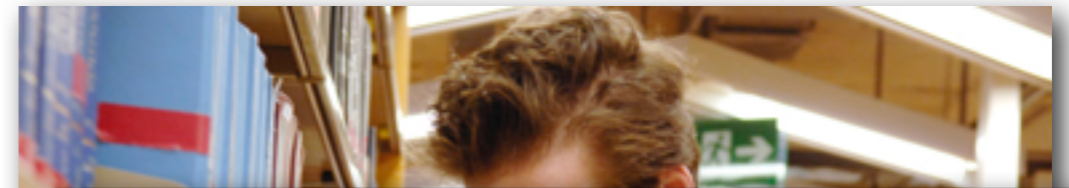
Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

```
tmp = y+1 ;  
for (int i=0; i<2; i++) {  
    z = i;  
    x[i] += tmp ;  
}
```

Semanticist



Elimination of run-time events
Reordering of run-time events
Introduction of run-time events
Operations on sets of events

```
Store z 0  
Load y 42  
Store x[0] 43  
Store z 1  
Load y 42  
Store x[1] 43
```

Example: loop invariant code motion

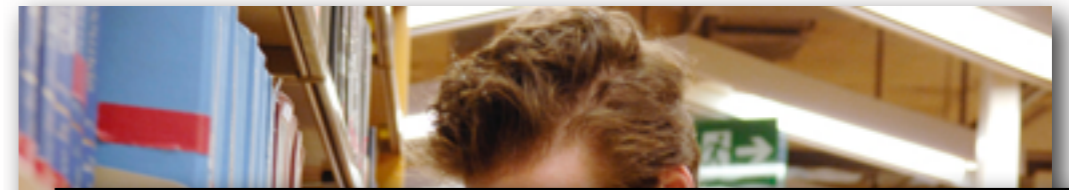
Compiler Writer



Sophisticated program analyses
Fancy algorithms
Source code or IR
Operations on AST

```
tmp = y+1 ;  
for (int i=0; i<2; i++) {  
    z = i;  
    x[i] += tmp ;  
}
```

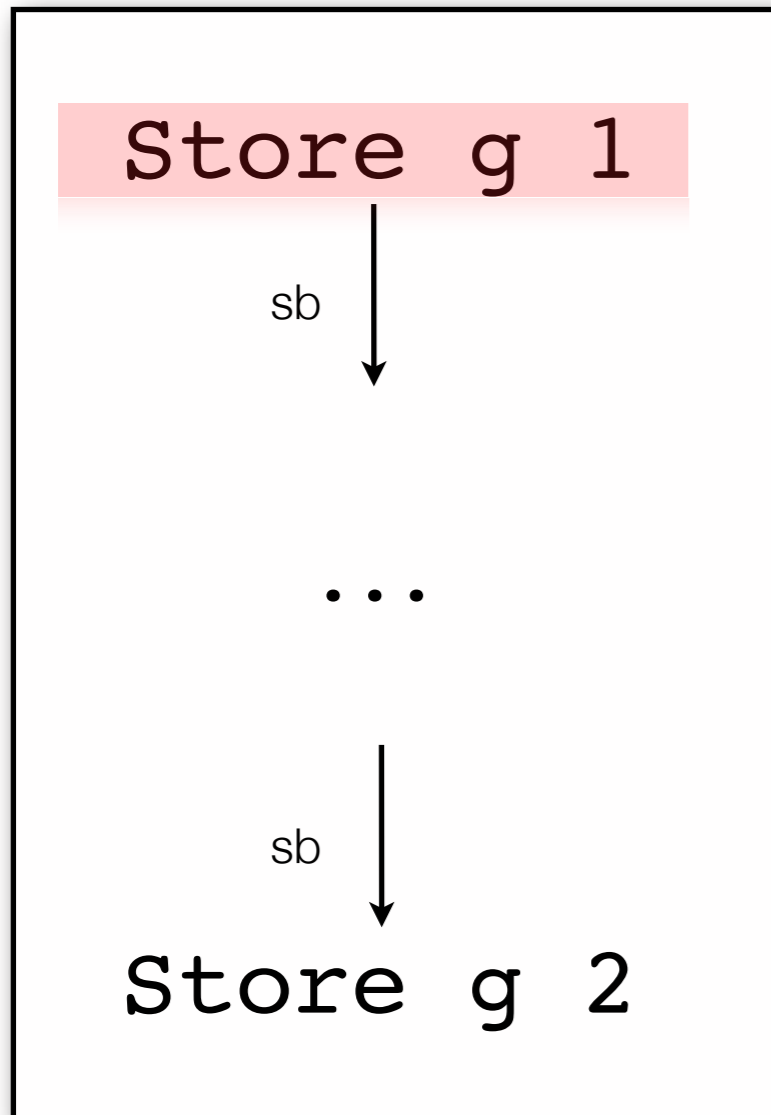
Semanticist



Elimination of run-time events
Reordering of run-time events
Introduction of run-time events
Operations on sets of events

```
Load y 42  
Store z 0  
  
Store x[0] 43  
Store z 1  
  
Store x[1] 43
```

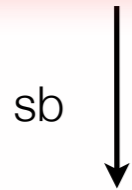
Elimination of *overwritten writes*



Under which conditions is it correct to eliminate the first store?

Elimination of *overwritten writes*

Store g 1



Under which conditions is it correct to eliminate the first store?

What is the semantics of concurrent C11/C++11 code?

The C11/C++11 memory model

C11/C++11 are based on the DRF approach:

- racy code is undefined
- race-free code must exhibit only sequentially consistent behaviours
- main synchronisation mechanism: lock/unlock

The C11/C++11 memory model

C11/C++11 are based on the DRF approach:

- racy code is undefined
- race-free code must exhibit only sequentially consistent behaviours
- main synchronisation mechanism: lock/unlock

Our first example is data-race free.

Printing 0 is disallowed by the standard.

The C11/C++11 memory model

C11/C++11 are based on the DRF approach:

- racy code is undefined
- race-free code must exhibit only sequentially consistent behaviours
- main synchronisation mechanism: lock/unlock

Escape mechanism for experts, *low-level atomics*:

- races allowed
- attributes on accesses specify their semantics:

MO_SEQ_CST

MO_RELEASE/MO_ACQUIRE

MO_RELAXED

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;  
f.store(1, MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE) == 0);  
printf ("%d", g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;  
f.store(1,MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE)==0);  
printf ("%d",g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;  
f.store(1, MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE) == 0);  
printf ("%d", g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;  
f.store(1, MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE) == 0);  
printf ("%d", g)
```


MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;  
f.store(1, MO_RELEASE);
```

sync



Thread 2

```
while (f.load(MO_ACQUIRE) == 0);  
printf ("%d", g)
```

MO_RELEASE / MO_ACQUIRE

```
g = 0; atomic f = 0;
```

Thread 1

```
g = 42;  
f.store(1, MO_RELEASE);
```

Thread 2

```
while (f.load(MO_ACQUIRE) == 0);  
printf ("%d", g)
```

The release/acquire synchronisation guarantees that:

- the program is DRF
- 42 is printed at the end of the execution

Remark: unlock \approx release, lock \approx acquire.

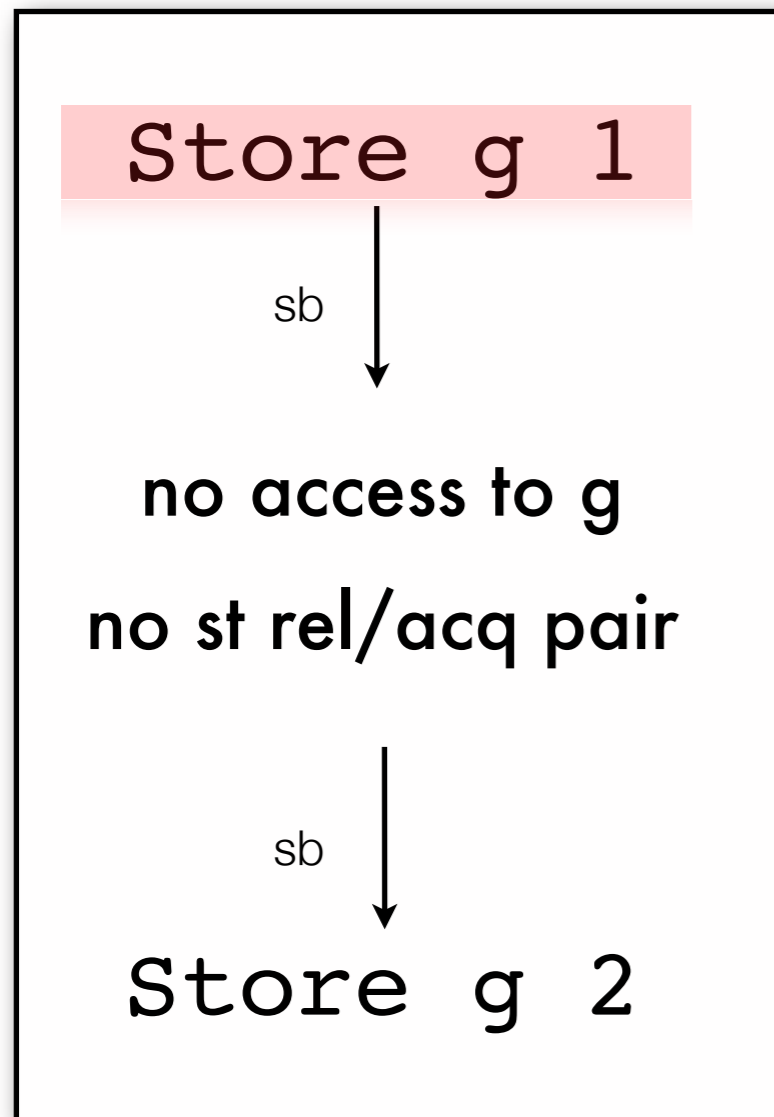
Same-thread release/acquire pairs

A same-thread release-acquire pair is a pair of a release action followed by an acquire action in program order.

