A Certified Data Race Analysis for a Java-like Language

Frédéric Dabrowski and David Pichardie

INRIA Rennes - Bretagne Atlantique

TPHOLs 2009, 18 August 2009, Munich

Data Races

- A fundamental issue in multi-threaded programming
- Definition: *the situation where two different processes attempt to access to the same memory location and at least one access is a write.*
- Leads to tricky bugs
  - difficult to reproduce and identify via manual code review or program testing
- Java Memory Model is a complex thing...
  - Data-race-free programs are sequentially consistent
  - We need to prove the data-race-freeness of a program before safely reasonning on its interleaving semantic.

Example

```
         C.f = C.g = 0;
1: x = C.g; ‖ 1: y = C.f;
2: C.f = 1; ‖ 2: C.g = 1;
```

Example

```
          C.f = C.g = 0;
1: x = C.g; ‖ 1: y = C.f;
2: C.f = 1; ‖ 2: C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

Example

```
      C.f = C.g = 0;
1: x = C.g; ‖ 1: y = C.f;
2: C.f = 1; ‖ 2: C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

```
1: x = C.g;
2: C.f = 1;
1: y = C.f;
2: C.g = 1;
```

Example

```
         C.f = C.g = 0;
1: x = C.g; ‖ 1: y = C.f;
2: C.f = 1; ‖ 2: C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

```
1: x = C.g;     1: y = C.f;
2: C.f = 1;     2: C.g = 1;
1: y = C.f;     1: x = C.g;
2: C.g = 1;     2: C.f = 1;
```

Example

```
        C.f = C.g = 0;
1: x = C.g; ‖ 1: y = C.f;
2: C.f = 1; ‖ 2: C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

```
1: x = C.g;     1: y = C.f;     1: y = C.f;
2: C.f = 1;     2: C.g = 1;     1: x = C.g;
                                                …
1: y = C.f;     1: x = C.g;     2: C.g = 1;
2: C.g = 1;     2: C.f = 1;     2: C.f = 1;
```

Example

```
            C.f = C.g = 0;
1: x = C.g; ‖ 1: y = C.f;
2: C.f = 1; ‖ 2: C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

```
1: x = C.g;     1: y = C.f;     1: y = C.f;
2: C.f = 1;     2: C.g = 1;     1: x = C.g;
1: y = C.f;     1: x = C.g;     2: C.g = 1;     ...
2: C.g = 1;     2: C.f = 1;     2: C.f = 1;
```

but such program may also lead to sequentially inconsistent execution.

Example

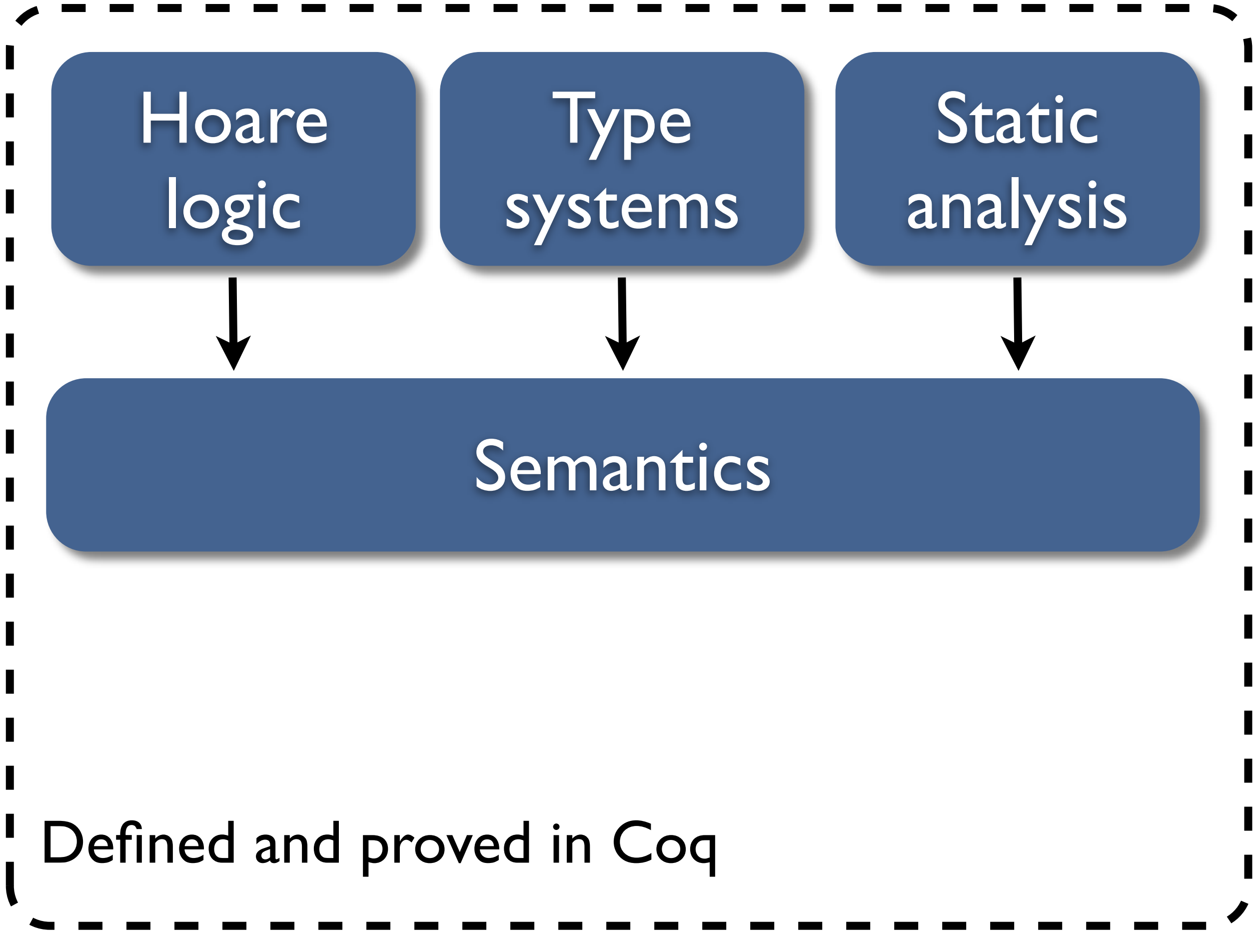
```
     C.f = C.g = 0;
1: x = C.g; ‖ 1: y = C.f;
2: C.f = 1; ‖ 2: C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

```
1: x = C.g;    1: y = C.f;    1: y = C.f;          2: C.g = 1;
2: C.f = 1;    2: C.g = 1;    1: x = C.g;          2: C.f = 1;
1: y = C.f;    1: x = C.g;    2: C.g = 1;   ...    1: x = C.g;
2: C.g = 1;    2: C.f = 1;    2: C.f = 1;          1: y = C.f;
          ✔              ✔              ✔                   ✗ x=1 and y=1 !
```

but such program may also lead to sequentially inconsistent execution.

Certified program verification



Hoare logic

Type systems

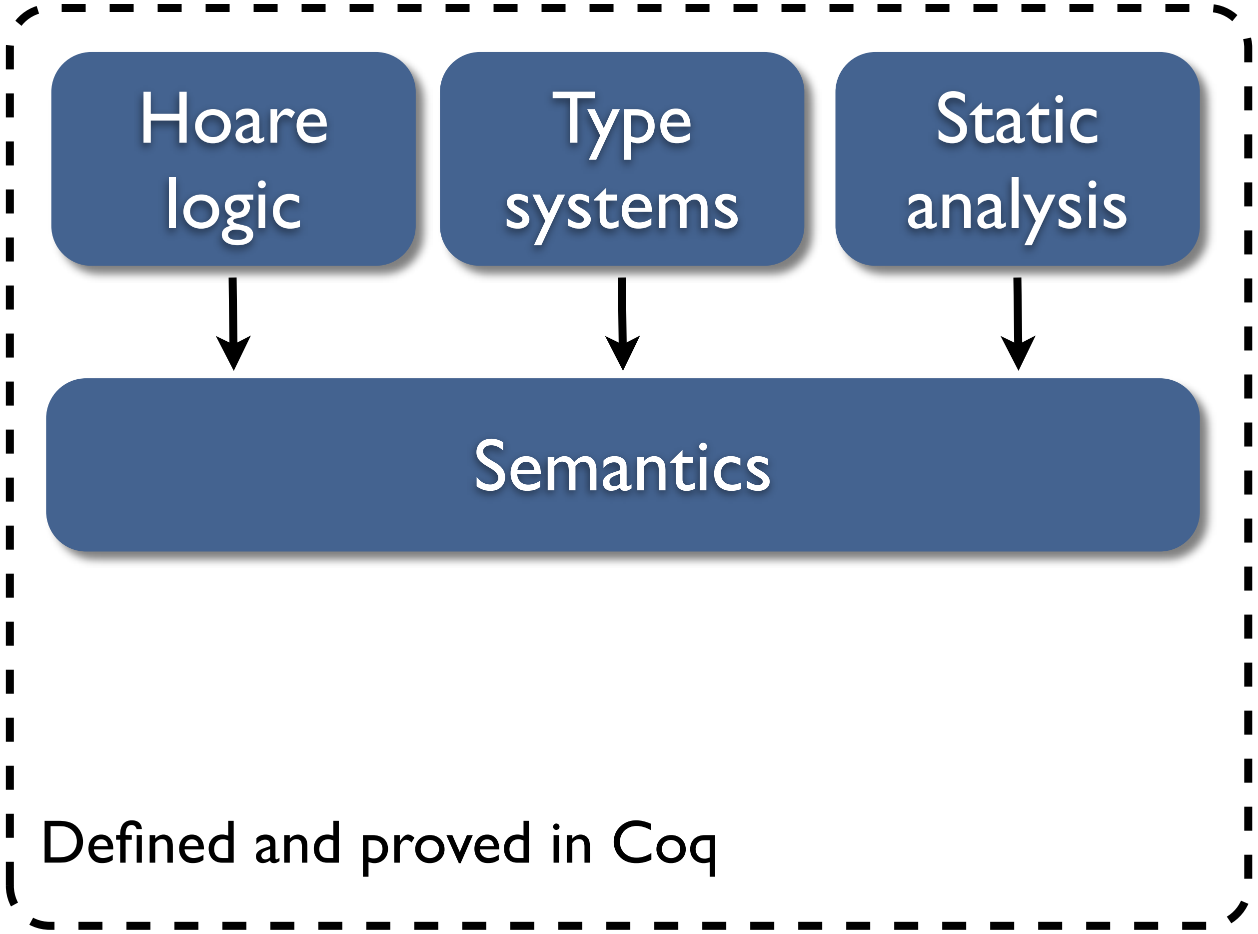
Static analysis

Semantics

Defined and proved in Coq

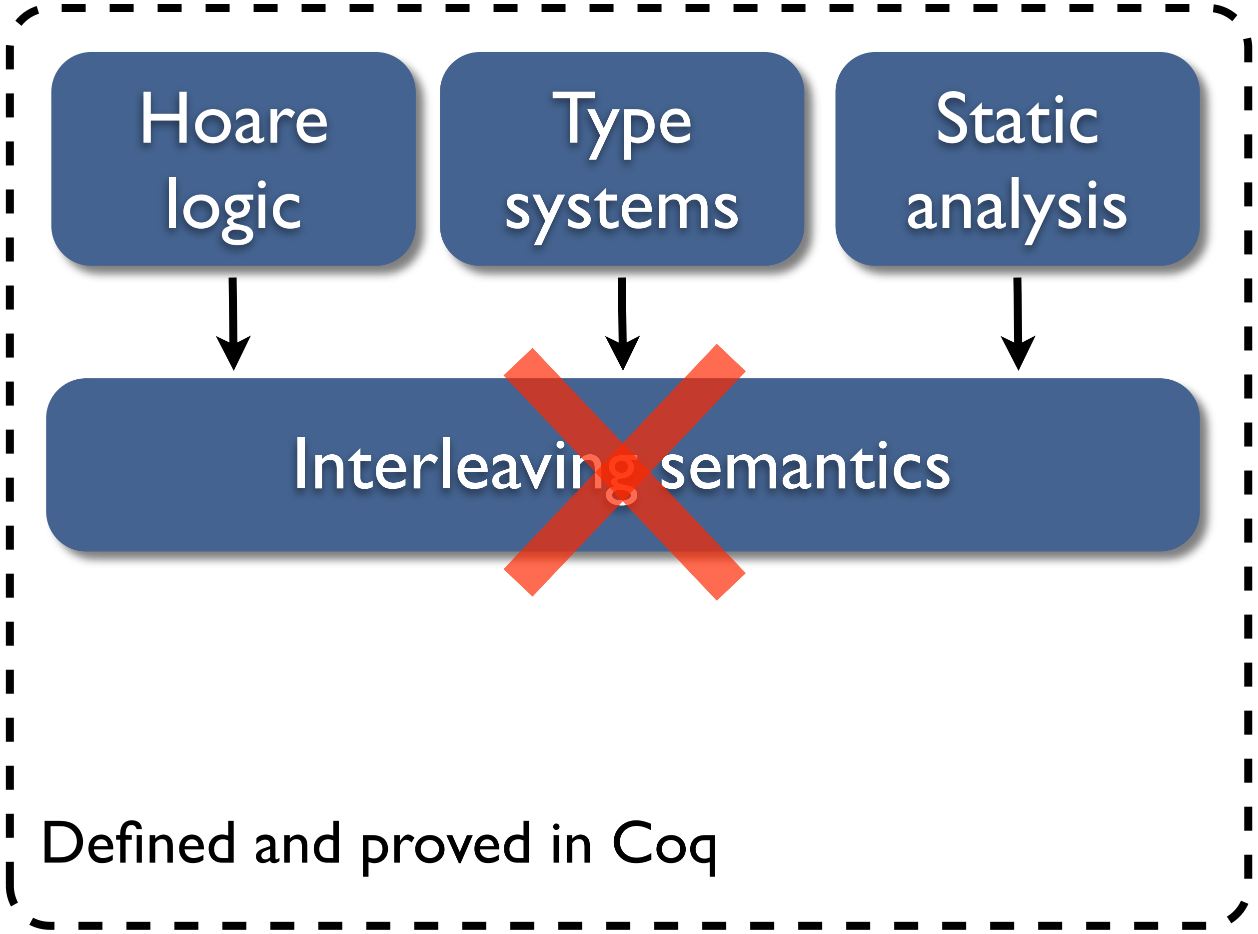
Certified program verification

- There is a growing interest in machine checked semantics proofs
- Program verification framework can be certified in a proof assistant
  - Example : MOBIUS project
  - All component are proved correct



