A Certified Data Race Analysis for a Java-like Language

Frédéric Dabrowski and David Pichardie

INRIA Rennes - Bretagne Altantique

ParSec meeting - 8 January 2009

Data Races

- A fundamental issue in multi-threaded programming
- Definition: *the situation where two 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
- 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;
x = C.g; ‖ y = C.f;
C.f = 1; ‖ C.g = 1;
```

Example

```
    C.f = C.g = 0;
x = C.g; ║ y = C.f;
C.f = 1; ║ C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

Example

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

Interleaving semantics gives only sequentially consistent execution,

```
C.f = C.g = 0; x = C.g; C.f = 1; y = C.f; C.g = 1;
```

Example

```
    C.f = C.g = 0;
x = C.g; ║ y = C.f;
C.f = 1; ║ C.g = 1;
```

Interleaving semantics gives only sequentially consistent execution,

```
C.f = C.g = 0; x = C.g; C.f = 1; y = C.f; C.g = 1;

C.f = C.g = 0; x = C.g; y = C.f; C.f = 1; C.g = 1;
```

Example

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

Interleaving semantics gives only sequentially consistent execution,

```
C.f = C.g = 0; x = C.g; C.f = 1; y = C.f; C.g = 1;

C.f = C.g = 0; x = C.g; y = C.f; C.f = 1; C.g = 1;

C.f = C.g = 0; y = C.f; x = C.g; C.g = 1; C.f = 1;
```

Example

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

Interleaving semantics gives only sequentially consistent execution,

```
C.f = C.g = 0; x = C.g; C.f = 1; y = C.f; C.g = 1;

C.f = C.g = 0; x = C.g; y = C.f; C.f = 1; C.g = 1;

C.f = C.g = 0; y = C.f; x = C.g; C.g = 1; C.f = 1;
```

but such program may also lead to sequentially inconsistent execution

Example

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

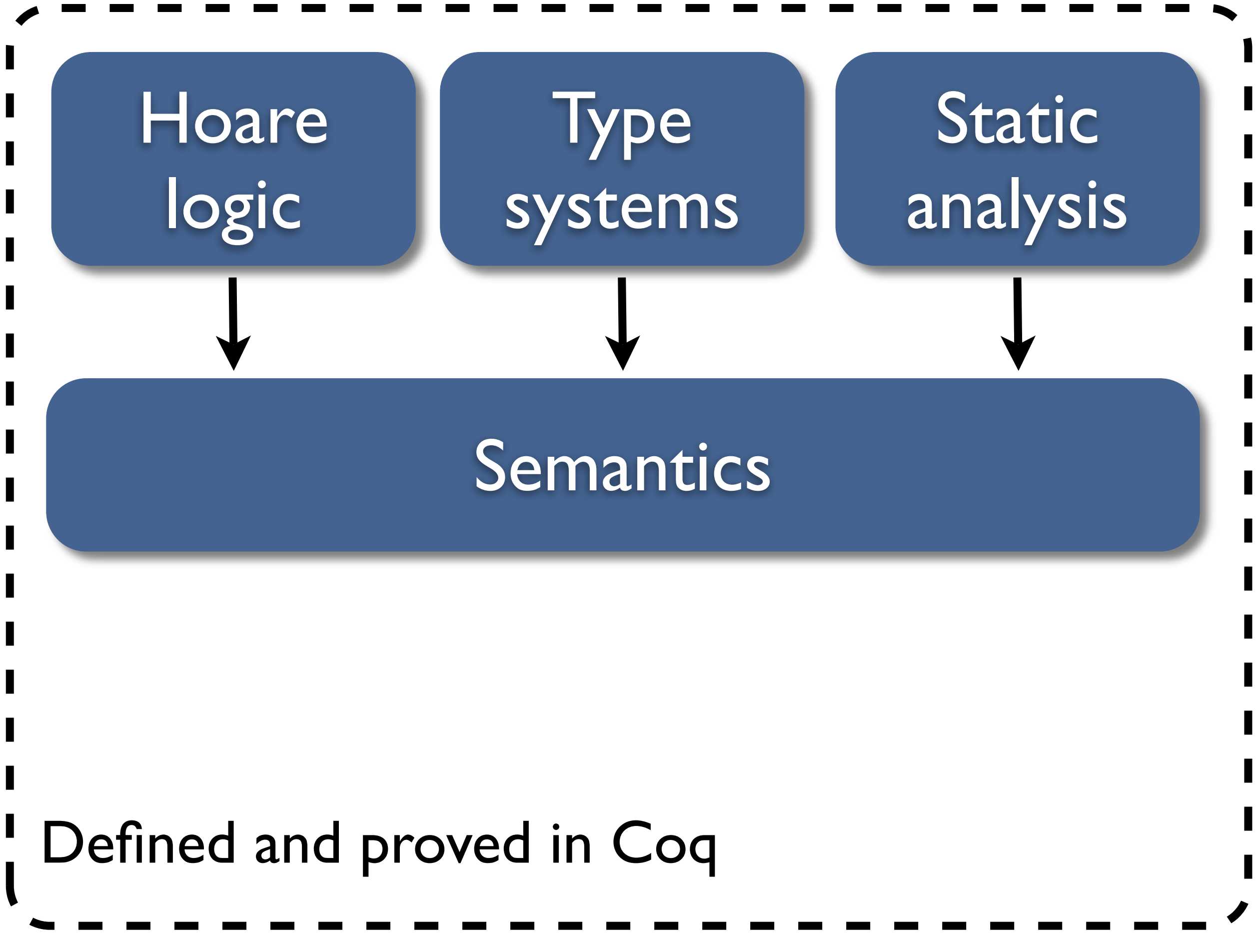
Interleaving semantics gives only sequentially consistent execution,

```
C.f = C.g = 0; x = C.g; C.f = 1; y = C.f; C.g = 1;

C.f = C.g = 0; x = C.g; y = C.f; C.f = 1; C.g = 1;

C.f = C.g = 0; y = C.f; x = C.g; C.g = 1; C.f = 1;
```