An action is a *release* if it is a possible source of a synchronisation
unlock mutex, release or seq_cst atomic write

An action is an *acquire* if it is a possible target of a synchronisation
lock mutex, acquire or seq_cst atomic read

Elimination of *overwritten writes*



It is safe to eliminate the first store if there are:

1. no intervening accesses to **g**
2. no intervening same-thread release-acquire pairs

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;  
f1.store(1, RELEASE);  
while(f2.load(ACQUIRE) == 0);  
g = 2;
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

candidate overwritten write

```
g = 1;  
f1.store(1, RELEASE);  
while(f2.load(ACQUIRE) == 0);  
g = 2;
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;
```

```
f1.store(1, RELEASE);
```

```
while(f2.load(ACQUIRE) == 0);
```

```
g = 2;
```

candidate overwritten write



same-thread release-acquire pair



The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;  
f1.store(1, RELEASE);  
while(f2.load(ACQUIRE) == 0);  
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE) == 0);  
printf("%d", g);  
f2.store(1, RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;  
f1.store(1, RELEASE);  
while(f2.load(ACQUIRE) == 0);  
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE) == 0);  
printf("%d", g);  
f2.store(1, RELEASE);
```

Thread 2 is non-racy

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;  
f1.store(1, RELEASE);  
while(f2.load(ACQUIRE) == 0);  
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE) == 0);  
printf("%d", g);  
f2.store(1, RELEASE);
```

Thread 2 is non-racy
The program should only print **1**

The soundness condition

Shared memory

`g = 0; atomic f1 = f2 = 0;`

Thread 1

`g = 1;`

`f1.store(1, RELEASE);`
`while(f2.load(ACQUIRE) == 0);`
`g = 2;`

Thread 2

`while(f1.load(ACQUIRE) == 0);`
`printf("%d", g);`
`f2.store(1, RELEASE);`

Thread 2 is non-racy

The program should only print **1**

If we perform overwritten write elimination it prints **0**

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;  
f1.store(1, RELEASE);  
while(f2.load(ACQUIRE) == 0);  
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE) == 0);  
printf("%d", g);  
f2.store(1, RELEASE);
```

sync



The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;  
f1.store(1, RELEASE);
```

```
g = 2;
```

sync

Thread 2

```
while(f1.load(ACQUIRE) == 0);  
printf("%d", g);  
f2.store(1, RELEASE);
```

The soundness condition

Shared memory

```
g = 0; atomic f1 = f2 = 0;
```

Thread 1

```
g = 1;  
f1.store(1, RELEASE);
```

```
g = 2;
```

Thread 2

```
while(f1.load(ACQUIRE)==0);  
printf("%d", g);  
f2.store(1, RELEASE);
```

sync



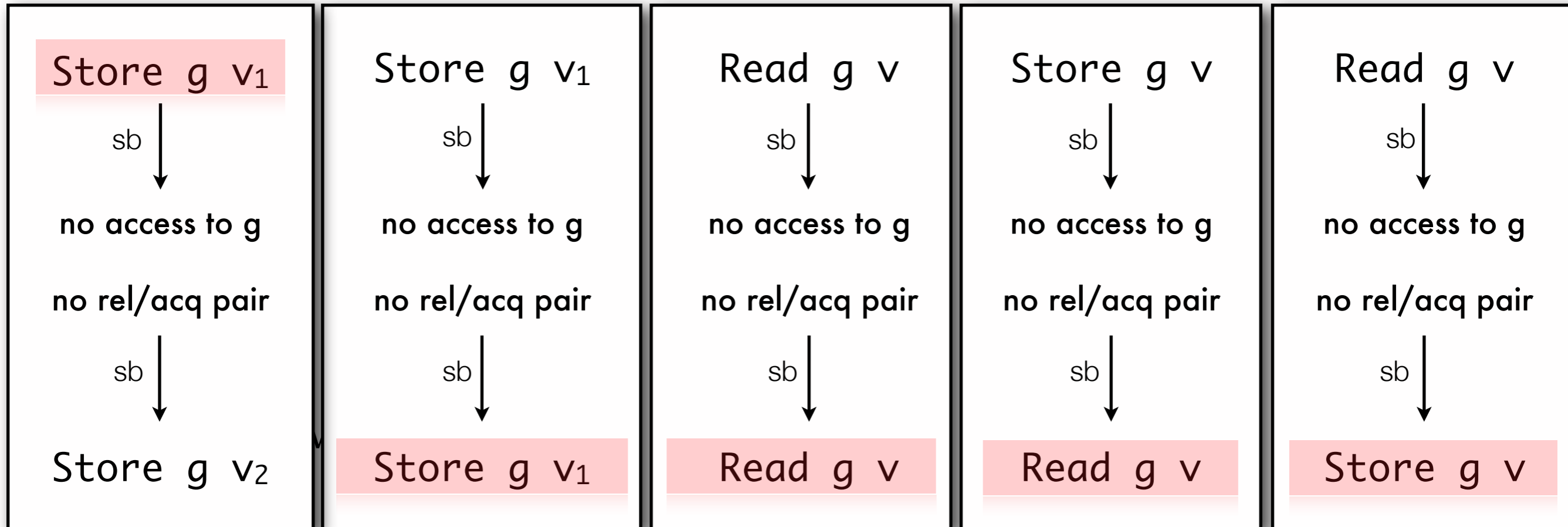
data race



If only a release (or acquire) is present, then
all discriminating contexts *are racy*.

It is sound to optimise the overwritten write.

Eliminations: bestiary



Overwritten-Write Write-after-Write Read-after-Read Read-after-Write Write-after-Read

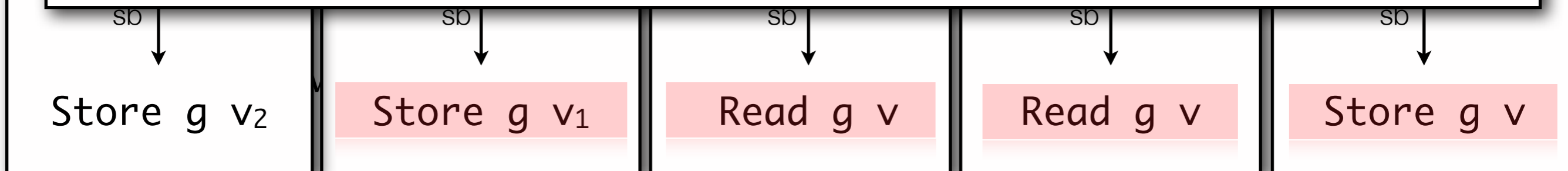
Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (*irrelevant reads*).

Eliminations: bestiary

Theorem

No non-racy context can observe these eliminations.

Proved w.r.t. Batty et al. (POPL 11) formalisation
of the C11/C++11 memory model



Overwritten-Write Write-after-Write Read-after-Read Read-after-Write Write-after-Read

Reads which are not used (via data or control dependencies) to decide a write or synchronisation event are also eliminable (*irrelevant reads*).

Reorderings and introductions

Correctness criterion for reordering events:

- different addresses
- no synchronisations in-between

Roach-motel reordering (reordering across locks) not observed in practice

Read introductions observed in practice (gcc, clang).

Introduction of eliminable reads proved correct.

Introduction of irrelevant reads does not introduce new behaviours, but cannot be proved correct in a DRF model.