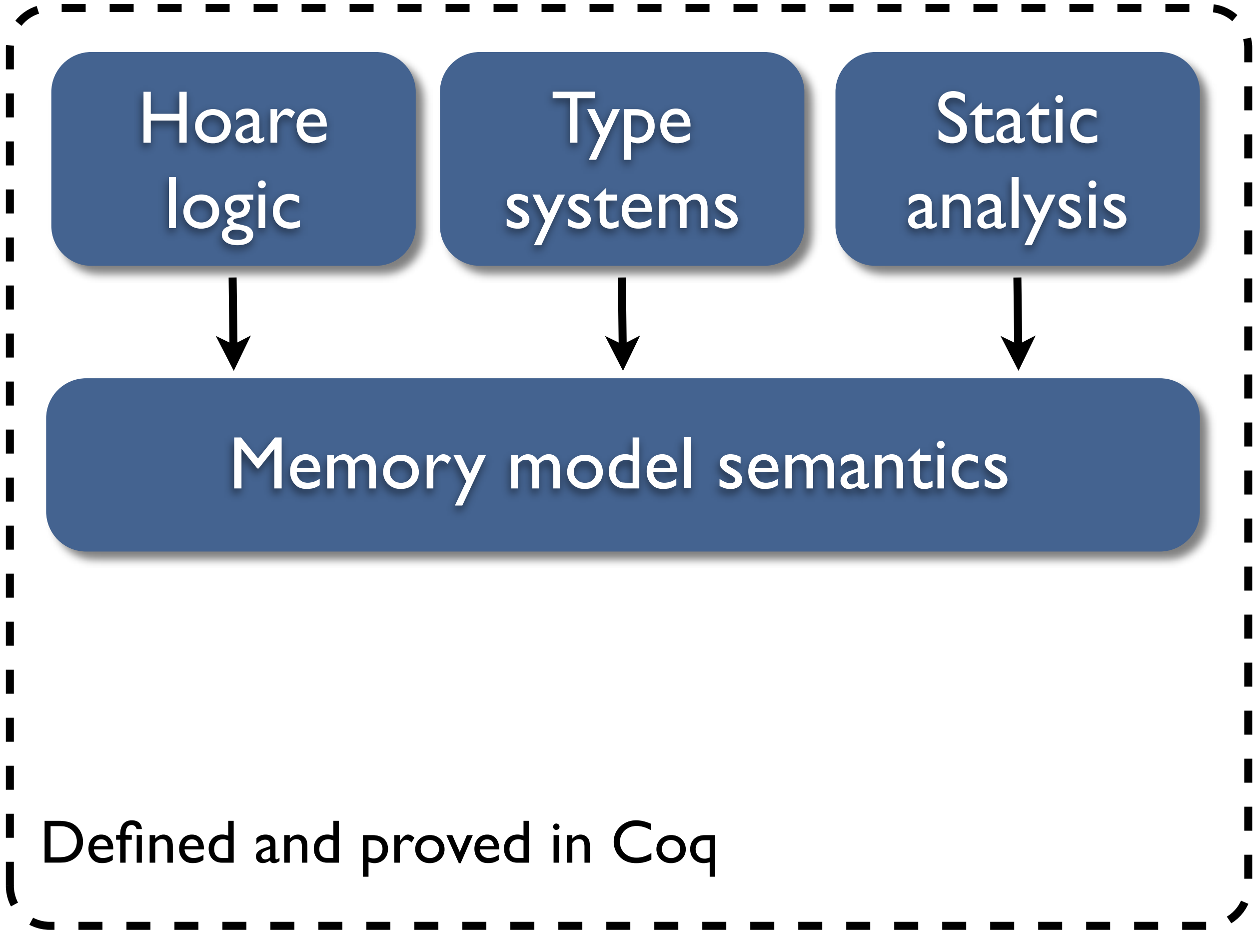
Certified program verification

- There is a growing interest in machine checked semantics proofs
- Program verification framework can be certified in a proof assistant
  - Example : MOBIUS project
  - All component are proved correct
- In a multi-threaded context
  - Using an interleaving semantics is unsound



Certified program verification

- There is a growing interest in machine checked semantics proofs
- Program verification framework can be certified in a proof assistant
  - Example : MOBIUS project
  - All component are proved correct
- In a multi-threaded context
  - Using an interleaving semantics is unsound
  - Reasoning directly on the JMM is very painful


Hoare logic

Type systems

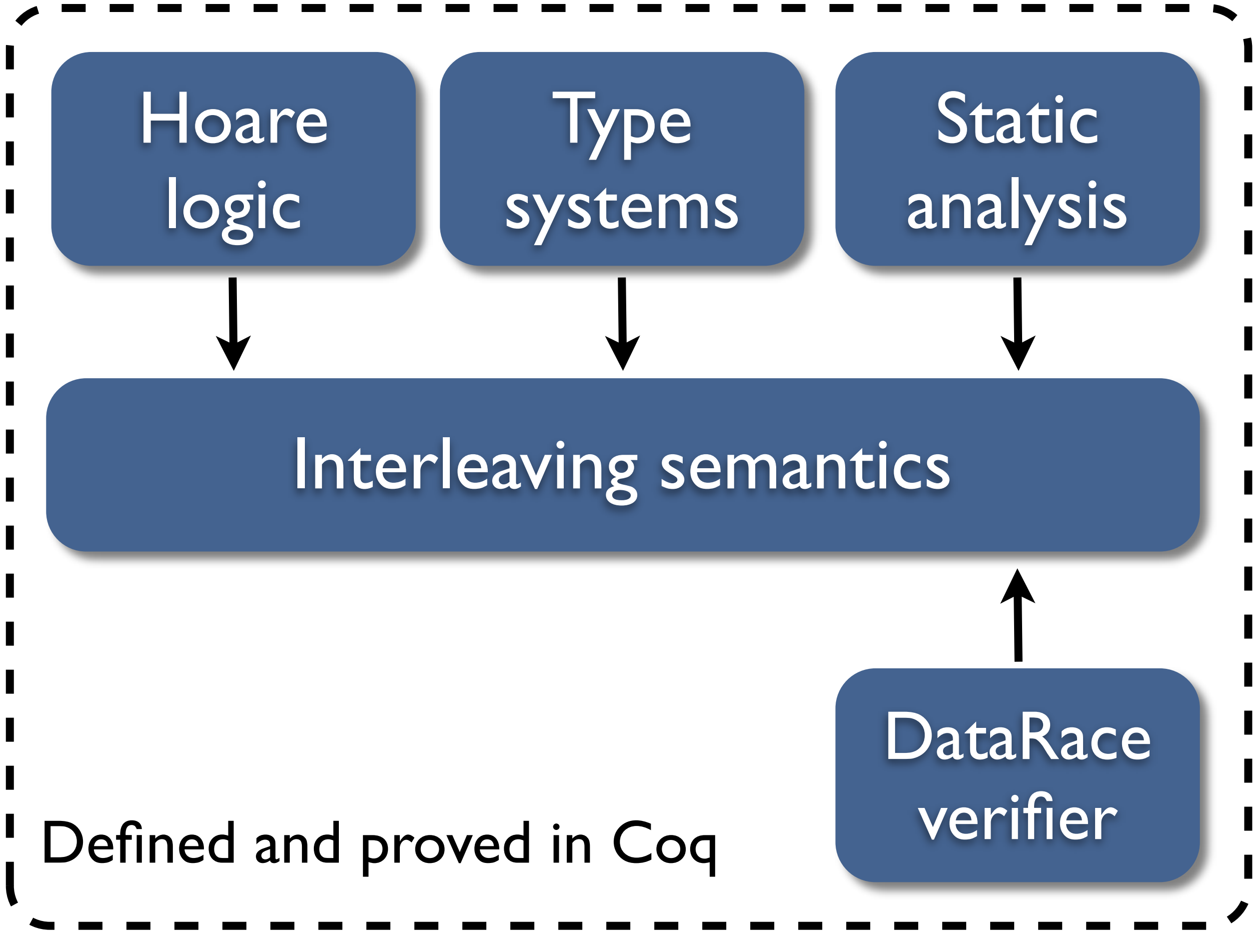
Static analysis

Memory model semantics

Defined and proved in Coq

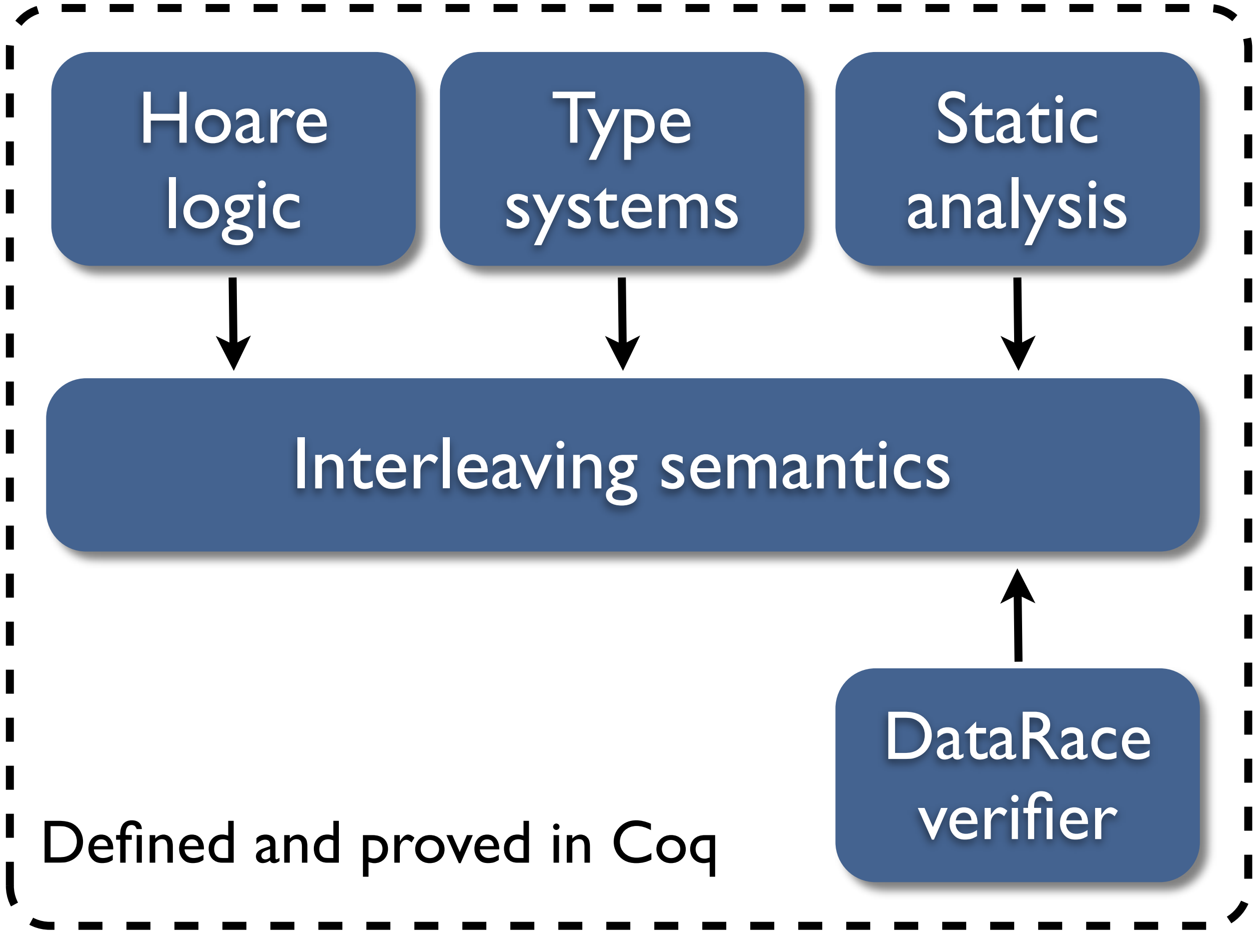
Certified program verification

- There is a growing interest in machine checked semantics proofs
- Program verification framework can be certified in a proof assistant
  - Example : MOBIUS project
  - All component are proved correct
- In a multi-threaded context
  - Using an interleaving semantics is unsound
  - Reasoning directly on the JMM is very painful
  - We need a certified verifier that checks if program are datarace free



Certified program verification

- There is a growing interest in machine checked semantics proofs
- Program verification framework can be certified in a proof assistant
  - Example : MOBIUS project
  - All component are proved correct
- In a multi-threaded context
  - Using an interleaving semantics is unsound
  - Reasoning directly on the JMM is very painful
  - We need a certified verifier that checks if program are datarace free



- A least one good news:
  - The verifier can be proved correct wrt. to an interleaving semantics

This work

- We specify and proved correct in Coq a *state-of-the-art* data race analysis for a representative subset of Java.
  - J. Choi, A. Loginov, and V. Sarkar. *Static datarace analysis for multithreaded object-oriented programs*. Tech. report, IBM Research Division, 2001.
  - M. Naik, A. Aiken, and J. Whaley. *Effective static race detection for java.* PLDI '06
  - M. Naik and A. Aiken. *Conditional must not aliasing for static race detection.* POPL'07
  - M. Naik. *Effective static race detection for java.* PhD thesis, Stanford university, 2008.
- We propose an extensible framework for certified points-to based data race analyses.