3. From theory to the Cmmtest tool



**Random
Generator**



**SEQUENTIAL
PROGRAM**

*optimising
compiler
under test*

EXECUTABLE

tracing



**MEMORY
TRACE**

reference
semantics



**REFERENCE
MEMORY
TRACE**



**Check: only transformations sound
in any concurrent non-racy context**

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

*optimising
compiler
under test*

reference
semantics

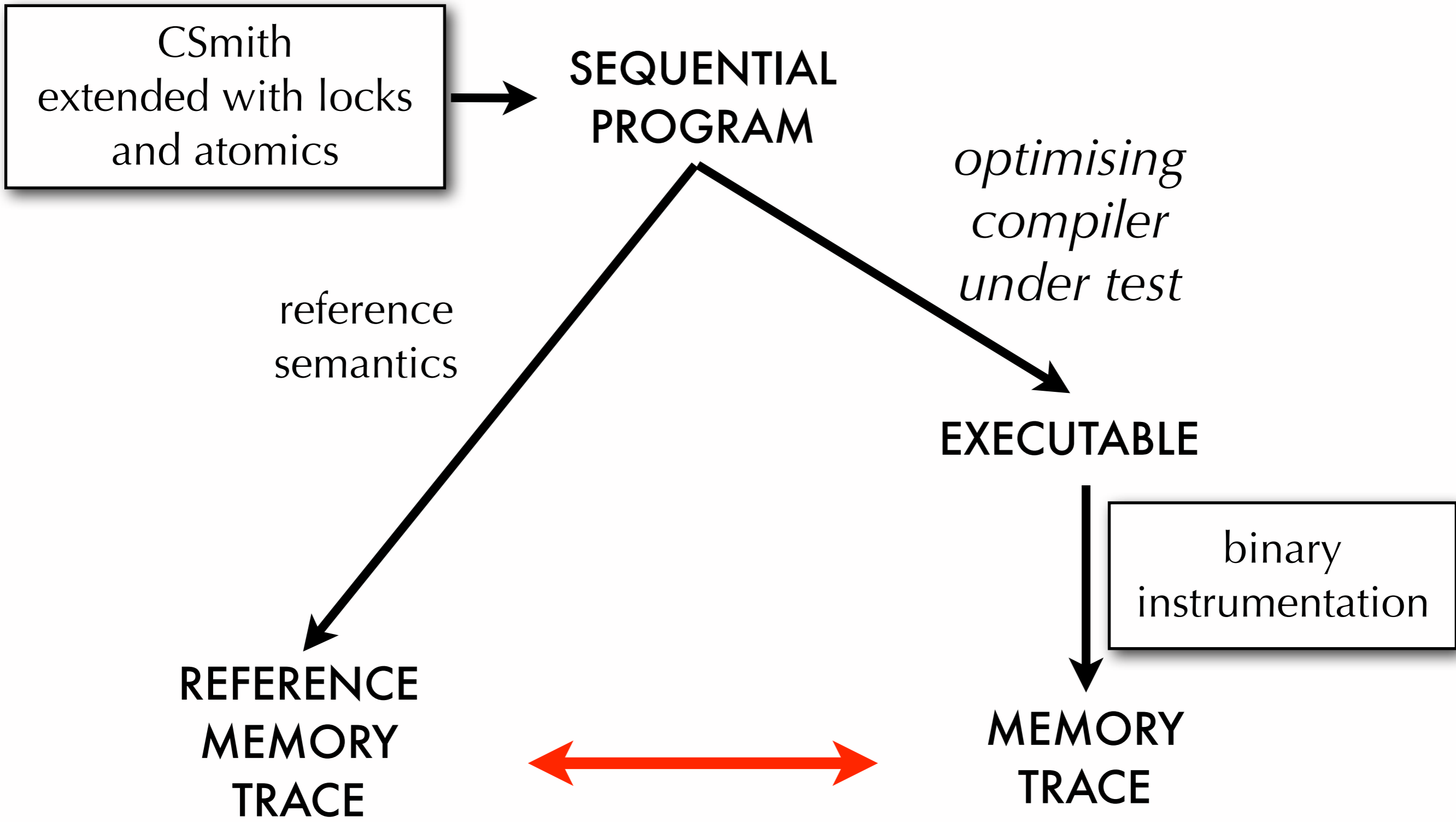
EXECUTABLE

tracing

REFERENCE
MEMORY
TRACE

MEMORY
TRACE

**Check: only transformations sound
in any concurrent non-racy context**



**Check: only transformations sound
in any concurrent non-racy context**

CSmith
extended with locks
and atomics

SEQUENTIAL
PROGRAM

*optimising
compiler
under test*

gcc/clang -O0

EXECUTABLE

EXECUTABLE

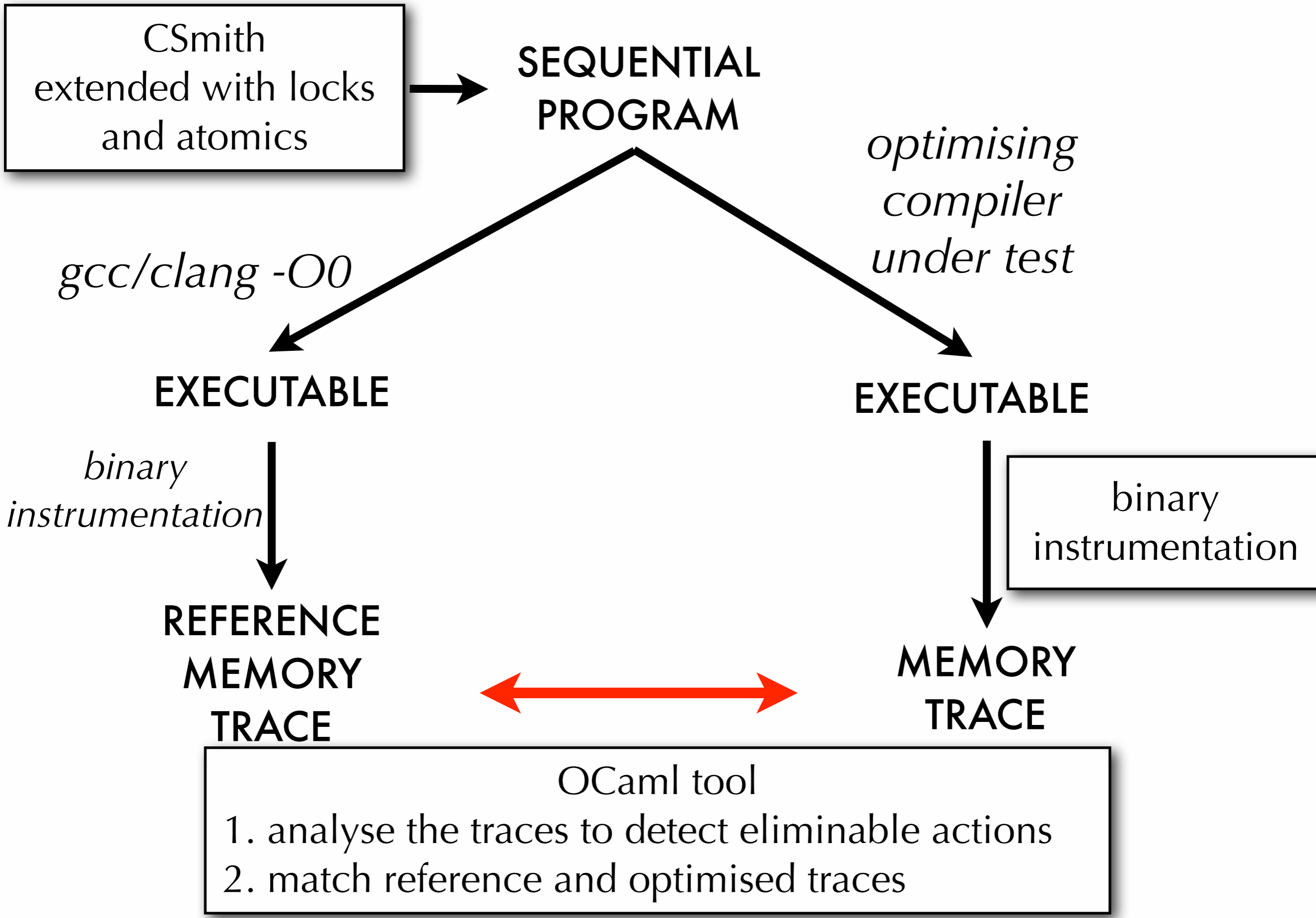
*binary
instrumentation*

REFERENCE
MEMORY
TRACE

binary
instrumentation

MEMORY
TRACE

**Check: only transformations sound
in any concurrent non-racy context**



```
const unsigned int g3 = 0UL;
long long g4 = 0x1;
int g6 = 6L;
volatile unsigned int g5 = 1UL;

void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7))*(*l102));
}
```

Start with a randomly generated well-defined program

```
const unsigned int g3 = 0UL; void func_1(void){
long long g4 = 0x1;          int *l8 = &g6;
int g6 = 6L;                 int l36 = 0x5E9D070FL;
volatile unsigned int g5 = 1UL; unsigned int l107 = 0xAA37C3ACL;
                               g4 &= g3;
                               g5++;
                               int *l102 = &l36;
                               for (g6 = 4; g6 < (-3); g6 += 1);
                               l102 = &g6;
                               *l102 = ((*l8) && (l107 << 7))*(*l102));
                               }
}
```

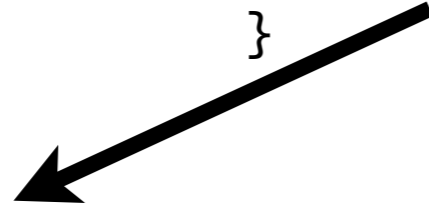
```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7))*(*l102));
}
```

```
Init g3 0
Init g4 1
Init g5 1
Init g6 6
```

```
void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7))*(*l102));
}
```

reference
semantics



```
Load g4 1
Store g4 0
Load g5 1
Store g5 2
Store g6 4
Load g6 4
Load g6 4
Load g6 4
Store g6 1
Load g4 0
```

Init g3 0
Init g4 1
Init g5 1
Init g6 6

```
void func_1(void){  
    int *l8 = &g6;  
    int l36 = 0x5E9D070FL;  
    unsigned int l107 = 0xAA37C3ACL;  
    g4 &= g3;  
    g5++;  
    int *l102 = &l36;  
    for (g6 = 4; g6 < (-3); g6 += 1);  
    l102 = &g6;  
    *l102 = ((*l8) && (l107 << 7))*(*l102));  
}
```

reference
semantics

gcc -O2 memory trace

Load g4 1
Store g4 0
Load g5 1
Store g5 2
Store g6 4
Load g6 4
Load g6 4
Load g6 4
Store g6 1
Load g4 0

Load g5 1
Store g4 0
Store g6 1
Store g5 2
Load g4 0


```

Init g3 0
Init g4 1
Init g5 1
Init g6 6

```

```

void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7))*(*l102));
}

```

reference semantics

gcc -O2 memory trace

```

RaW* Load g4 1
      Store g4 0
RaW* Load g5 1
      Store g5 2
OW* Store g6 4
RaW* Load g6 4
RaR* Load g6 4
RaR* Load g6 4
      Store g6 1
RaW* Load g4 0

```

```

Load g5 1
Store g4 0
Store g6 1
Store g5 2
Load g4 0

```



```

Init g3 0
Init g4 1
Init g5 1
Init g6 6

```

```

void func_1(void){
    int *l8 = &g6;
    int l36 = 0x5E9D070FL;
    unsigned int l107 = 0xAA37C3ACL;
    g4 &= g3;
    g5++;
    int *l102 = &l36;
    for (g6 = 4; g6 < (-3); g6 += 1);
    l102 = &g6;
    *l102 = ((*l8) && (l107 << 7))*(*l102));
}

```

reference semantics

gcc -O2 memory trace

```

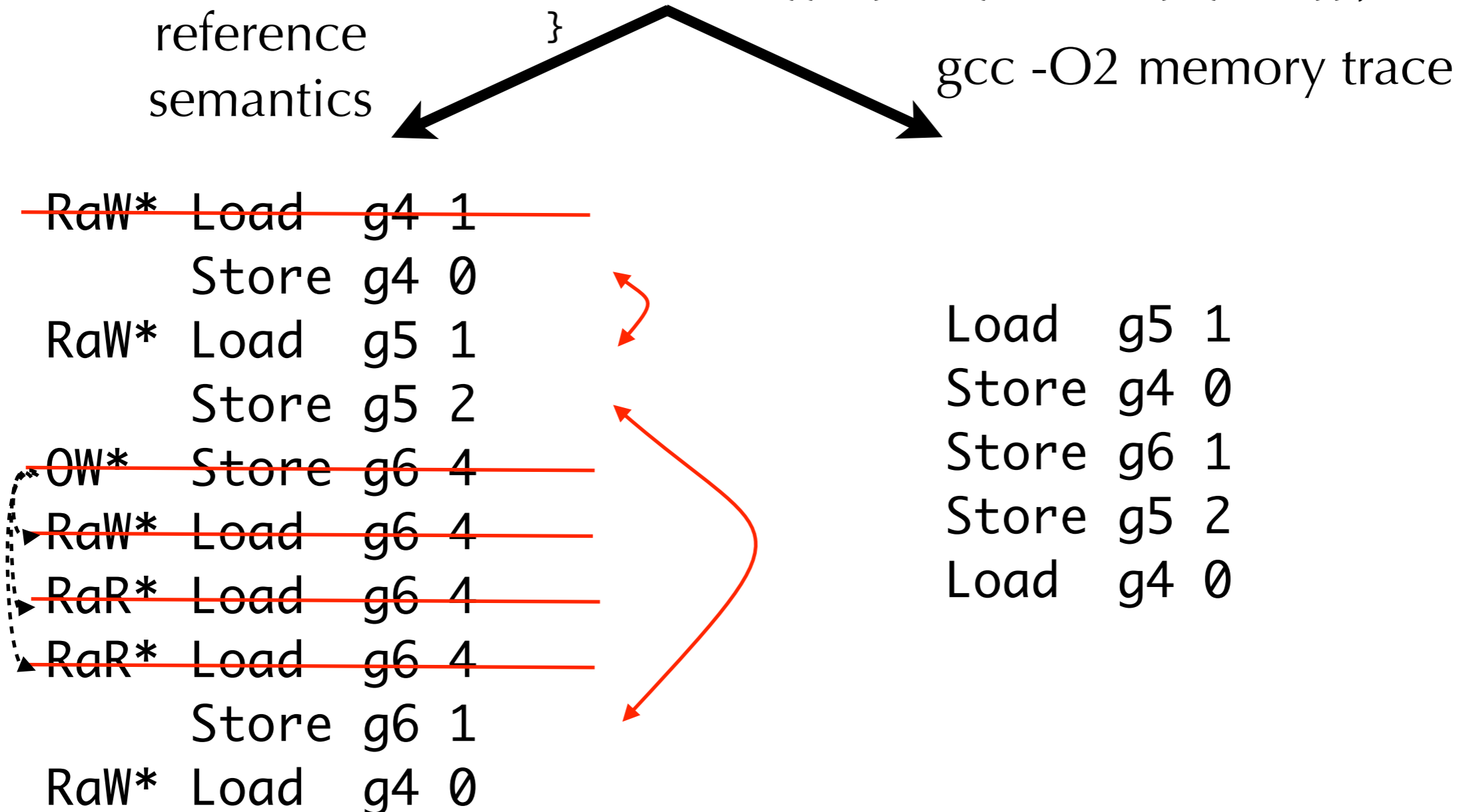
RaW* Load g4 1
      Store g4 0
RaW* Load g5 1
      Store g5 2
OW* Store g6 4
RaW* Load g6 4
RaR* Load g6 4
RaR* Load g6 4
      Store g6 1
RaW* Load g4 0

```

```

Load g5 1
Store g4 0
Store g6 1
Store g5 2
Load g4 0

```



Init g3 0

```
void func_1(void){  
    int *l8 = &g6;
```

Can match applying
only correct eliminations and reorderings

```
*l102 = ((*l8) && (l107 << 7))*(*l102));
```

reference
semantics

gcc -O2 memory trace

~~RaW*~~ Load g4 1
Store g4 0
RaW* Load g5 1
Store g5 2
~~OW*~~ Store g6 4
~~RaW*~~ Load g6 4
~~RaR*~~ Load g6 4
~~RaR*~~ Load g6 4
Store g6 1
RaW* Load g4 0

Load g5 1
Store g4 0
Store g6 1
Store g5 2
Load g4 0



```
int a = 1;  
int b = 0;
```

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

If we focus on the miscompiled initial example...

```
int a = 1;
int b = 0;
```

```
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b>=26; ++b)
        ;
}
```

```
int a = 1;  
int b = 0;
```

```
int s;  
for (s=0; s!=4; s++) {  
    if (a==1)  
        return NULL;  
    for (b=0; b>=26; ++b)  
        ;  
}
```

reference
semantics



Load a 1

```
int a = 1;
int b = 0;
```

```
int s;
for (s=0; s!=4; s++) {
    if (a==1)
        return NULL;
    for (b=0; b<=26; ++b)
        ;
}
```