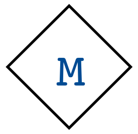
Running example

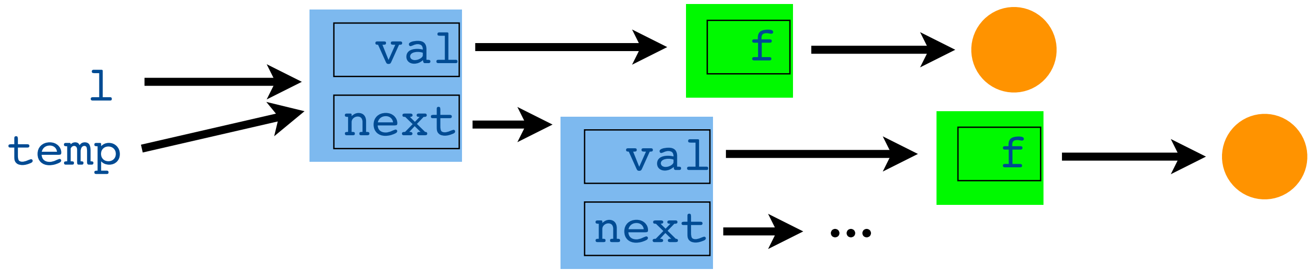
```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Running example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

1. We create a link list `l`

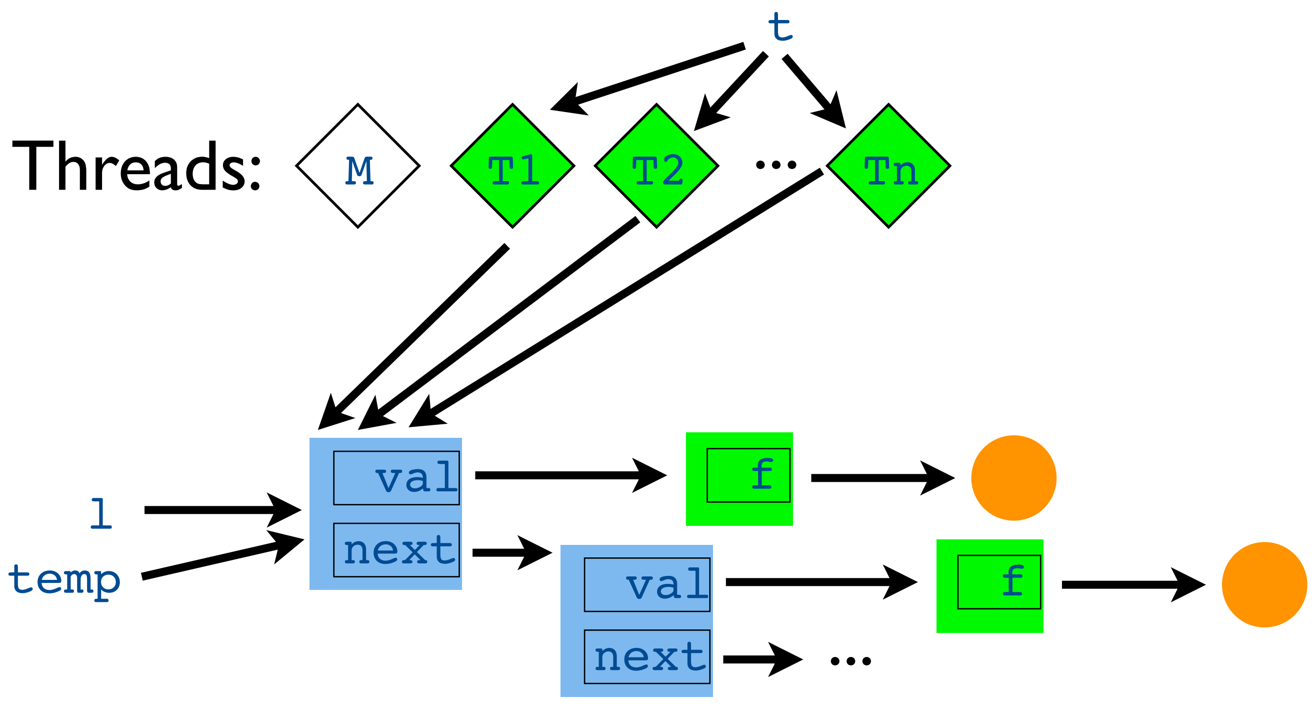
Threads: 



Running example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

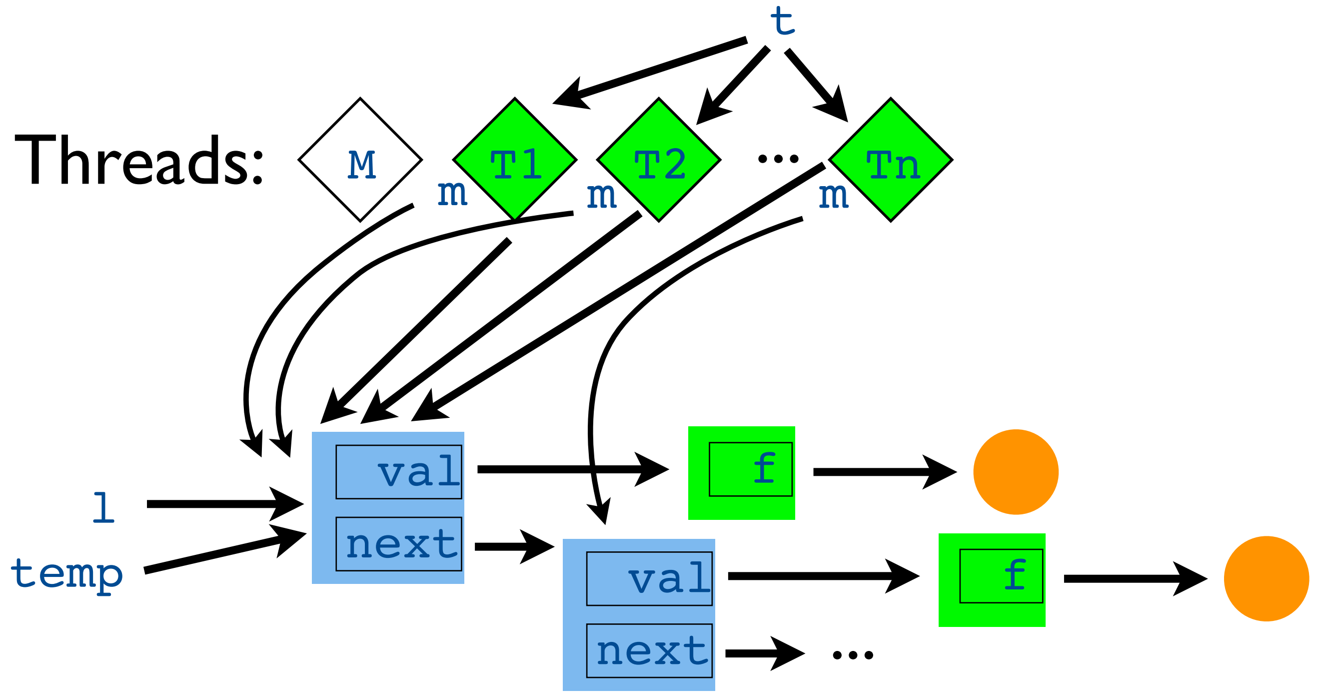
1. We create a link list `l`
2. We create a bunch of thread that all share the list `l`



Running example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

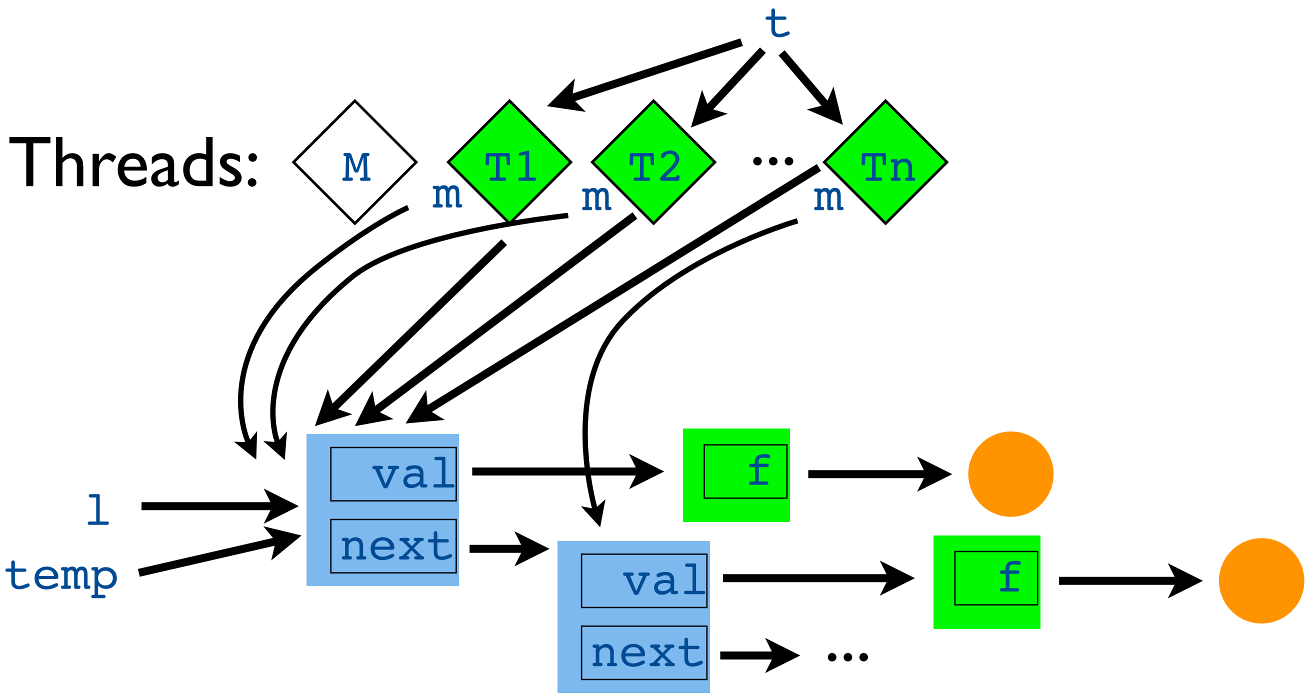
1. We create a link list `l`
2. We create a bunch of thread that all share the list `l`
3. Each thread chooses a cell, takes a lock on it and updates it.



Running example

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

1. We create a link list `l`
2. We create a bunch of thread that all share the list `l`
3. Each thread chooses a cell, takes a lock on it and updates it.



Our Java-like language

- We consider a bytecode language with
  - unstructured control flow,
  - operand stack,
  - objects,
  - virtual method calls,
  - lock and unlock operations for thread synchronisation.

$$\begin{array}{rl} \mathit{inst} ::= & \texttt{aconstnull} \mid \texttt{new}\ \mathit{cid} \mid \texttt{aload}\ x \mid \texttt{astore}\ x \mid \texttt{getfield}\ f \mid \texttt{putfield}\ f \\ & \mid \texttt{areturn} \mid \texttt{return} \mid \texttt{invokevirtual}\ \mathit{mid} : (\mathit{cid}^n)\mathit{rtype} \qquad (n \geq 0) \\ & \mid \texttt{monitorenter} \mid \texttt{monitorexit} \mid \texttt{start} \mid \texttt{ifnd}\ \ell \mid \texttt{goto}\ \ell \end{array}$$

Semantics

- Semantic domains

$$
\begin{array}{rlll}
\mathbb{O} \ni & \ell & & \text{(memory location)} \\
\mathbb{O}_\perp \ni & v & ::= \ell \mid \texttt{Null} & \text{(value)} \\
 & s & ::= v :: s \mid \varepsilon & \text{(operand stack)} \\
\mathbb{V} \to \mathbb{O}_\perp \ni & \rho & & \text{(local variables)} \\
\mathbb{O} \rightharpoonup \mathbb{C}_{id} \times (\mathbb{F} \to \mathbb{O}_\perp) \ni & \sigma & & \text{(heap)} \\
PPT = \mathbb{M} \times \mathbb{N} \ni & ppt & ::= (m, i) & \text{(program point)} \\
CS \ni & cs & ::= (m, i, s, \rho) :: cs \mid \varepsilon & \text{(call stack)} \\
\mathbb{O} \rightharpoonup CS \ni & L & & \text{(thread call stacks)} \\
\mathbb{O} \to ((\mathbb{O} \times \mathbb{N}^*) \cup \{\texttt{free}\}) \ni & \mu & & \text{(locking state)} \\
 & st & ::= (L, \sigma, \mu) & \text{(state)} \\
 & e & ::= \tau \mid (\ell, ?^{ppt}_{f}, \ell') \mid (\ell, !^{ppt}_{f}, \ell') & \text{(event)}
\end{array}
$$

- Transition system

$$
\frac{L\,\ell = cs \qquad L, \ell \vdash (cs, \sigma, \mu) \xrightarrow{e} (L', \sigma', \mu')}{(L, \sigma, \mu) \xrightarrow{e} (L', \sigma', \mu')}
$$

Semantics

- Transition rules (excerpt)

$$\frac{\begin{array}{c}(m.\texttt{body})\ i = \texttt{new}\ c_{id} \quad \ell' \notin dom(\sigma) \\ L' = L[\ell \mapsto (m, i+1, \ell' :: s, \rho) :: cs]\end{array}}{L, \ell \vdash ((m, i, s, \rho) :: cs, \sigma, \mu) \xrightarrow{\tau} (L', \sigma[\ell' \mapsto new(c_{id})], \mu)}$$

$$\frac{\begin{array}{c}(m.\texttt{body})\ i = \texttt{start} \qquad \neg(\ell' \in dom(L)) \\ Lookup\ (run : ()\texttt{void})\ \texttt{class}(\sigma, \ell') = m_1 \qquad \rho_1 = [0 \mapsto \ell'] \\ L' = L[\ell \mapsto (m, i+1, s', \rho) :: cs, \ell' \mapsto (m_1, 0, \varepsilon, \rho_1) :: \varepsilon]\end{array}}{L, \ell \vdash ((m, i, \ell' :: s', \rho) :: cs, \sigma, \mu) \rightarrow (L', \sigma, \mu)}$$

$$\frac{\begin{array}{c}(m.\texttt{body})\ i = \texttt{monitorenter} \qquad \mu\ \ell' \in \{\texttt{free}, (\ell, n)\} \qquad \mu' = acquire\ \ell\ \ell'\ \mu \\ L' = L[\ell \mapsto (m, i+1, s, \rho) :: cs]\end{array}}{L, \ell \vdash ((m, i, \ell' :: s, \rho) :: cs, \sigma, \mu) \rightarrow (L', \sigma, \mu')}$$

- Races

$$\frac{st \in \mathrm{ReachableStates}(P) \qquad st \xrightarrow{\ell_1 !_f^{ppt_1} \ell_0} st_1 \qquad st \xrightarrow{\ell_2 \mathcal{R} \ell_0} st_2 \qquad \mathcal{R} \in \{?_f^{ppt_2}, !_f^{ppt_2}\} \qquad \ell_1 \neq \ell_2}{Race(P, ppt_1, f, ppt_2)}$$

Data Race Analysis

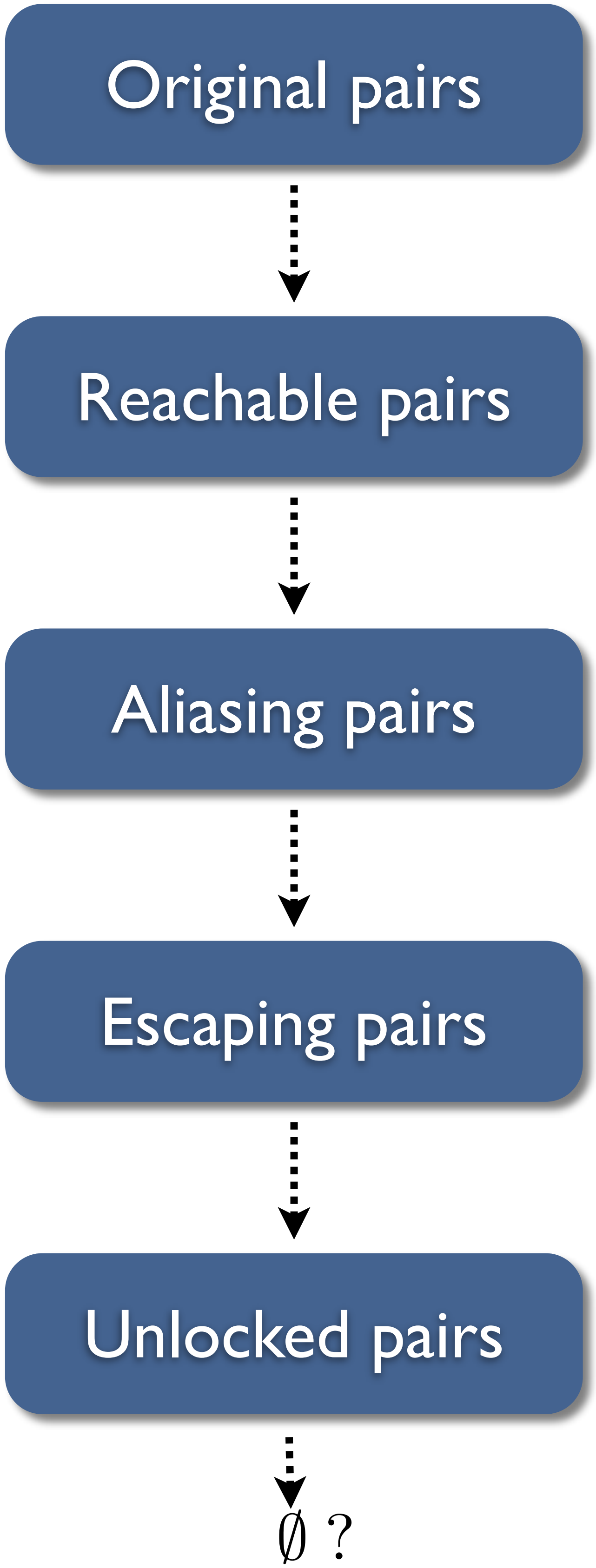
- We start from a large set of all potential race pairs.
- We successively remove pairs that are proved to be false races.
- Each potential races sets are proved sound:

$$\begin{aligned}&\forall (ppt_1, f, ppt_2),\\ &\qquad Race(P, ppt_1, f, ppt_2) \Rightarrow\\ &\qquad (ppt_1, f, ppt_2) \in PotentialRacePairs(P)\end{aligned}$$

Data Race Analysis

- We start from a large set of all potential race pairs.
- We successively remove pairs that are proved to be false races.
- Each potential races sets are proved sound:

$$\begin{aligned}&\forall (ppt_1, f, ppt_2),\\ &\qquad Race(P, ppt_1, f, ppt_2) \Rightarrow\\ &\qquad (ppt_1, f, ppt_2) \in PotentialRacePairs(P)\end{aligned}$$



Original pairs

Reachable pairs

Aliasing pairs

Escaping pairs

Unlocked pairs

$\emptyset$?

Original pairs

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Java’s strong typing dictates that a pair of accesses may be involved in a race only if both access the same field.

Here :

Original pairs

```
class List{ T val; List next; }

class Main() {
 void main(){
  List l = null;
  while (*) {
   List temp = new List();
1: temp.val = new T();
2: temp.val.f = new A();
3: temp.next = l;
   l = temp }
  while (*) {
   T t = new T();
4: t.data = l;
   t.start();
5: t.f = ...;}
  return;
 }
}

class T {
 A f;
 List data;
 void run(){
  while(*){
6: List m = this.data;
7: while (*) { m = m.next; }
8: synchronized(m){ m.val.f = ...;}}
  return;}}
```

Java’s strong typing dictates that a pair of accesses may be involved in a race only if both access the same field.

Here : we start with 13 potential races.

```
(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(4,data,4),(5,f,5), (2,f,5),

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)
```

Original pairs

```
class List{ T val; List next; }

class Main() {
 void main(){
  List l = null;
  while (*) {
   List temp = new List();
1: temp.val = new T();
2: temp.val.f = new A();
3: temp.next = l;
   l = temp }
  while (*) {
   T t = new T();
4: t.data = l;
   t.start();
5: t.f = ...;}
  return;
  }
}

class T {
 A f;
 List data;
 void run(){
  while(*){
6: List m = this.data;
7: while (*) { m = m.next; }
8: synchronized(m){ m.val.f = ...;}}
  return;}}
```

Java's strong typing dictates that a pair of accesses may be involved in a race only if both access the same field.

Here : we start with 13 potential races.

```
(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(4,data,4),(5,f,5), (2,f,5),

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)
```

Original pairs

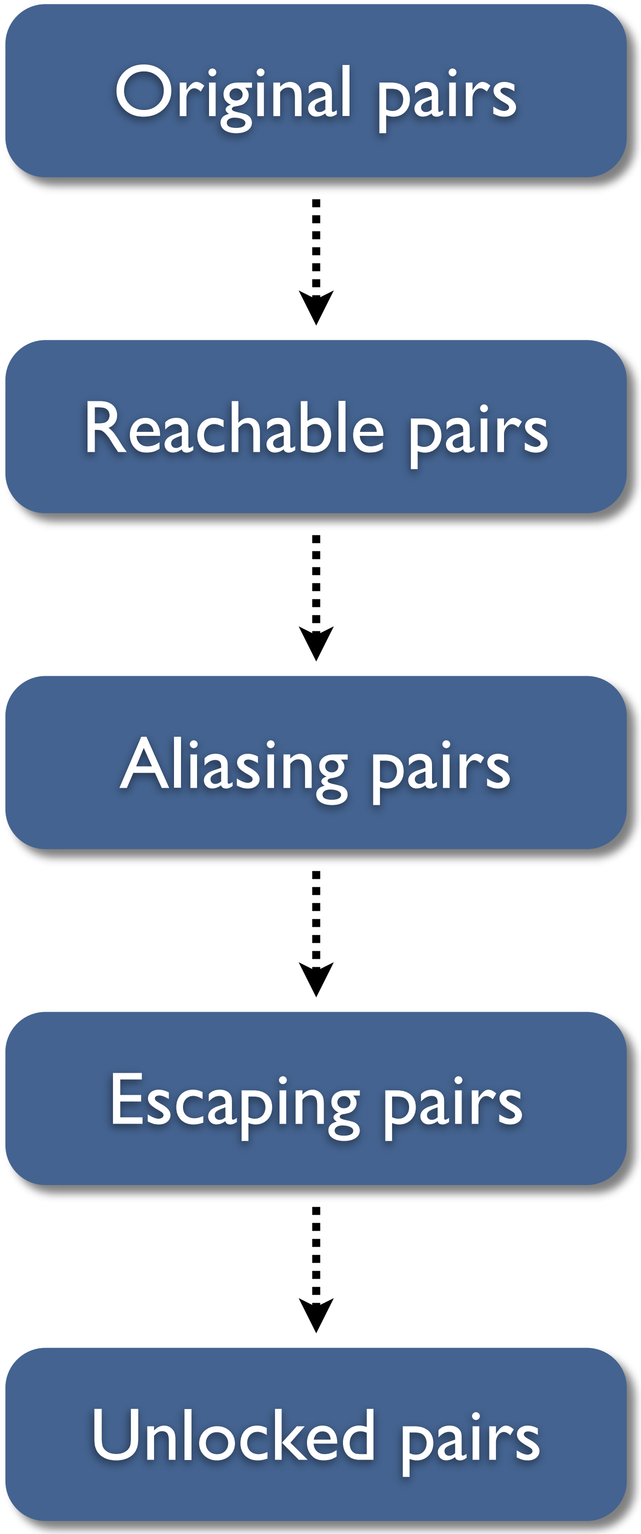
```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Java's strong typing dictates that a pair of accesses may be involved in a race only if both access the same field.

Here : we start with 13 potential races.

```
(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(4,data,4),(5,f,5), (2,f,5),

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)
```

Data Race Analysis



Original pairs

Reachable pairs

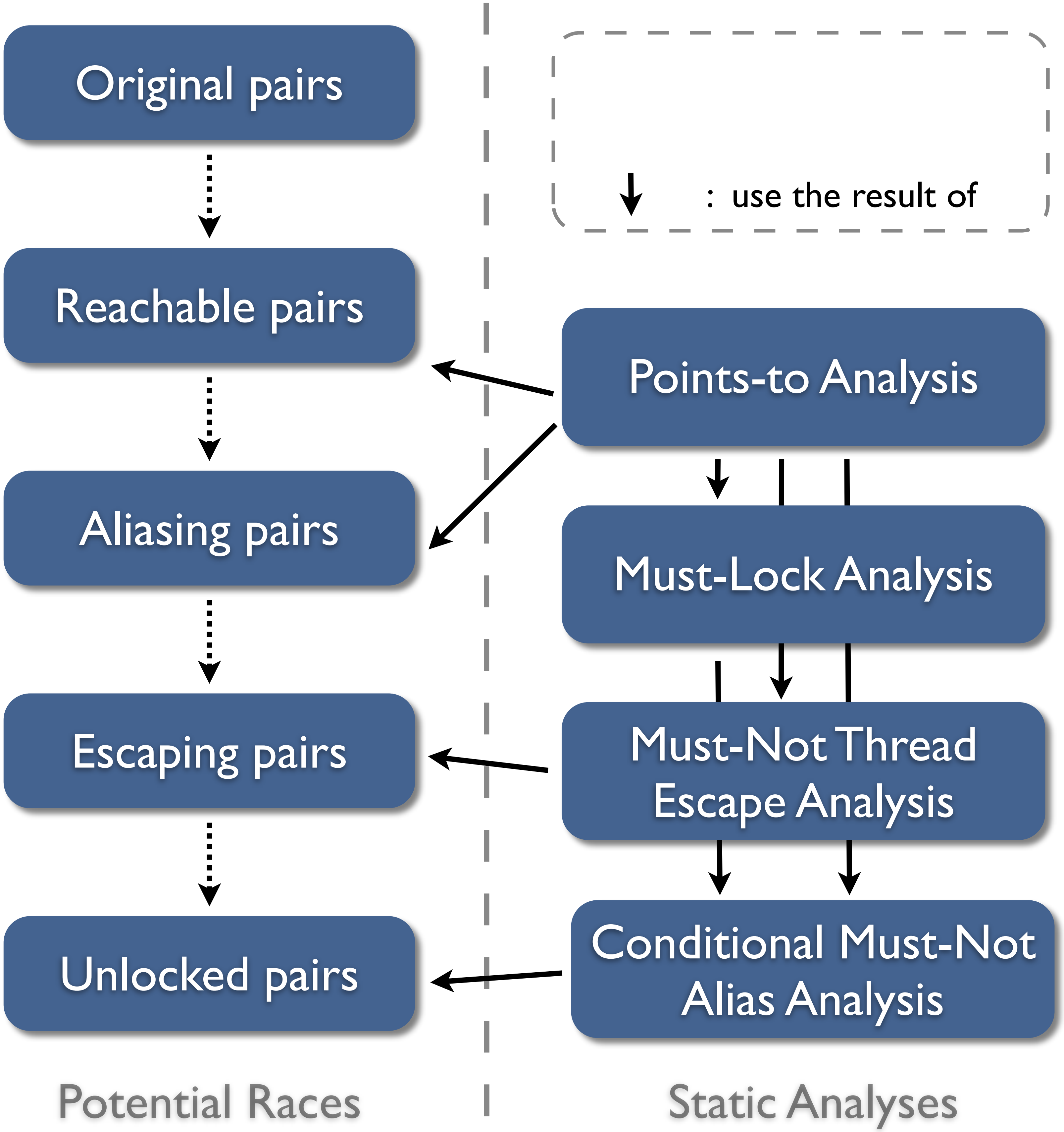
Aliasing pairs

Escaping pairs

Unlocked pairs

Potential Races

Data Race Analysis



Original pairs

Reachable pairs

Aliasing pairs

Escaping pairs

Unlocked pairs

Potential Races

↓ : use the result of

Points-to Analysis

Must-Lock Analysis

Must-Not Thread Escape Analysis

Conditional Must-Not Alias Analysis

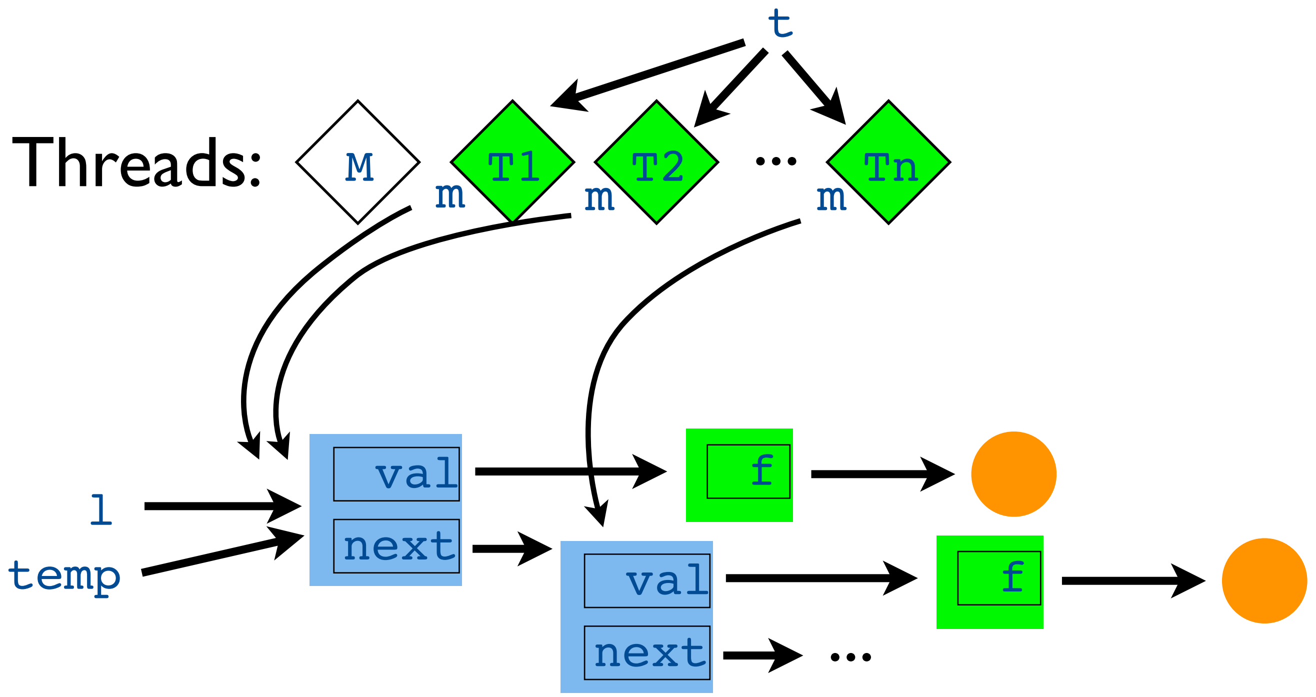
Static Analyses

Points-to analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site

```
(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(4,data,4),(5,f,5), (2,f,5),

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)
```

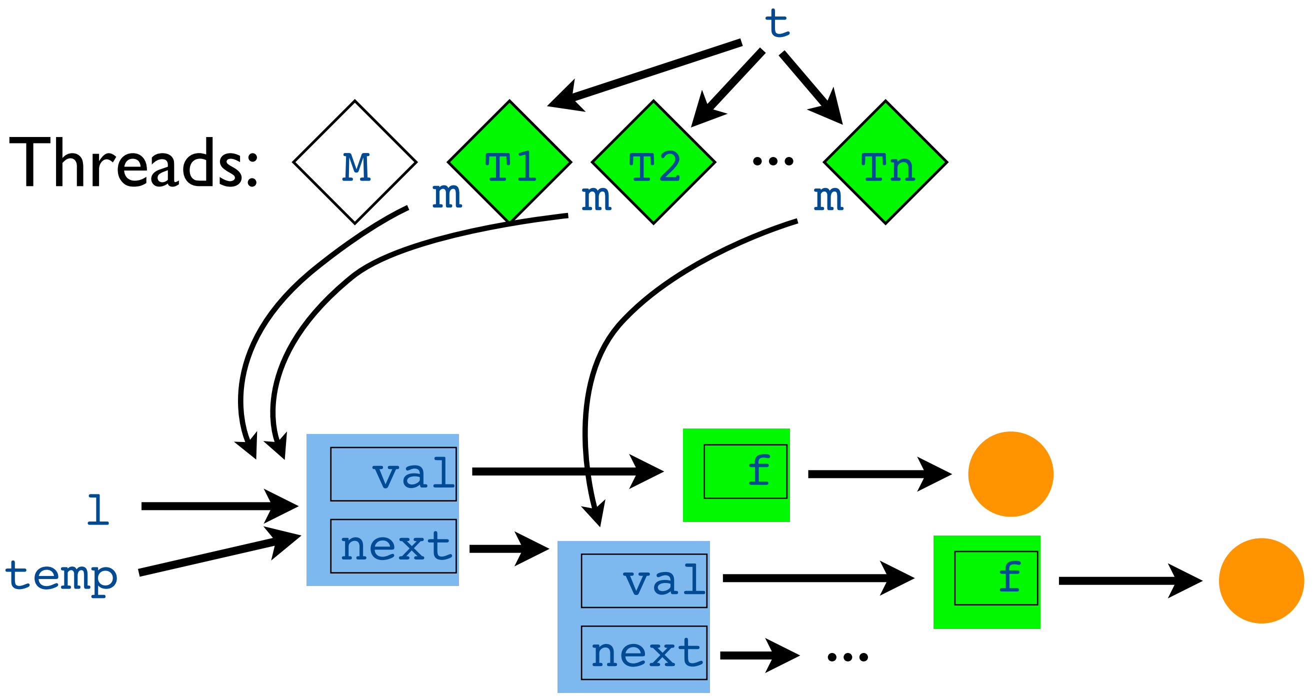


Points-to analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 List temp = new List();
1:  h2 temp.val = new T();
2:  h3 temp.val.f = new A();
3:       temp.next = l;
         l = temp }
    while (*) {
    h4 T t = new T();
4:       t.data = l;
         t.start();
5:       t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:     List m = this.data;
7:     while (*) { m = m.next; }
8:     synchronized(m){ m.val.f = ...;}}
    return;}}
```

Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site