reference
semantics



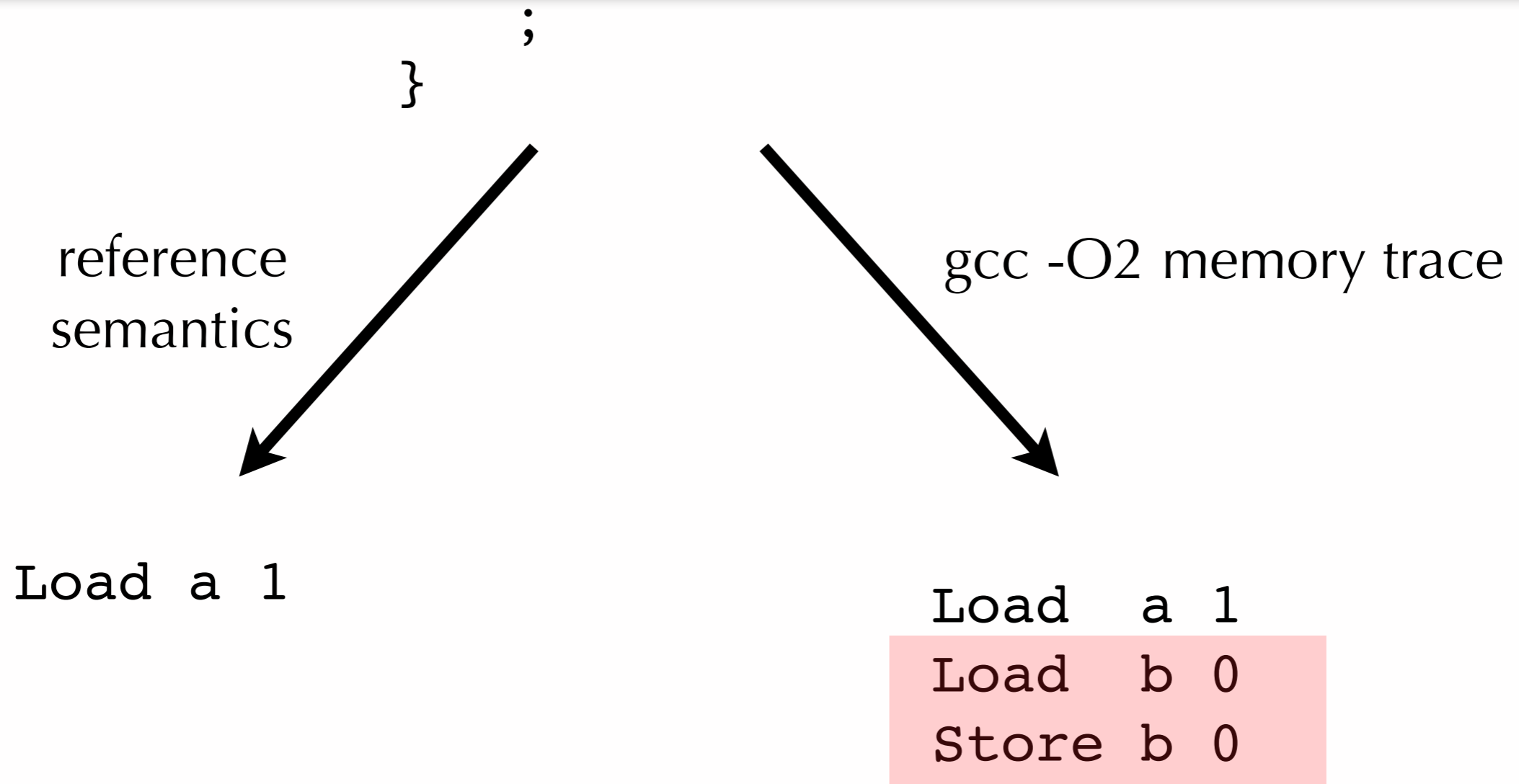
Load a 1

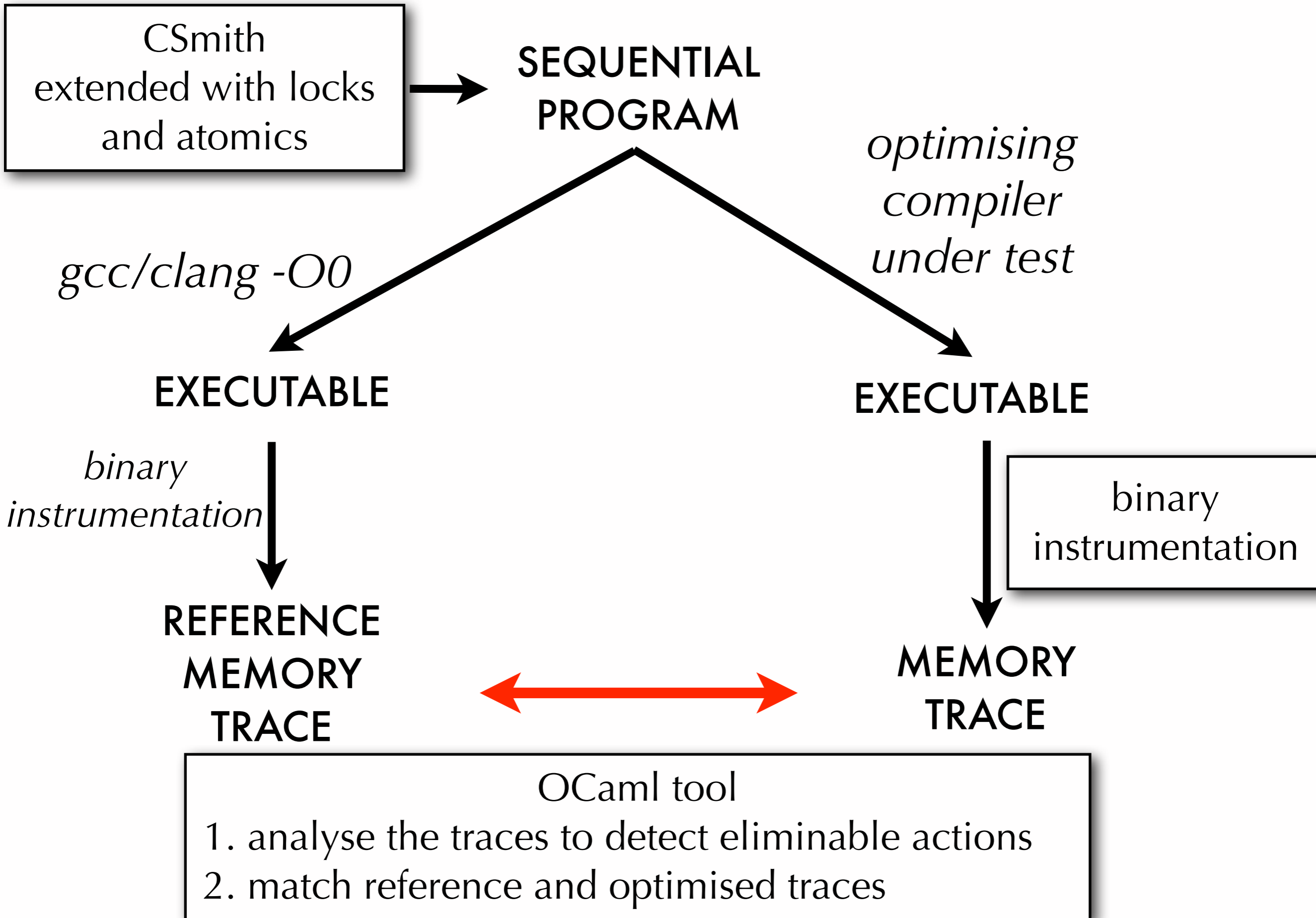
gcc -O2 memory trace



Load a 1
Load b 0
Store b 0

Cannot match some events \longrightarrow detect compiler bug





Subtleties:

- *dependencies between eliminable events*
- *some optimisations (e.g. merging of accesses) cannot be expressed in the C11/C++11 formalisation*
- *the tool also ensures that the compilation of atomic accesses is preserved by the optimiser*

**REFERENCE
MEMORY
TRACE**

**MEMORY
TRACE**



OCaml tool

1. analyse the traces to detect eliminable actions
2. match reference and optimised traces

instrumentation

Subtleties:

- *dependencies between eliminable events*
- *some optimisations (e.g. merging of accesses) cannot be expressed in the C11/C++11 formalisation*
- *the pre*

*Going from theory
to a tool that scales to real compilers
is a
full-time job*

1. *analyse the traces to detect eliminable actions*
2. *match reference and optimised traces*

4. Impact on the non-academic world



1. Some GCC concurrency bugs

Some concurrency compiler bugs found
in the latest version of GCC.

Store introductions performed by loop invariant motion or
if-conversion optimisations.

Remark: these bugs break the Posix thread model too.

All promptly fixed.

Good timing

Existing compilers are being adapted to the C11/C++11 standard

```
> Just to get this straight, am I to assume that the default code  
> generation for GCC is a single threaded environment?
```

```
It certainly is, though we are getting more careful about this stuff  
in recent years and generally only read data-races are ok.
```

store introductions performed by loop invariant motion or
if-conversion optimisations.

Remark: these bugs break the Posix thread model too.

All promptly fixed.

Good timing

Existing compilers are being adapted to the C11/C++11 standard

```
> Just to get this straight, am I to assume that the default code  
> generation for GCC is a single threaded environment?
```

It c
in r

Friends

Previous work introduced us to several gcc developers that pushed for our bug reports to be fixed.

All promptly fixed.

store r
if-conv
Reman

2. Checking compiler invariants

GCC internal invariant: never reorder with an atomic access

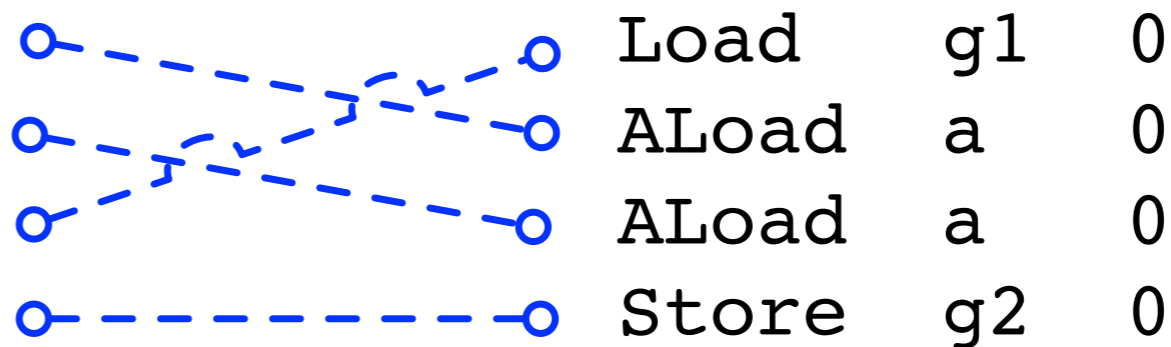
Baked this invariant into the tool and found a counterexample...

...not a bug, but fixed anyway

```
atomic_uint a;  
int32_t g1, g2;
```


```
int main (int, char *[]) {  
    a.load() & a.load ();  
    g2 = g1 != 0;  
}
```

```
ALoad  a  0  
ALoad  a  0  
Load   g1 0  
Store  g2 0
```



3. Detecting unexpected behaviours

uint16_t g

for (; g==0; g--);  g=0;

If `g` is initialised with `0`, a load gets replaced by a store:

Load g 0  Store g 0

The introduced store cannot be observed by a non-racy context.

Still, arguable if a compiler should do this or not.

3. Detecting unexpected behaviours

Spin-off

Collaboration with ThreadSanitizer developers to detect false positives due to memory access introductions.

Load g 0) — ? — (Store g 0

The introduced store cannot be observed by a non-racy context.

Still, arguable if a compiler should do this or not.

5. Behind the scenes





Low-level software

Applications

Language definitions



Architectures

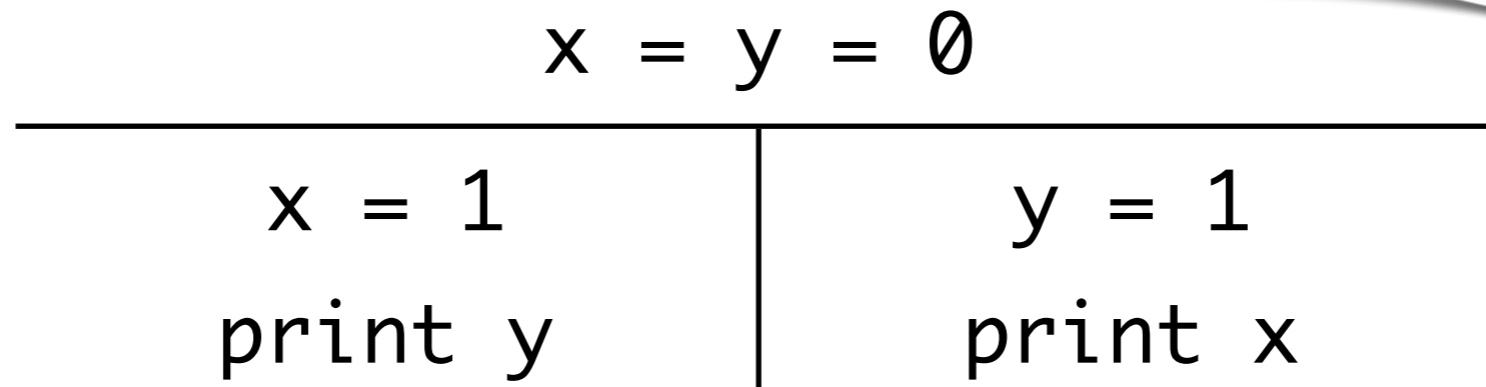
Hardware

Low-level software



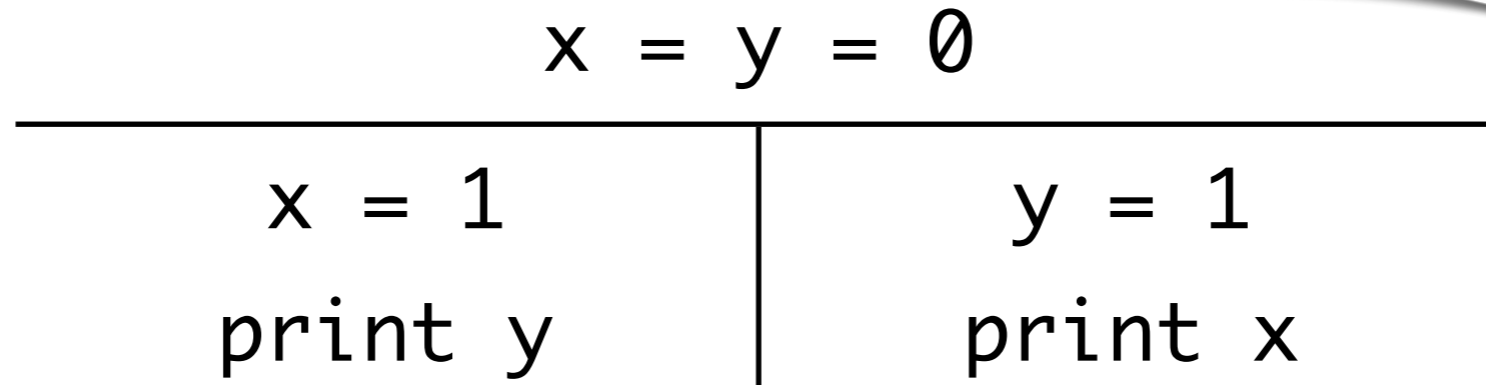
Hardware memory models

Dekker algorithm



Hardware memory models

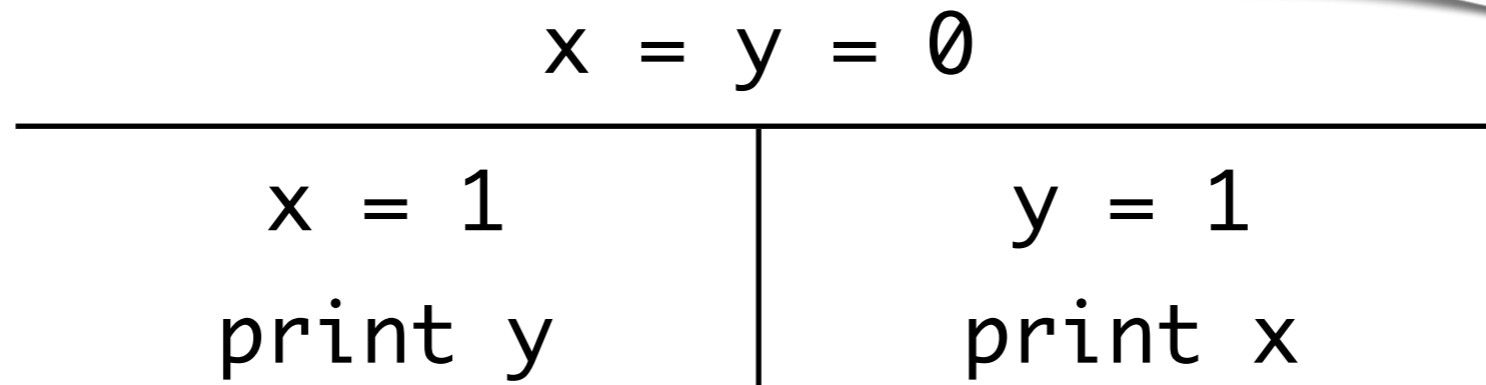
Dekker algorithm



Should not terminate printing $\emptyset \emptyset$.

Hardware memory models

Dekker algorithm



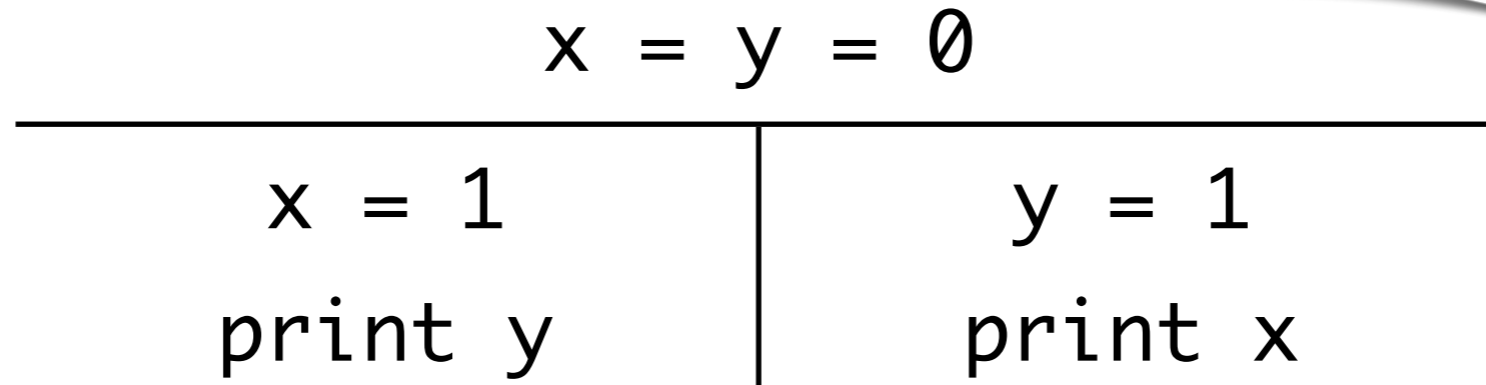
Should not terminate printing 0 0.

Let's see...



Hardware memory models

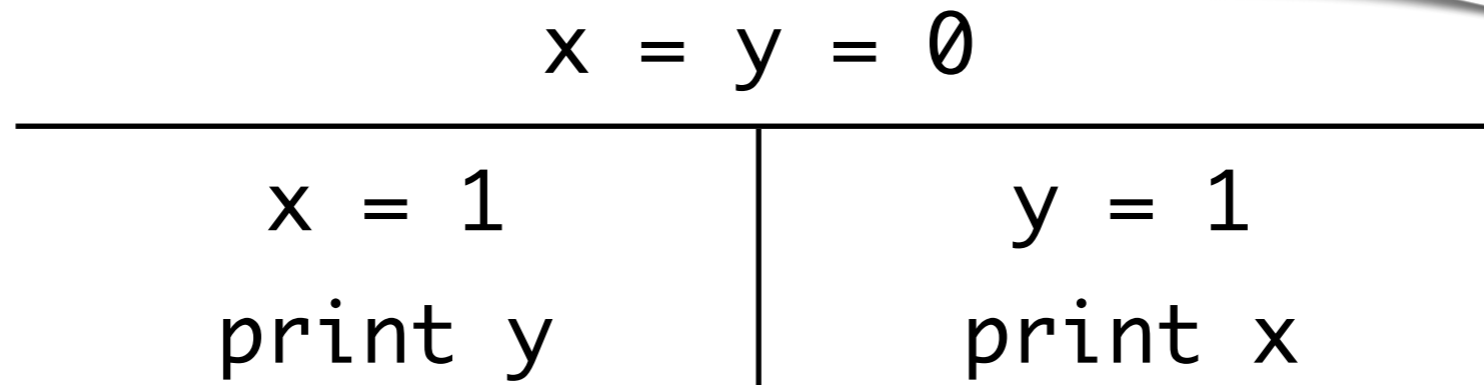
Dekker algorithm



The output **0 0** can be observed on x86 and on ARM/Power.