```
(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(4,data,4),(5,f,5), (2,f,5),
(5,f,8),
(4,data,6),(3,next,7),(1,val,8),(2,f,8),
(8,f,8)
```



Points-to analysis

```
class List{ T val; List next; }

class Main() {
 void main(){
  List l = null;
  while (*) {
  h1 List temp = new List();
1: h2 temp.val = new T();
2: h3 temp.val.f = new A();
3:     temp.next = l;
       l = temp }
  while (*) {
  h4 T t = new T();
4:     t.data = l;
       t.start();
5:     t.f = ...;}
  return;
 }
}

class T {
 A f;
 List data;
 void run(){
  while(*){
6:     List m = this.data;
7:     while (*) { m = m.next; }
8:     synchronized(m){ m.val.f = ...;}}
  return;}}
```

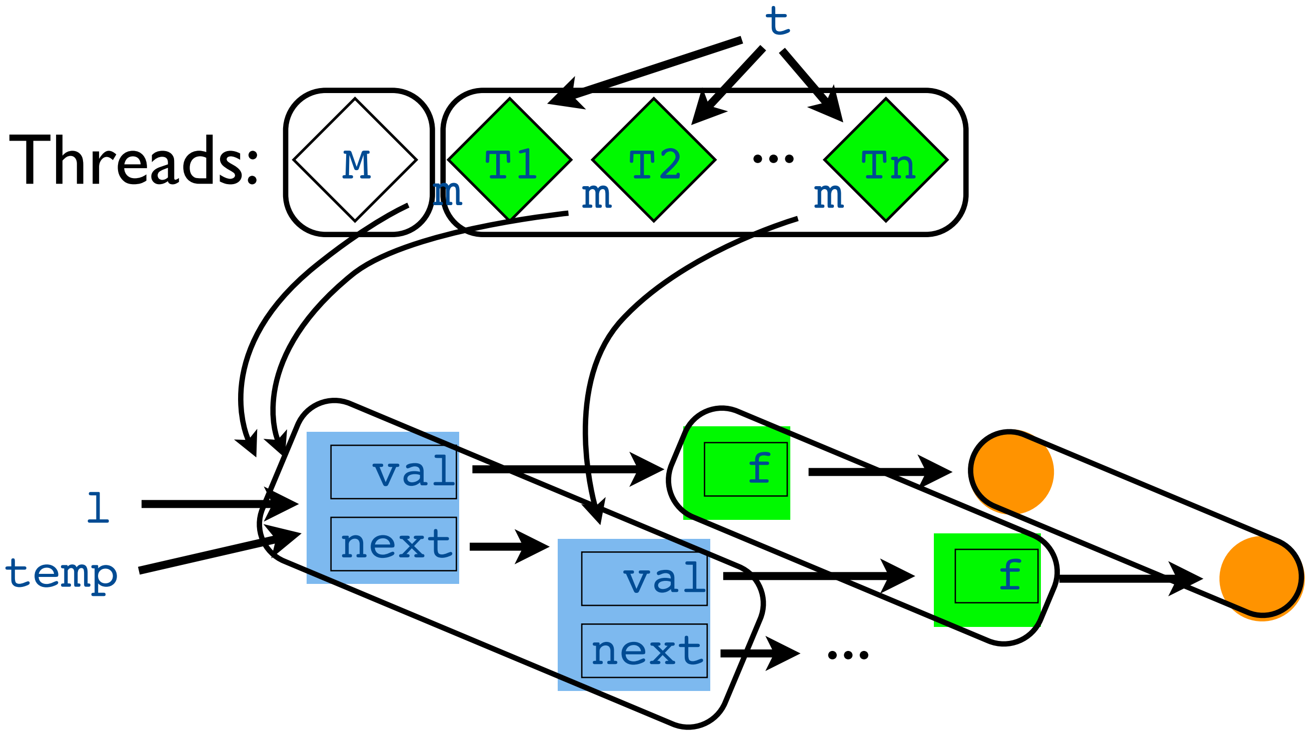
Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site

(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(4,data,4),(5,f,5), (2,f,5),

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)

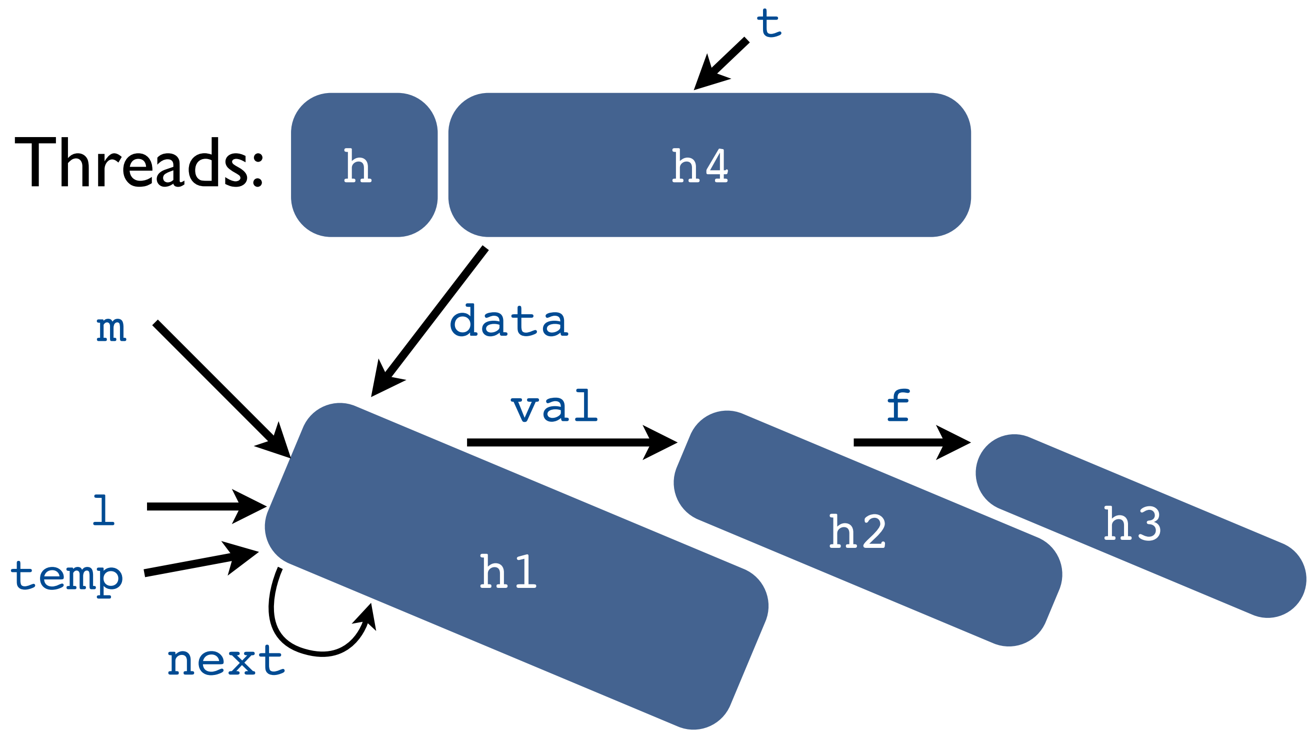


Points-to analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 List temp = new List();
1:  h2 temp.val = new T();
2:  h3 temp.val.f = new A();
3:     temp.next = l;
       l = temp }
    while (*) {
    h4 T t = new T();
4:     t.data = l;
       t.start();
5:     t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:     List m = this.data;
7:     while (*) { m = m.next; }
8:     synchronized(m){ m.val.f = ...;}}
    return;}}
```

Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site

```
(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(4,data,4),(5,f,5), (2,f,5),

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)
```



Points-to analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 List temp = new List();
1:  h2 temp.val = new T();
2:  h3 temp.val.f = new A();
3:     temp.next = l;
       l = temp }
    while (*) {
    h4 T t = new T();
4:     t.data = l;
       t.start();
5:     t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:     List m = this.data;
7:     while (*) { m = m.next; }
8:     synchronized(m){ m.val.f = ...;}}
    return;}}
```

For all these potential races, all accesses correspond to a same thread.

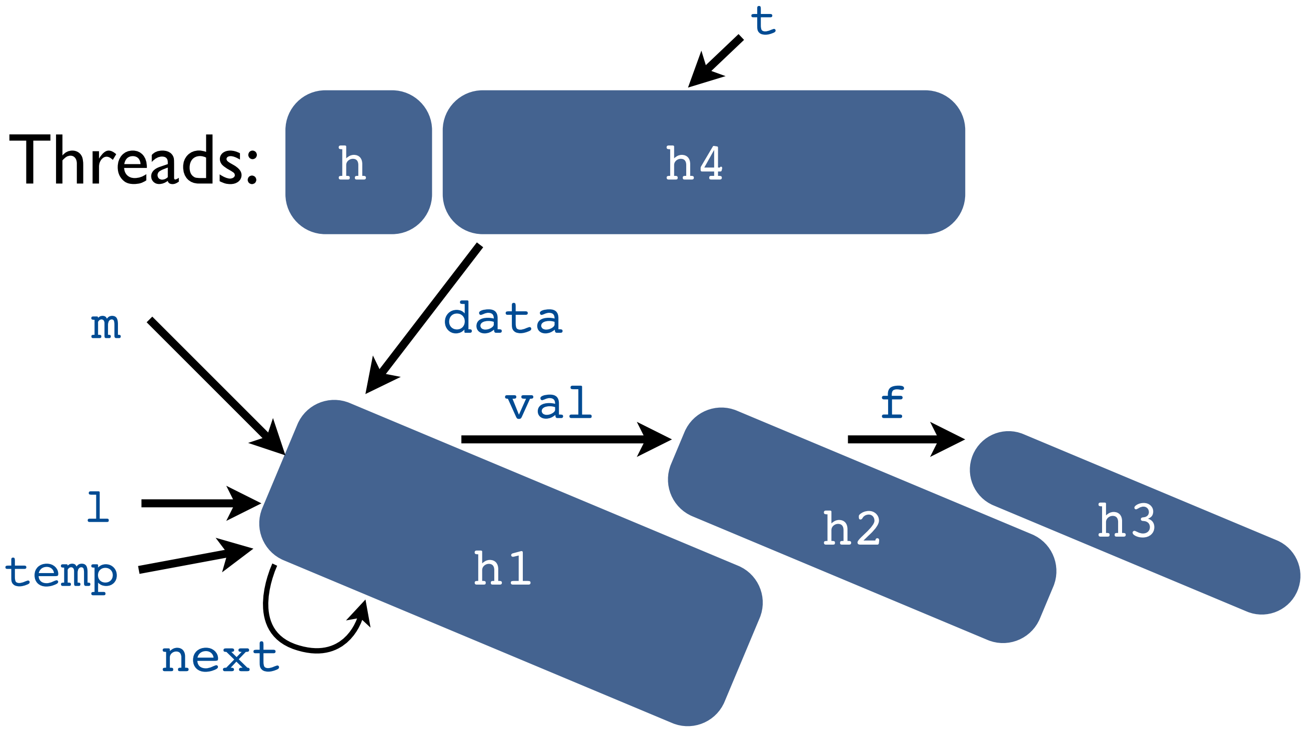
- `h` is a *single-instance* allocation site

`(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),`
`(4,data,4),(5,f,5), (2,f,5),`

`(5,f,8),`

`(4,data,6),(3,next,7),(1,val,8),(2,f,8),`

`(8,f,8)`



Points-to analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 List temp = new List();
1:  h2 temp.val = new T();
2:  h3 temp.val.f = new A();
3:     temp.next = l;
       l = temp }
    while (*) {
    h4 T t = new T();
4:     t.data = l;
       t.start();
5:     t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:     List m = this.data;
7:     while (*) { m = m.next; }
8:     synchronized(m){ m.val.f = ...;}}
    return;}}
```

For all these potential races, all accesses correspond to a same thread.