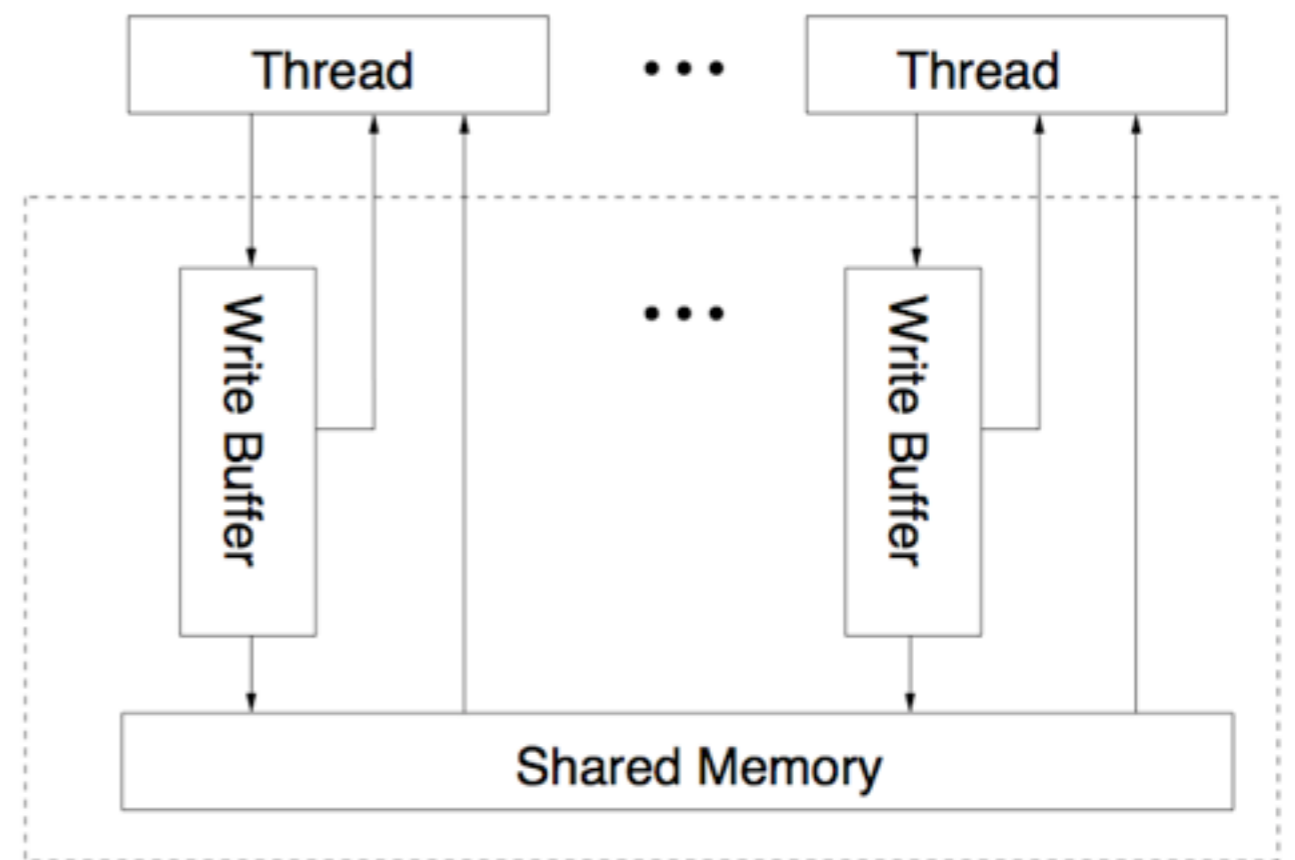
Hardware memory models

Dekker algorithm



The output **0 0** can be observed on x86 and on ARM/Power.

Write buffers hide the latency of writes to memory but are observable by concurrent code.



Memory models differ between architectures

message passing

$x = f = 0$

$x = 1$

print f

$f = 1$

print x

Memory models differ between architectures

message passing

$$x = f = 0$$

$x = 1$	$\text{print } f$
$f = 1$	$\text{print } x$

Should not terminate printing **1 0**.

Memory models differ between architectures

message passing

$x = f = 0$	
$x = 1$	print f
$f = 1$	print x

Should not terminate printing **1 0**.

Let's see...



Memory models differ between architectures

message passing

$x = f = 0$

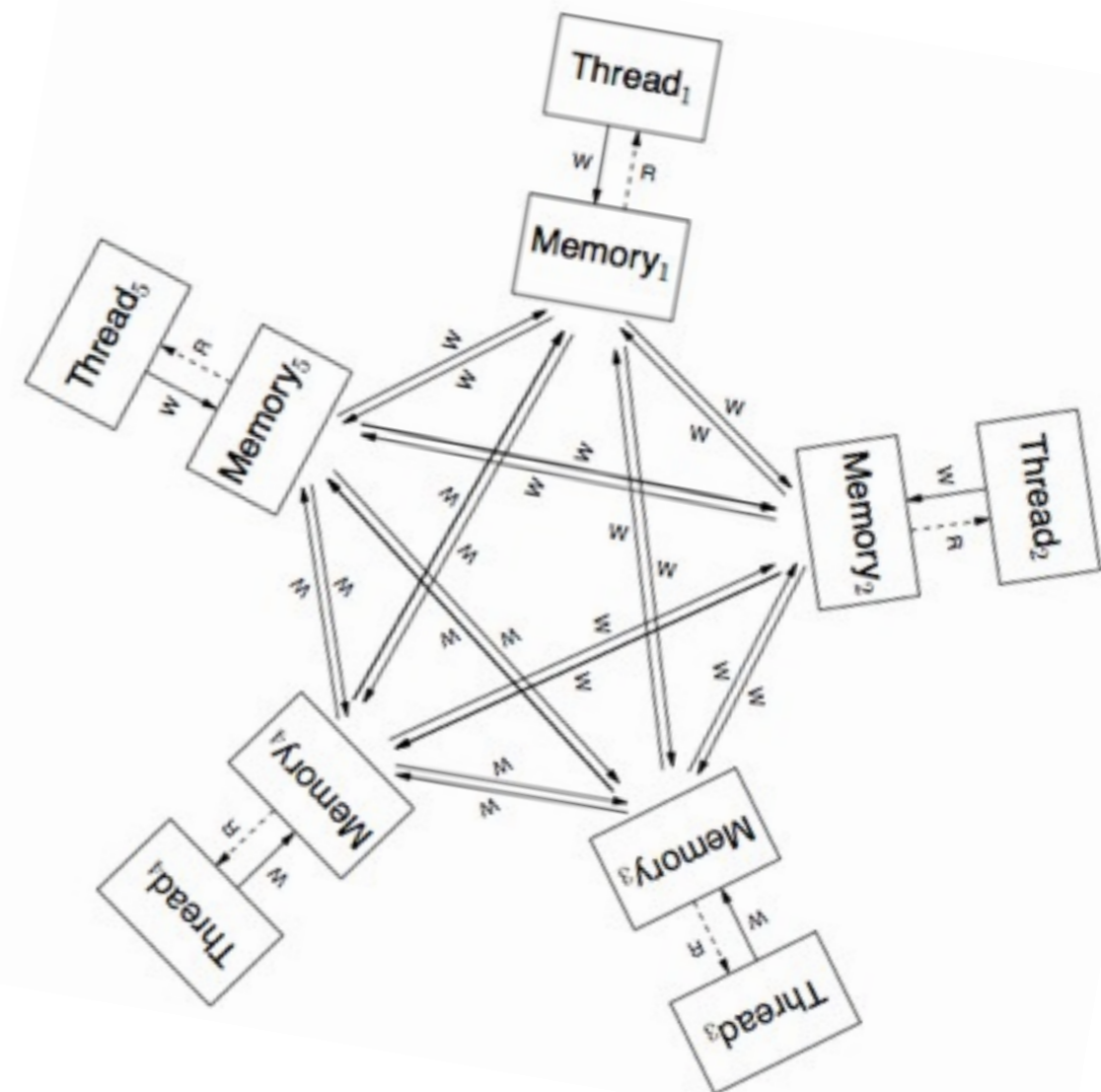
$x = 1$

print f

$f = 1$

print x

The output **0 0** can be observed on ARM/Power but not on x86.



Establishing models (done, almost)

X86

Sarkar, Owens, Sewell, Zappa Nardelli

ARM/Power

Alglave, Maranget, Sarkar, Sewell, Williams

C11/C++11

Batty, Owens, Sarkar, Memarian, Sewell, Weber

Testing against the models (doable)

Soundness of optimisations in JSR-133 and C11/C++11

Sevcik, Morisset, Pawan, Zappa Nardelli

Model checker for C11/C++11 atomics

Norris, Demsky

Verification above the models (in its infancy)

Compilation scheme from C11/C++11 to Power/ARM and X86

Sarkar, Owens, Memarian, Batty, Sewell, Owens

Verified compilation from a concurrent C-like language to X86

Sevcik, Vafeiadis, Zappa Nardelli, Jagannathan, Sewell

doing for C11/C++11 would be much harder, doing it for GCC would be impossible

Fence-elimination optimisations

Vafeiadis, Zappa Nardelli

X86 lock algorithms, work-stealing algorithms on X86 and Power/ARM

Owens, Lê, Morisset, Guatto, Cohen, Zappa Nardelli

...

Take-up in Industrial Concurrency Community?



Handled the real behaviour

Found some bugs

Ongoing dialogue with
language designers
and developers

Take-up in Industrial Concurrency Community?



Still, many open problems.

ur

Take-up in Industrial Concurrency Community?