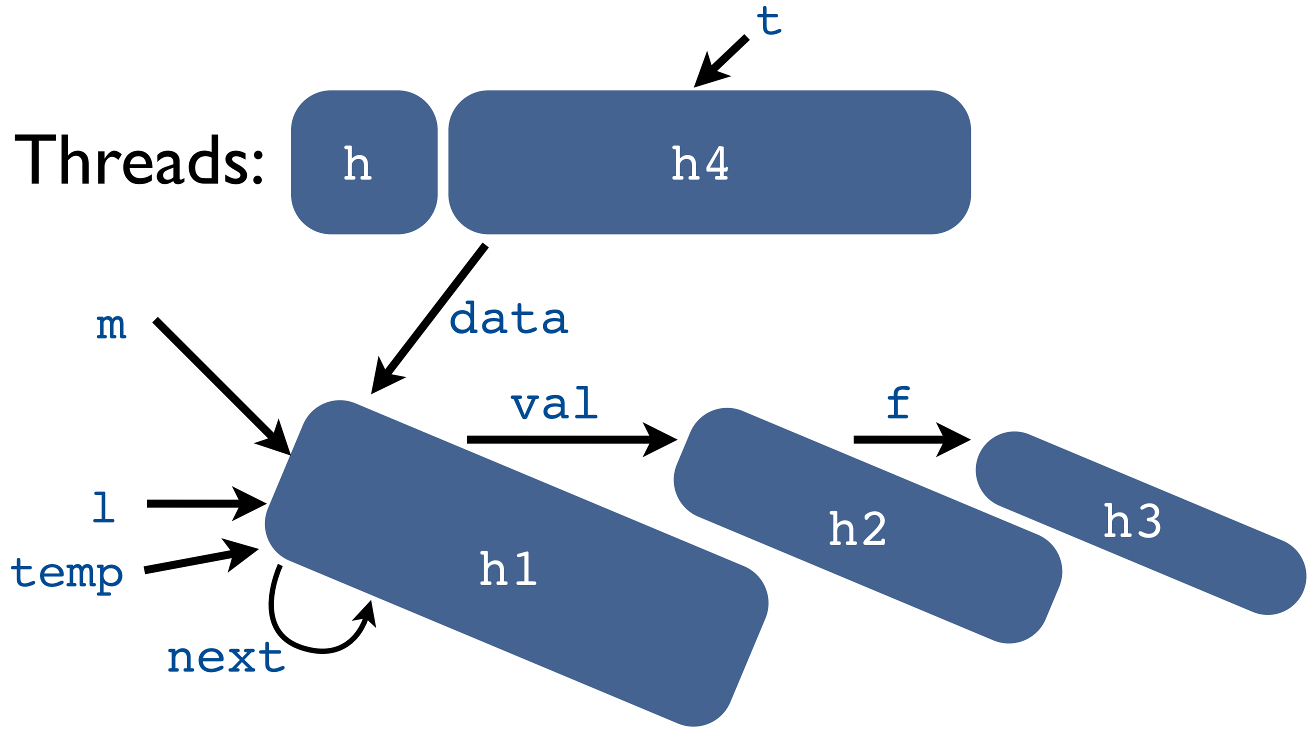
- `h` is a *single-instance* allocation site

~~(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),~~
~~(4,data,4),(5,f,5), (2,f,5),~~

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)



Points-to analysis

```
     class List{ T val; List next; }

     class Main() {
       void main(){
         List l = null;
         while (*) {
         h1 List temp = new List();
1:       h2 temp.val = new T();
2:       h3 temp.val.f = new A();
3:          temp.next = l;
            l = temp }
         while (*) {
         h4 T t = new T();
4:          t.data = l;
            t.start();
5:          t.f = ...;}
         return;
        }
     }

     class T {
       A f;
       List data;
       void run(){
         while(*){
6:          List m = this.data;
7:          while (*) { m = m.next; }
8:          synchronized(m){ m.val.f = ...;}}
         return;}}
```

For all these potential races, accesses correspond to different locations.

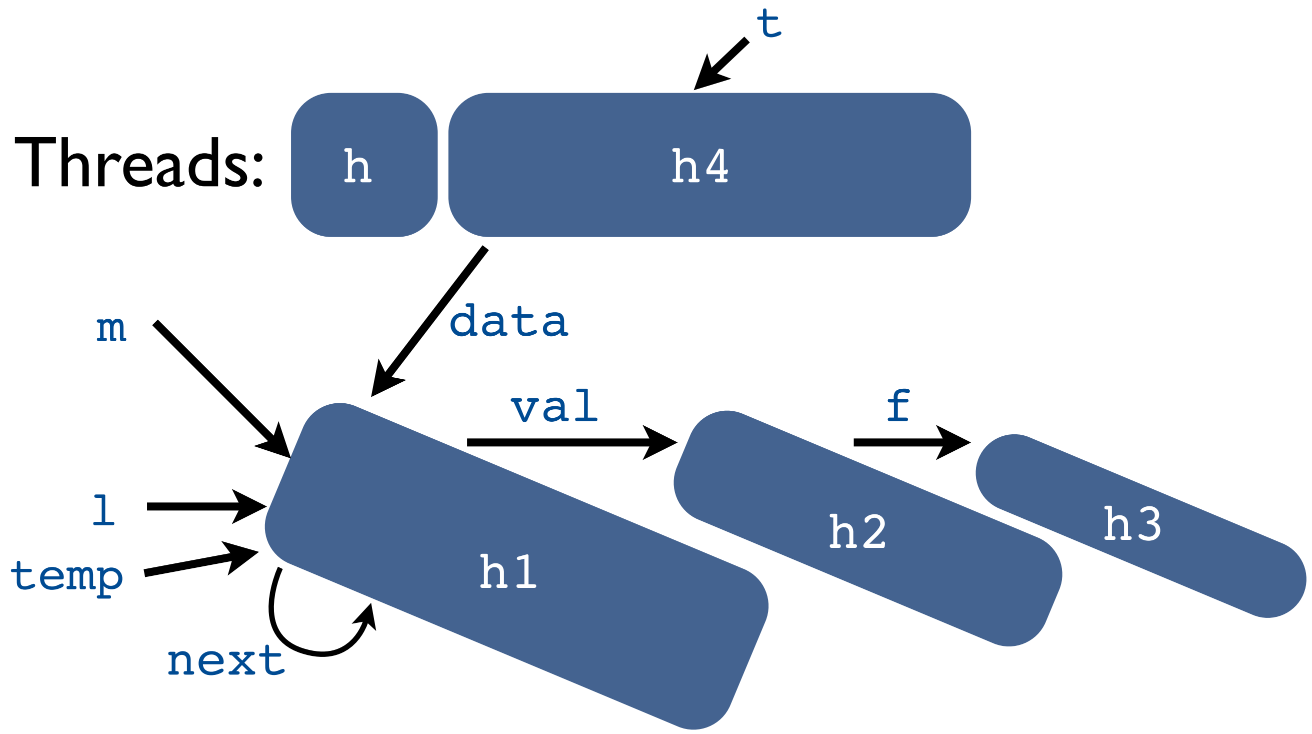
- `t` points-to `h4`
- `m.val` points to `h2`

~~(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),~~
~~(4,data,4),(5,f,5), (2,f,5),~~

(5,f,8),

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)



Points-to analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
h1  List temp = new List();
1:  h2  temp.val = new T();
2:  h3  temp.val.f = new A();
3:      temp.next = l;
        l = temp }
    while (*) {
h4  T t = new T();
4:      t.data = l;
        t.start();
5:      t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:      List m = this.data;
7:      while (*) { m = m.next; }
8:      synchronized(m){ m.val.f = ...;}}
    return;}}
```

For all these potential races, accesses correspond to different locations.

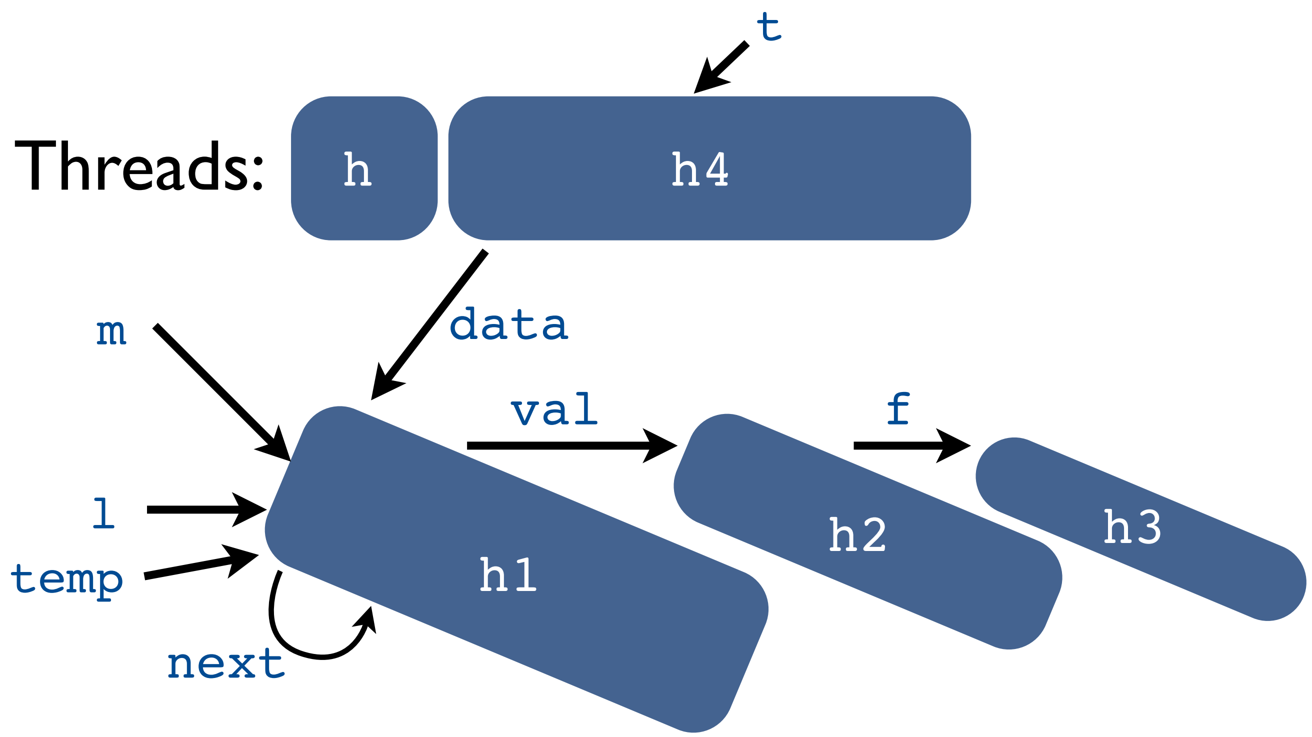
- `t` points-to `h4`
- `m.val` points to `h2`

~~(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),~~
~~(4,data,4),(5,f,5), (2,f,5),~~

~~(5,f,8),~~

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)



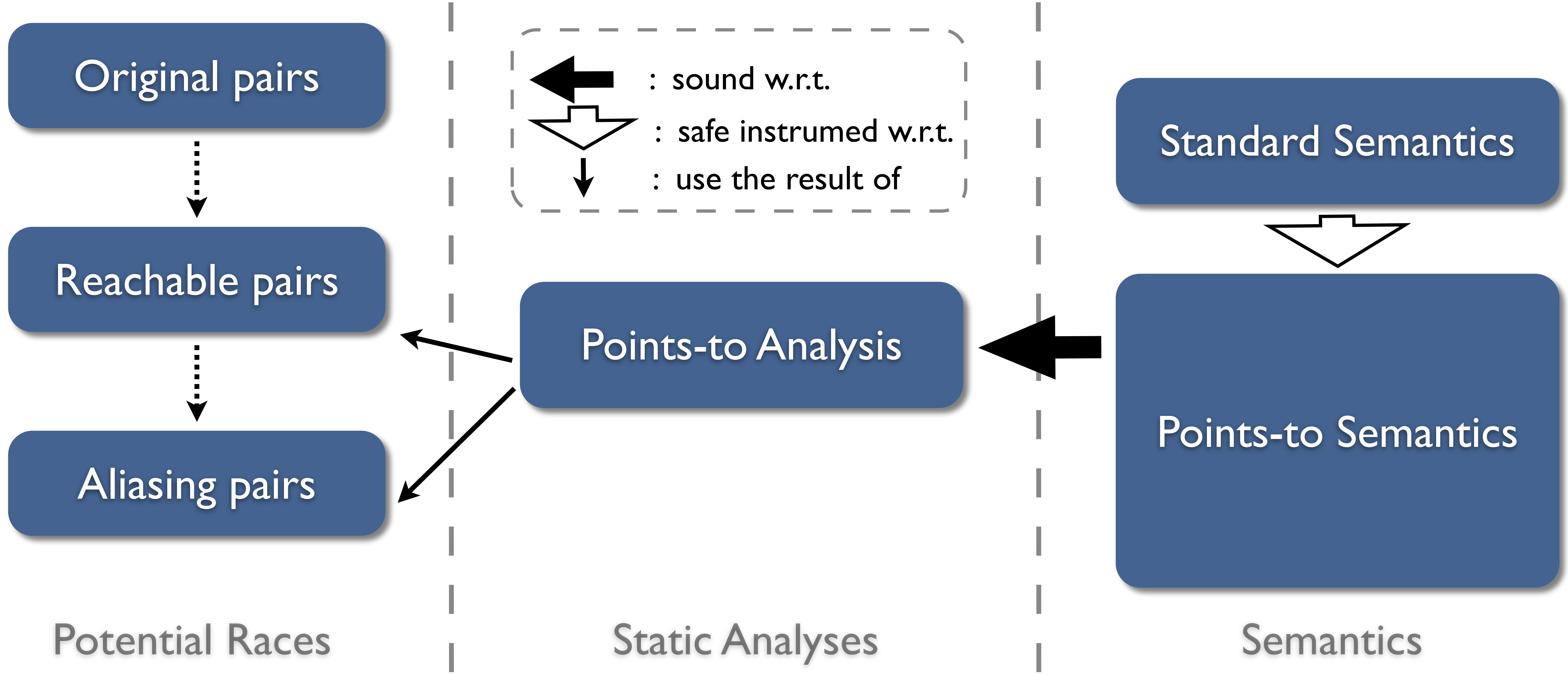
Points-to analysis in Coq

The analysis is parameterized by an abstract notion of *context* which captures a large variety of points-to context.

```
Module Type CONTEXT.

  Parameter pcontext : Set. (* pointer context *)
  Parameter mcontext : Set. (* method context *)

  Parameter make_new_context : method -> line -> classId -> mcontext -> pcontext.
  Parameter make_call_context : method -> line -> mcontext -> pcontext -> mcontext.
  Parameter get_class : program -> pcontext -> option classId.

  Parameter class_make_new_context : forall p m i cid c,
    body m i = Some (New cid) ->
    get_class p (make_new_context m i cid c) = Some cid.

  Parameter init_mcontext : mcontext.
  Parameter init_pcontext : pcontext.

  Parameter eq_pcontext : forall c1 c2:pcontext, {c1=c2}+{c1<>c2}.
  Parameter eq_mcontext : forall c1 c2:mcontext, {c1=c2}+{c1<>c2}.

End CONTEXT.
```

Points-to analysis in Coq

We prove the soundness of the analysis with respect to an instrumented *points-to semantics*.



Must-Not Thread Escape analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
h1  List temp = new List();
1:  h2  temp.val = new T();
2:  h3  temp.val.f = new A();
3:      temp.next = l;
        l = temp }
    while (*) {
h4  T t = new T();
4:      t.data = l;
        t.start();
5:      t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

For all these potential races, the main thread access location that are not (yet) shared

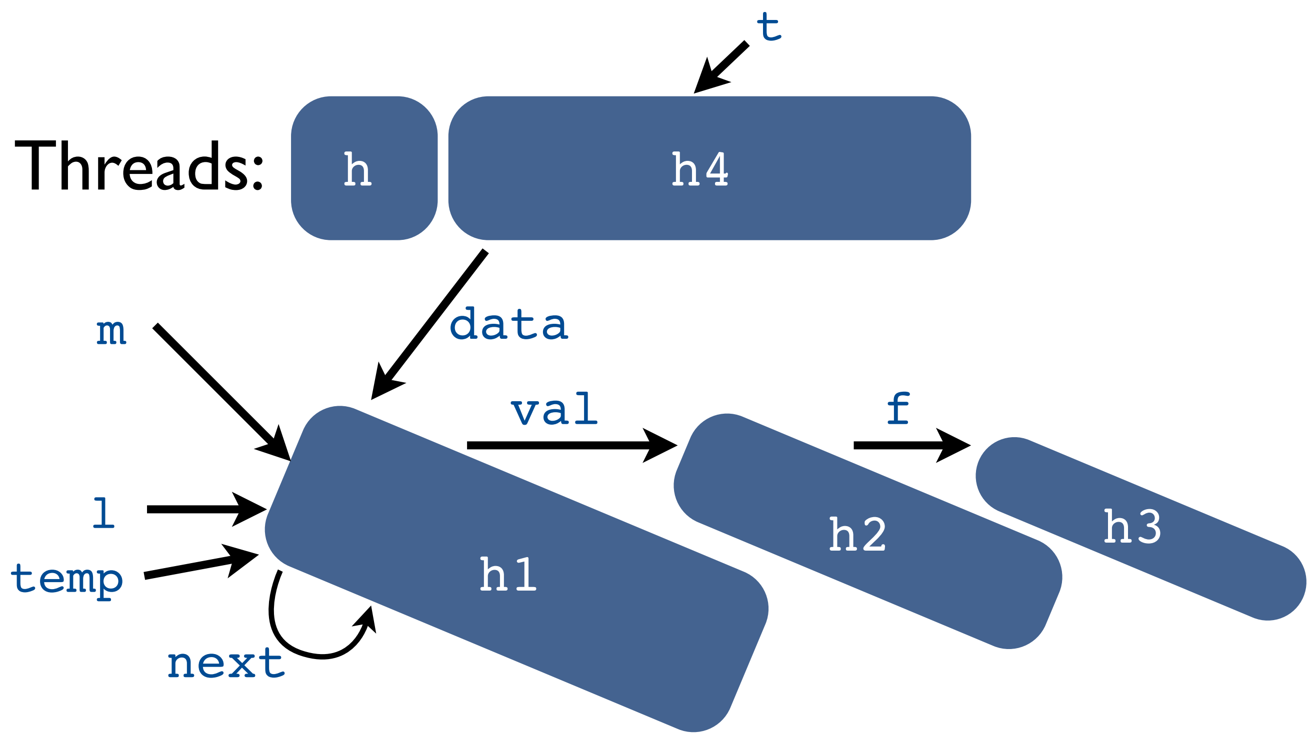
- We uses a flow sensitive thread-escape analysis
- The analysis is *iteration* sensitive

~~(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),~~
~~(4,data,4),(5,f,5), (2,f,5),~~

~~(5,f,8),~~

(4,data,6),(3,next,7),(1,val,8),(2,f,8),

(8,f,8)



Must-Not Thread Escape analysis

```
class List{ T val; List next; }

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 List temp = new List();
1:  h2 temp.val = new T();
2:  h3 temp.val.f = new A();
3:     temp.next = l;
       l = temp }
    while (*) {
    h4 T t = new T();
4:     t.data = l;
       t.start();
5:     t.f = ...;}
    return;
   }
}

class T {
  A f;
  List data;
  void run(){
    while(*){
6:     List m = this.data;
7:     while (*) { m = m.next; }
8:     synchronized(m){ m.val.f = ...;}}
    return;}}
```

For all these potential races, the main thread access location that are not (yet) shared

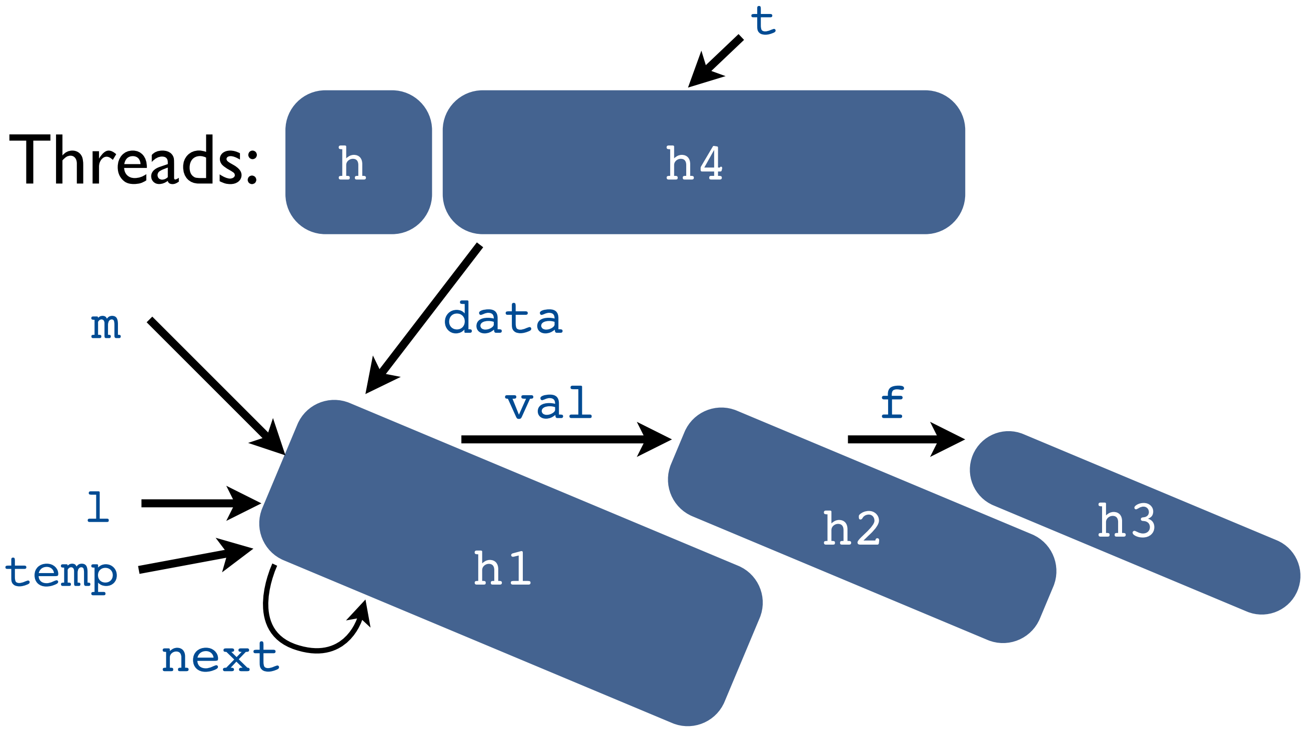
- We uses a flow sensitive thread-escape analysis
- The analysis is *iteration* sensitive

~~(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),~~
~~(4,data,4),(5,f,5), (2,f,5),~~

~~(5,f,8),~~

~~(4,data,6),(3,next,7),(1,val,8),(2,f,8),~~

(8,f,8)



The last one...

```
synchronize(m){ m.val.f = ...;}  ‖  synchronize(m){ m.val.f = ...;}
```

The last one...

```
synchronize(m){ m.val.f = ...;}  ∥  synchronize(m){ m.val.f = ...;}
```

- If the two threads lock the same location OK

The last one...