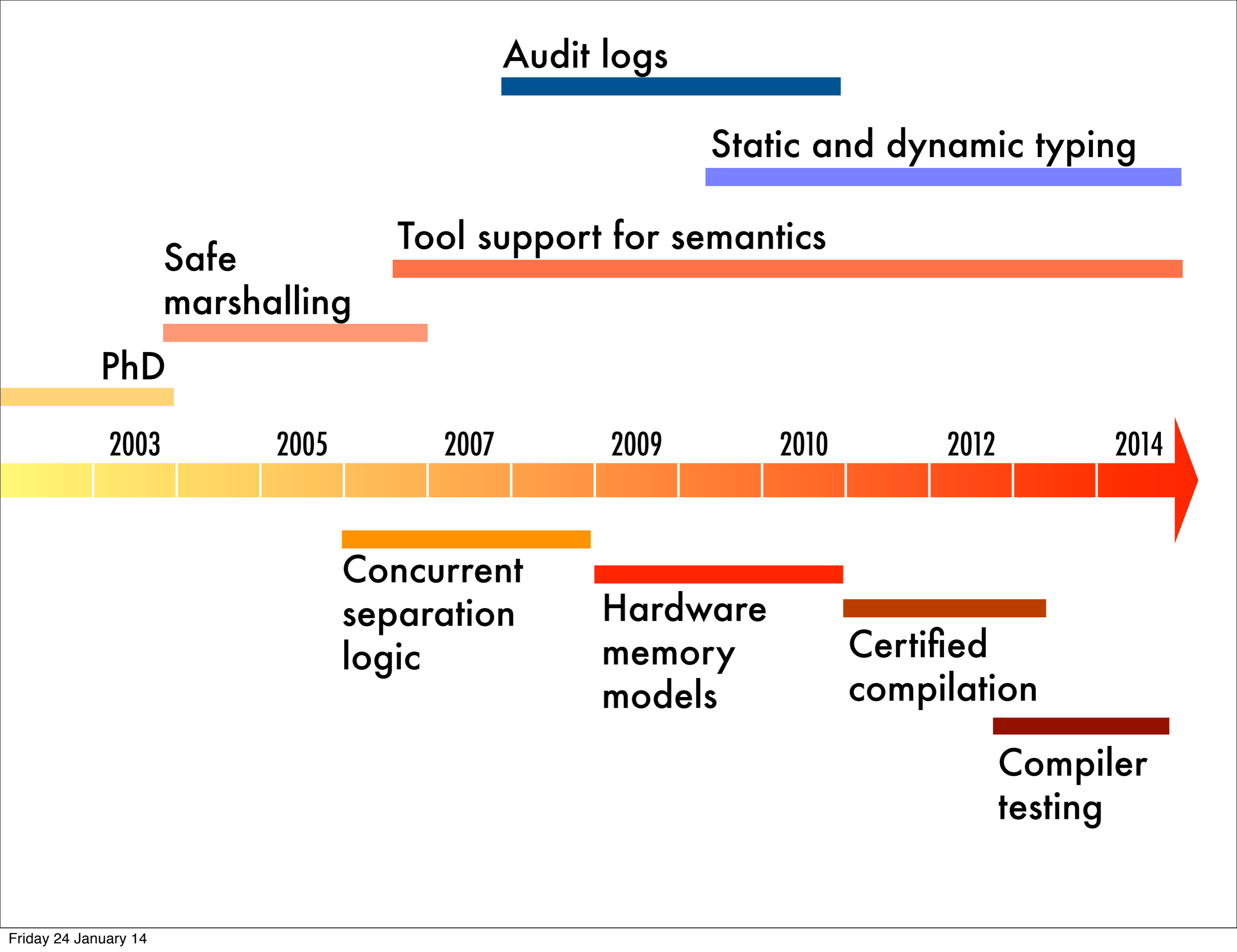


Still, many research opportunities!

ur

6. The Big Picture





Audit logs

Static and dynamic typing

Tool support for semantics

Safe marshalling

PhD

2003

2005

2007

2009

2010

2012

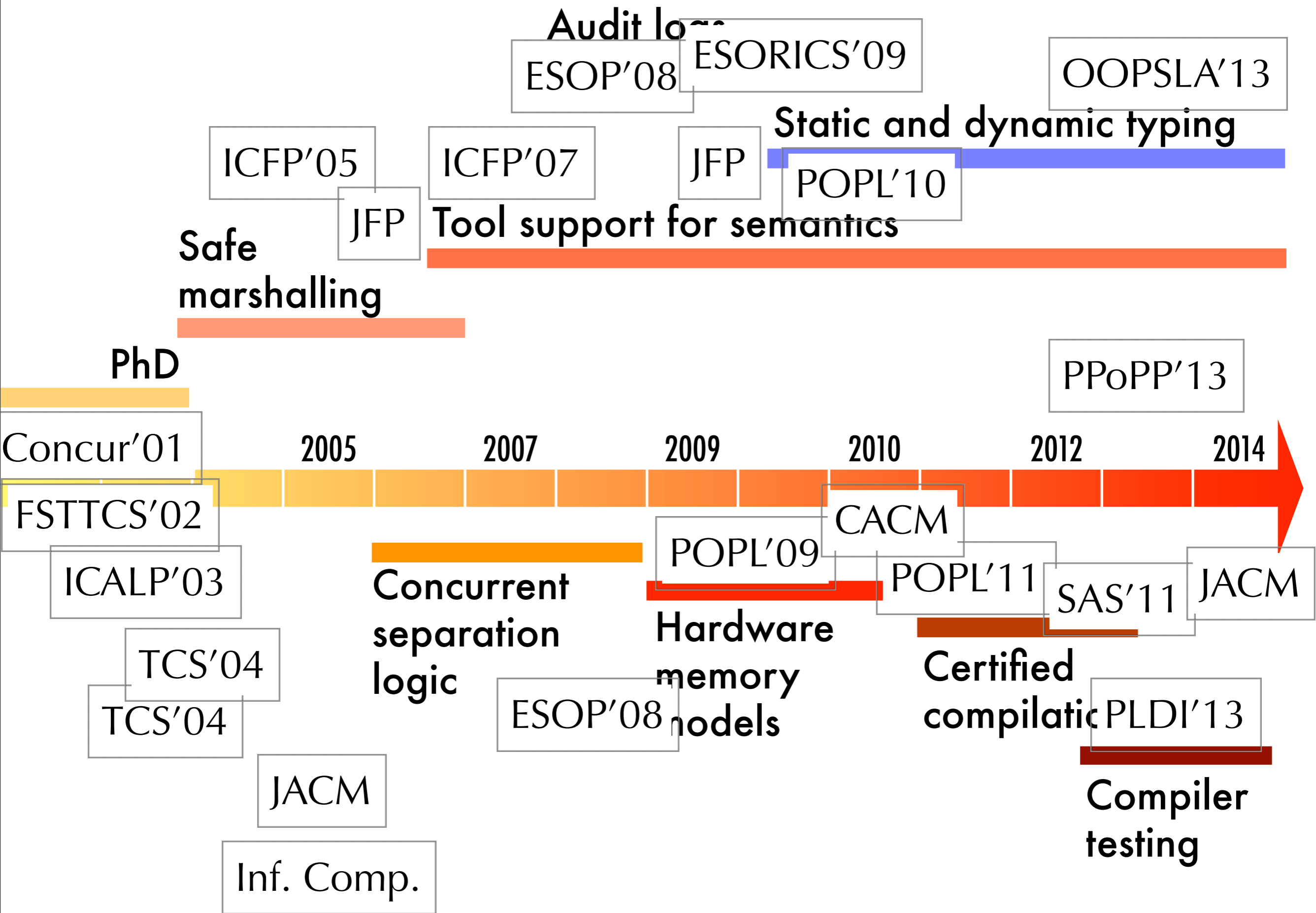
2014

Concurrent separation logic

Hardware memory models

Certified compilation

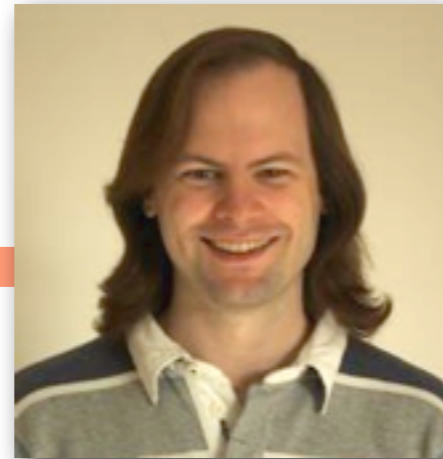
Compiler testing





Sto

t for sema



2005

2007

2009

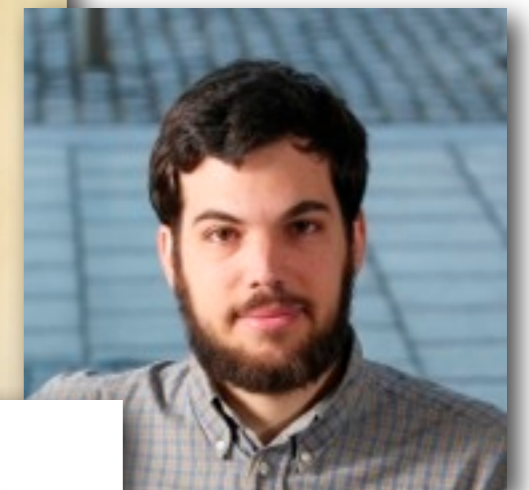
2010

2012

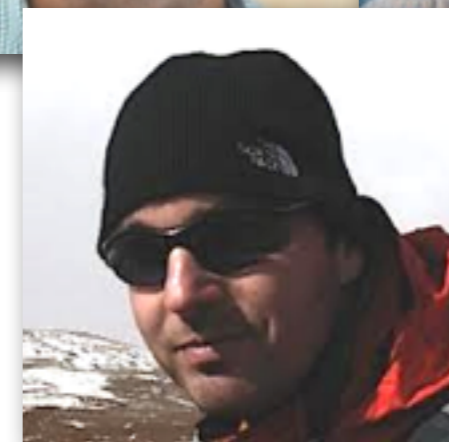
2014



are
y



comple
esting





atic and dynamic typing

Tool s ntics

Safe marshalling

PhD



2005

2007

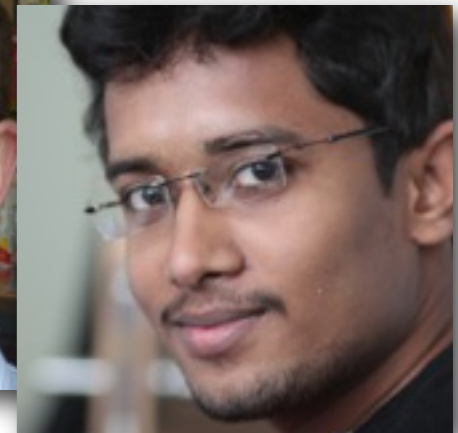
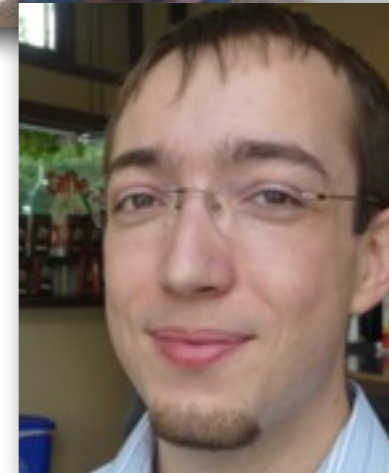
2009

2010

2012

2014

Concur
separat
logic



Questions?