```
synchronize(m){ m.val.f = ...;}  ‖  synchronize(m){ m.val.f = ...;}
```

- If the two threads lock the same location OK
- If the two threads lock different locations, we must prove that they access different location with `m.val`

The last one...

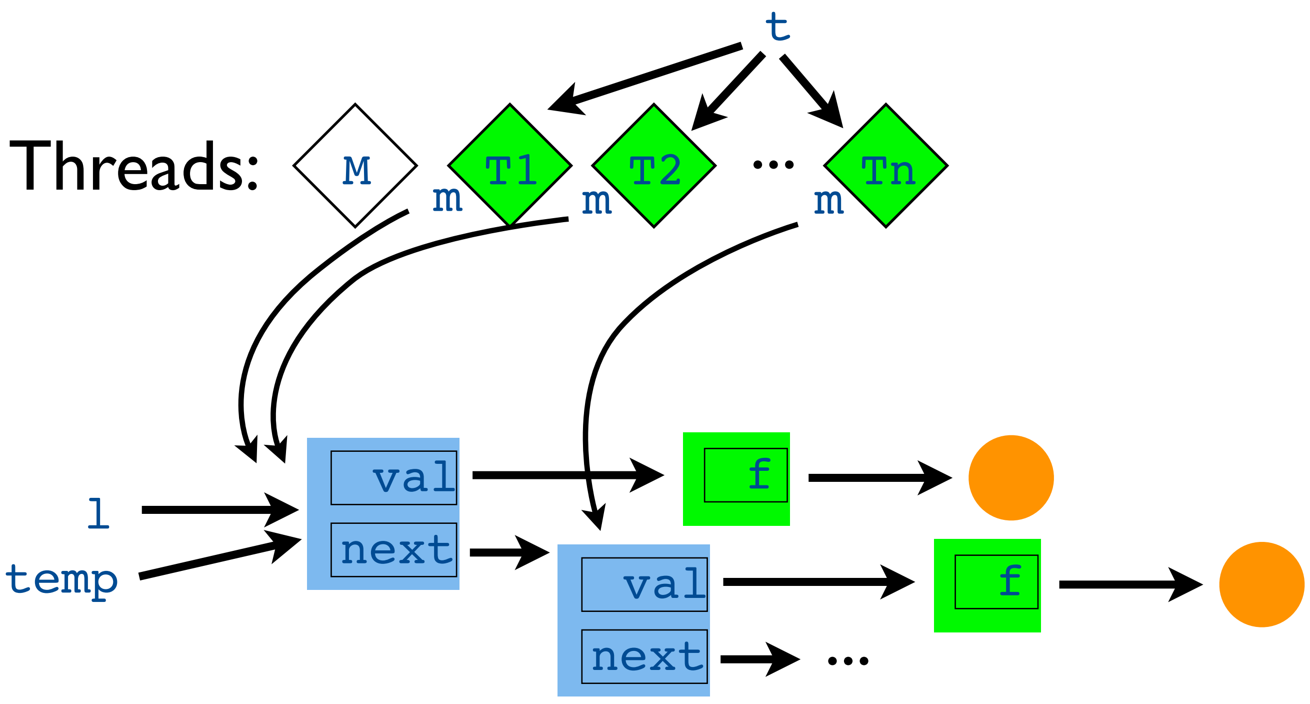
```
synchronize(m){ m.val.f = ...;}    ||    synchronize(m){ m.val.f = ...;}
```

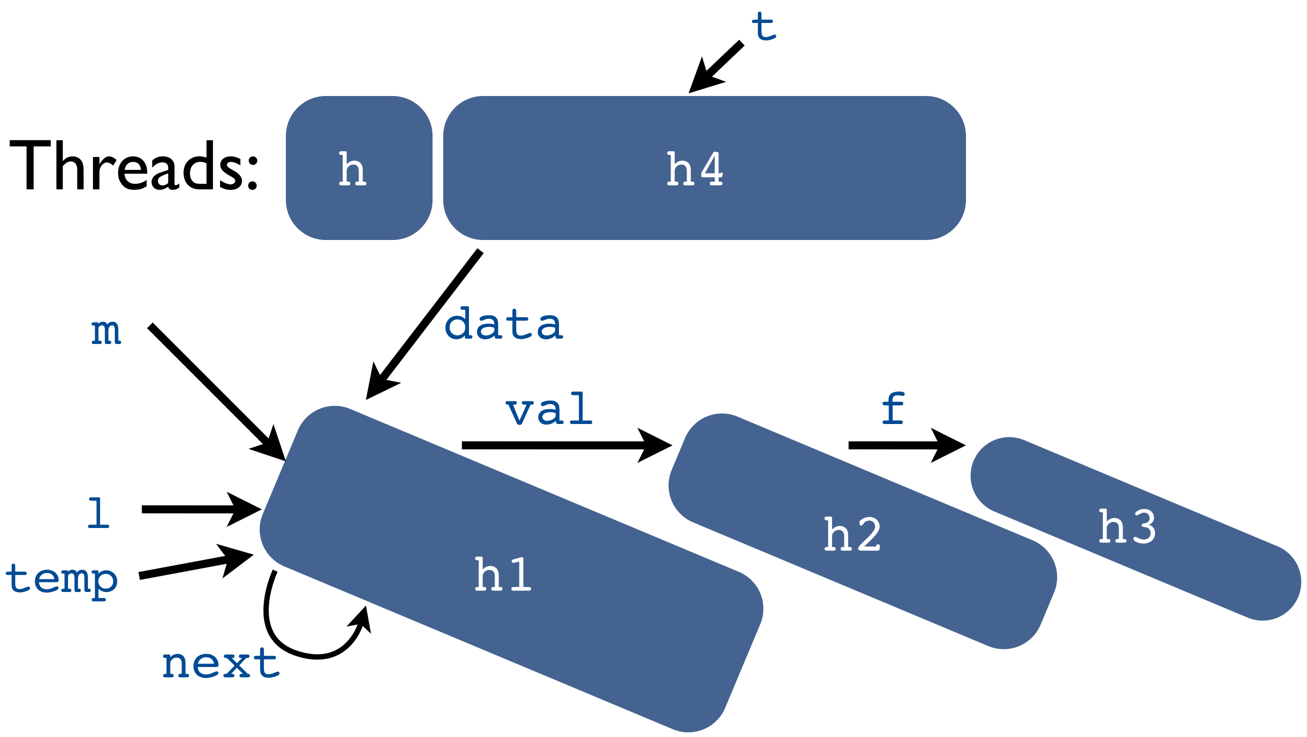
- If the two threads lock the same location OK
- If the two threads lock different locations, we must prove that they access different location with `m.val`
- Disjoint Reachability: $h \in DR_{Paths}(H)$ *for* H *a set of allocation sites, if and only if whenever an object* o *allocated at site* h *may be reachable by a field path in set* $Paths$ *from two objects* o_1 *and* o_2 *allocated at any sites in* H*, then* o_1 *and* o_2 *are one and the same object.*

Disjoint Reachability: example

- Disjoint Reachability: $h \in DR_{Paths}(H)$ *for* H *a set of allocation sites, if and only if whenever an object* o *allocated at site* h *may be reachable by a field path in set* $Paths$ *from two objects* o_1 *and* o_2 *allocated at any sites in* H*, then* o_1 *and* o_2 *are one and the same object.*

$$DR_{\{[\texttt{val}]\}}(\{h_1\}) = \ ?$$

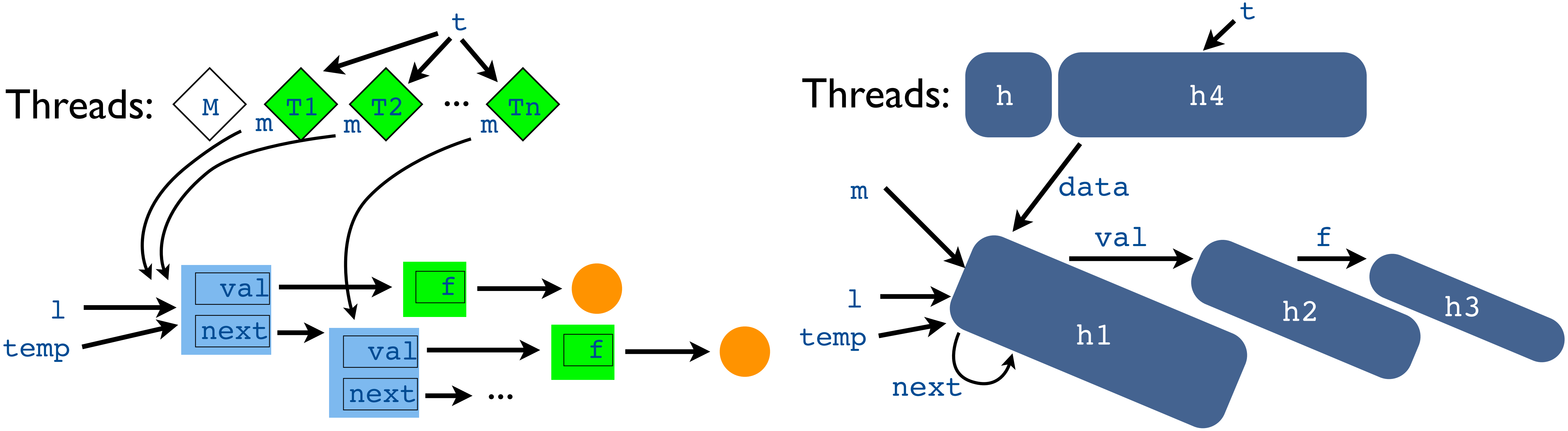




Disjoint Reachability: example

- Disjoint Reachability: $h \in DR_{Paths}(H)$ *for* H *a set of allocation sites, if and only if whenever an object* o *allocated at site* h *may be reachable by a field path in set* $Paths$ *from two objects* o_1 *and* o_2 *allocated at any sites in* H*, then* o_1 *and* o_2 *are one and the same object.*

$$DR_{\{[\mathtt{val}]\}}(\{h_1\}) = \{h_2\}$$



Disjoint Reachability

- We extend the formalisation made by Naik and Aiken for a While language to our bytecode language.
- Main steps:

1. Define an instrumented semantic with loop counters: at each allocation site, the new location is tagged with the current loop counter.
2. Formally prove that the instrumentation completely identifies locations: two location tagged with the same loop counter must be equal.
3. Define and prove correct a type and effect system that computes a set Σ of couples `(h1,h2)` such that `h1` points to `h2` but the two corresponding objects were allocated in the same loop iteration.
4. Define and prove correct a sound under-approximation DR^{Σ}_{Paths} of the disjoint reachability set, using the previous type system.

$$DR^{\Sigma}_{Paths} \subseteq DR_{Paths}$$

Using Disjoint Reachability

Disjoint reachability is mixed with two other analyses

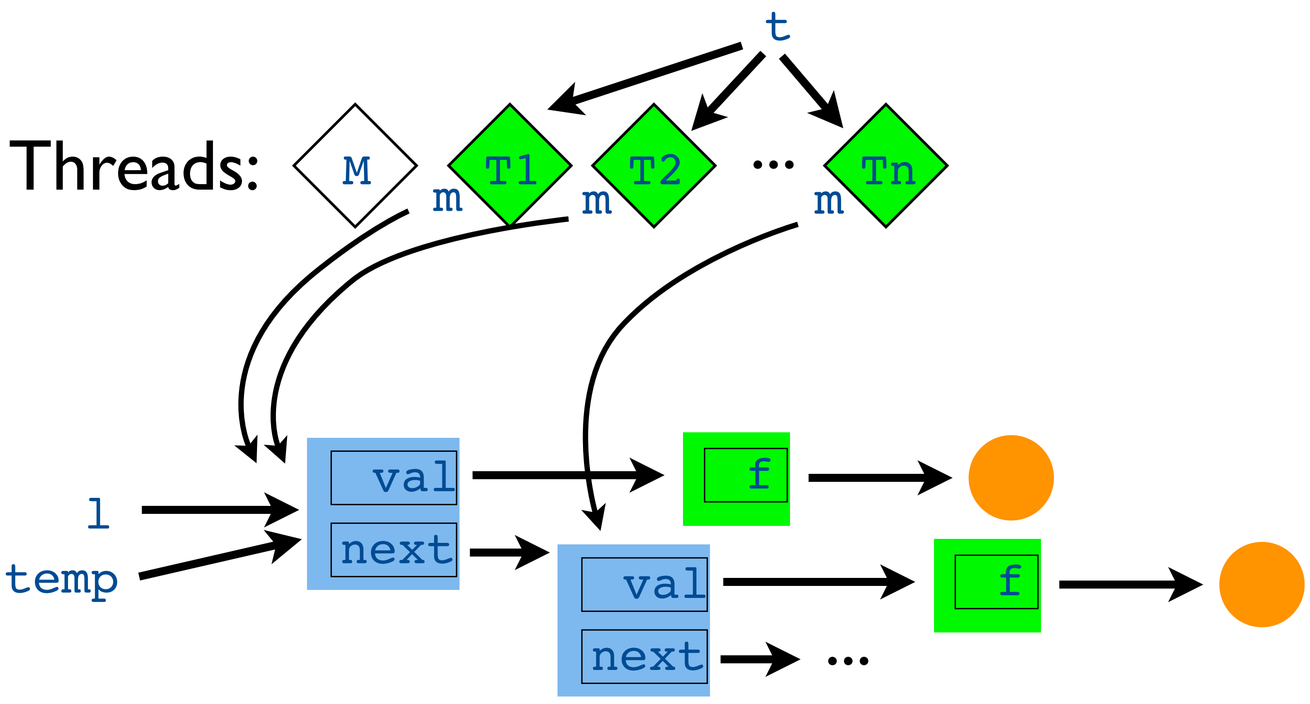
- A must-lock analysis computes a *must* information: for all location targeted by a read or a write, which locks **must** be held by the current thread and from which the location is accessible wrt to the history of heaps ?
- Points-to analysis gives standard *may* information: the set of locations that **may** be targeted by a read or a write.
- We mix all these analyses and remove the potential races (ppt_1, f, ppt_2) such that $Must(ppt_1) \neq \emptyset$, $Must(ppt_2) \neq \emptyset$ and

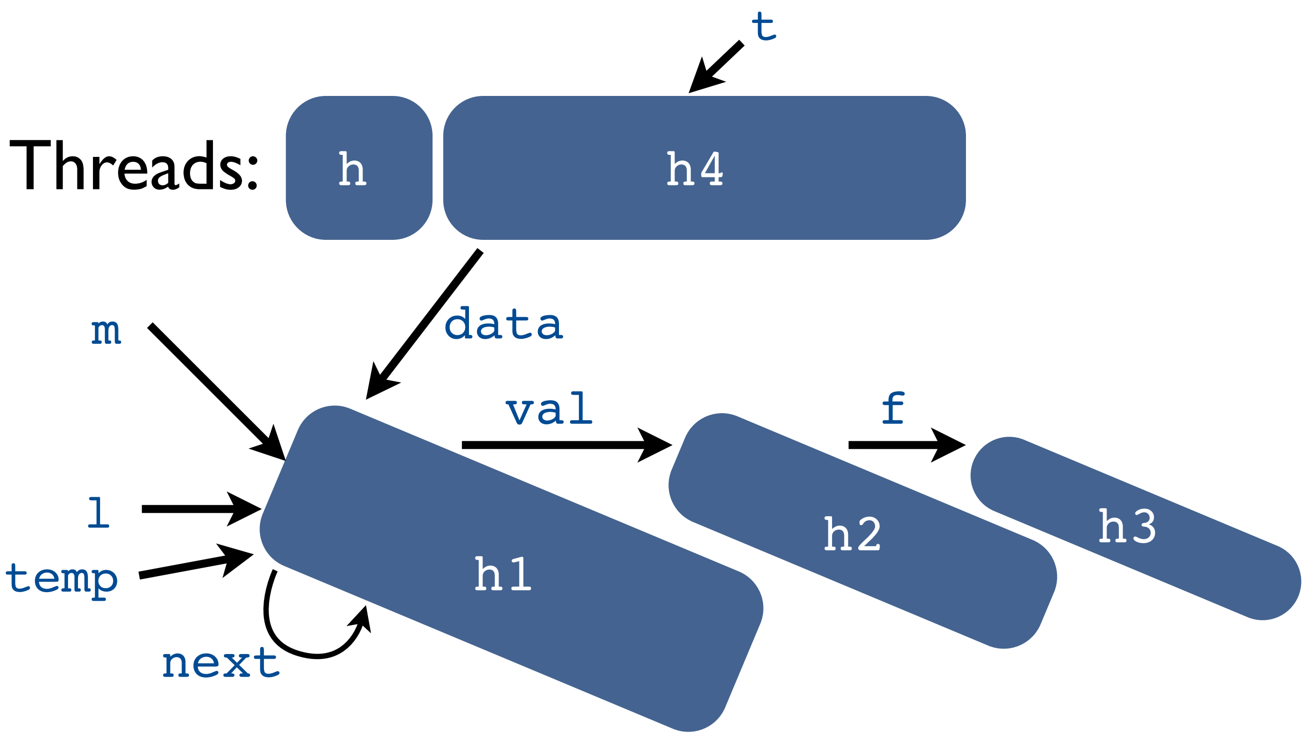
$$May(ppt_1) \cap May(ppt_2) \subseteq DR^{\Sigma}_{Paths}(Must(ppt_1) \cup Must(ppt_2))$$

Running Example

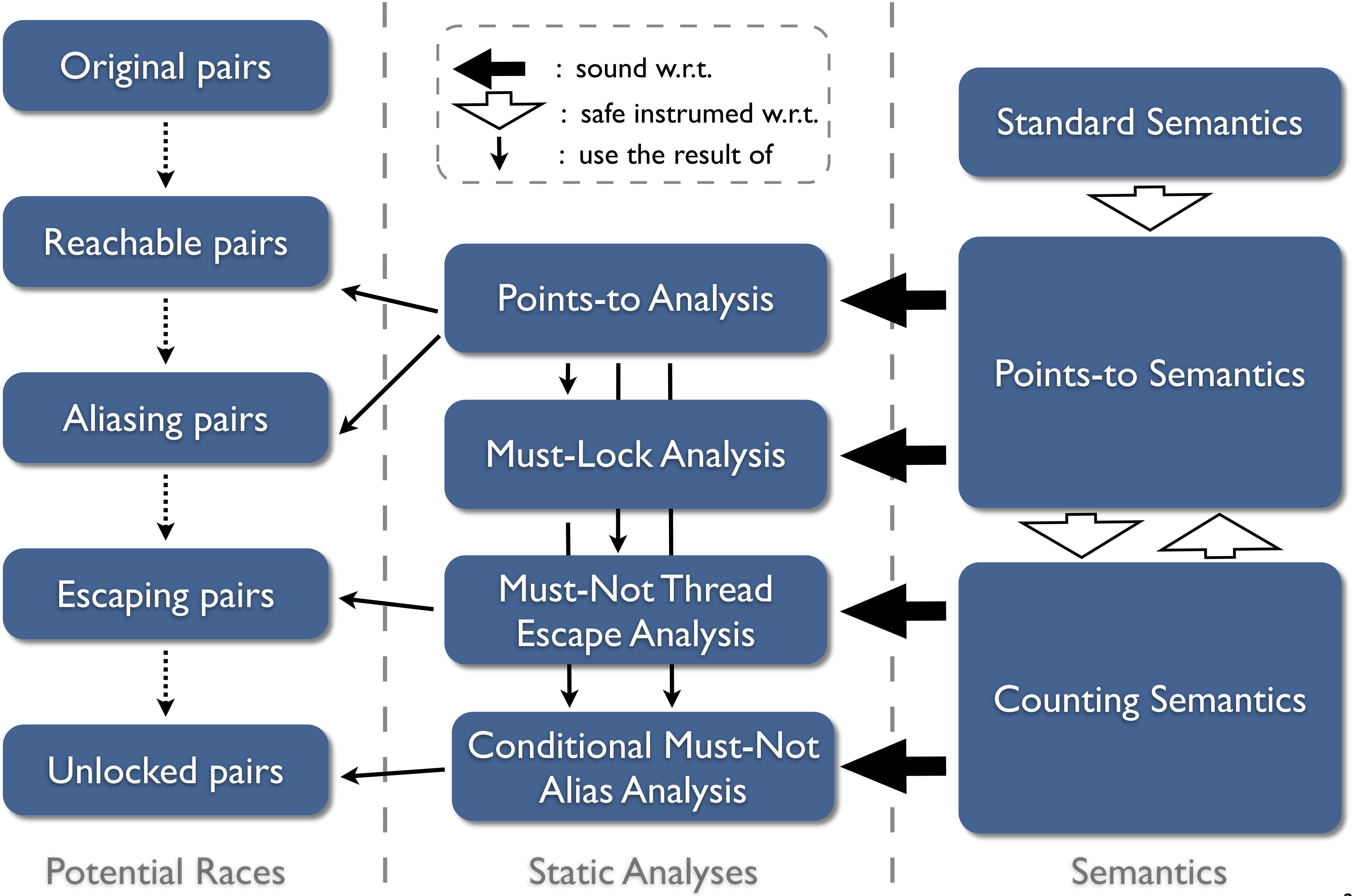
```
synchronize(m){ m.val.f = ...;} ‖ synchronize(m){ m.val.f = ...;}
```

$$\begin{aligned} May_1 &= May_2 = \{h_2\} \\ Must_1 &= Must_2 = \{h_1\} \\ Paths &= \{[\mathtt{val}]\} \\ DR_{\{[\mathtt{val}]\}}(\{h_1\}) &= \{h_2\} \end{aligned} \implies \begin{aligned} & Must_1 \neq \emptyset \wedge \\ & Must_2 \neq \emptyset \wedge \\ & May_1 \cap May_2 \subseteq DR^{\Sigma}_{Paths}(Must_1 \cup Must_2) \end{aligned}$$





The Big Picture



⬅ : sound w.r.t.

⇩ : safe instrumed w.r.t.

↓ : use the result of

Original pairs

Reachable pairs

Aliasing pairs

Escaping pairs

Unlocked pairs

Points-to Analysis

Must-Lock Analysis

Must-Not Thread Escape Analysis

Conditional Must-Not Alias Analysis

Standard Semantics

Points-to Semantics

Counting Semantics

Potential Races

Static Analyses

Semantics

Conclusions and Perspectives

- Points-to static analyses give powerful tools to prove data-race-freeness.
- We need to assemble several complex blocks of this kind to obtain a good tool.
  - Our current formalisation (10.000 line of Coq) should be sufficiently modular to handle new blocks without major reconstruction.
  - Our ultimate goal is to build a powerful certified datarace verifier for bytecode Java.
- But the current formalisation is not executable.
  - Building an efficient certified analyser/checker is a big challenge.
    - Scalable implementations rely on BDDs.
  - We could refine the current formalisation to something executable.

Summary of potential races

```
class Main() {
  void main(){
    List l = null;
    while (*) {
      List temp = new List();
1:    temp.val = new T();
2:    temp.val.f = new A();
3:    temp.next = l;
      l = temp }
    while (*) {
      T t = new T();
4:    t.data = l;
      t.start();
5:    t.f = ...;}
    return;
   }}
```

```
class List{ T val; List next; }

class T {
  A f;
  List data;
  void run(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronized(m){ m.val.f = ...;}}
    return;}}
```

Original	Reachable	Aliasing	Unlocked	Escaping
$(1, \texttt{val}, 1), (1, \texttt{val}, 2), (2, \texttt{f}, 2), (3, \texttt{next}, 3),$ $(4, \texttt{data}, 4)$		✓	✓	
$(5, \texttt{f}, 5)$		✓	✓	✓
$(2, \texttt{f}, 5)$			✓	
$(5, \texttt{f}, 8)$	✓		✓	✓
$(4, \texttt{data}, 6), (3, \texttt{next}, 7), (1, \texttt{val}, 8), (2, \texttt{f}, 8)$	✓	✓	✓	
$(8, \texttt{f}, 8)$	✓	✓		✓