but such program may also lead to sequentially inconsistent execution

`C.f = C.g = 0; C.f = 1; C.g = 1; x = C.g; y = C.f;`   `x=1` and `y=1` !

Certified program verification



Hoare logic

Type systems

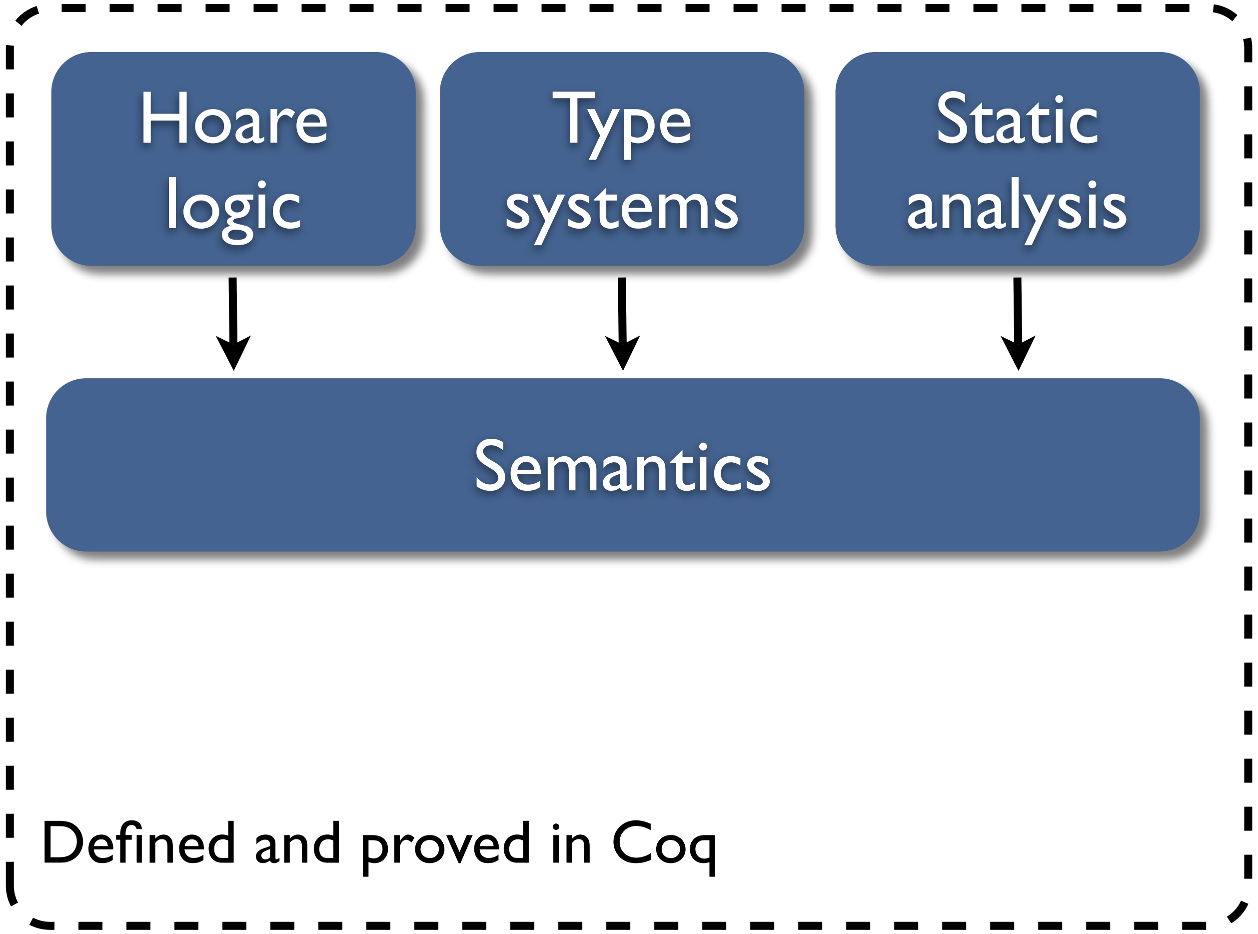
Static analysis

Semantics

Defined and proved in Coq

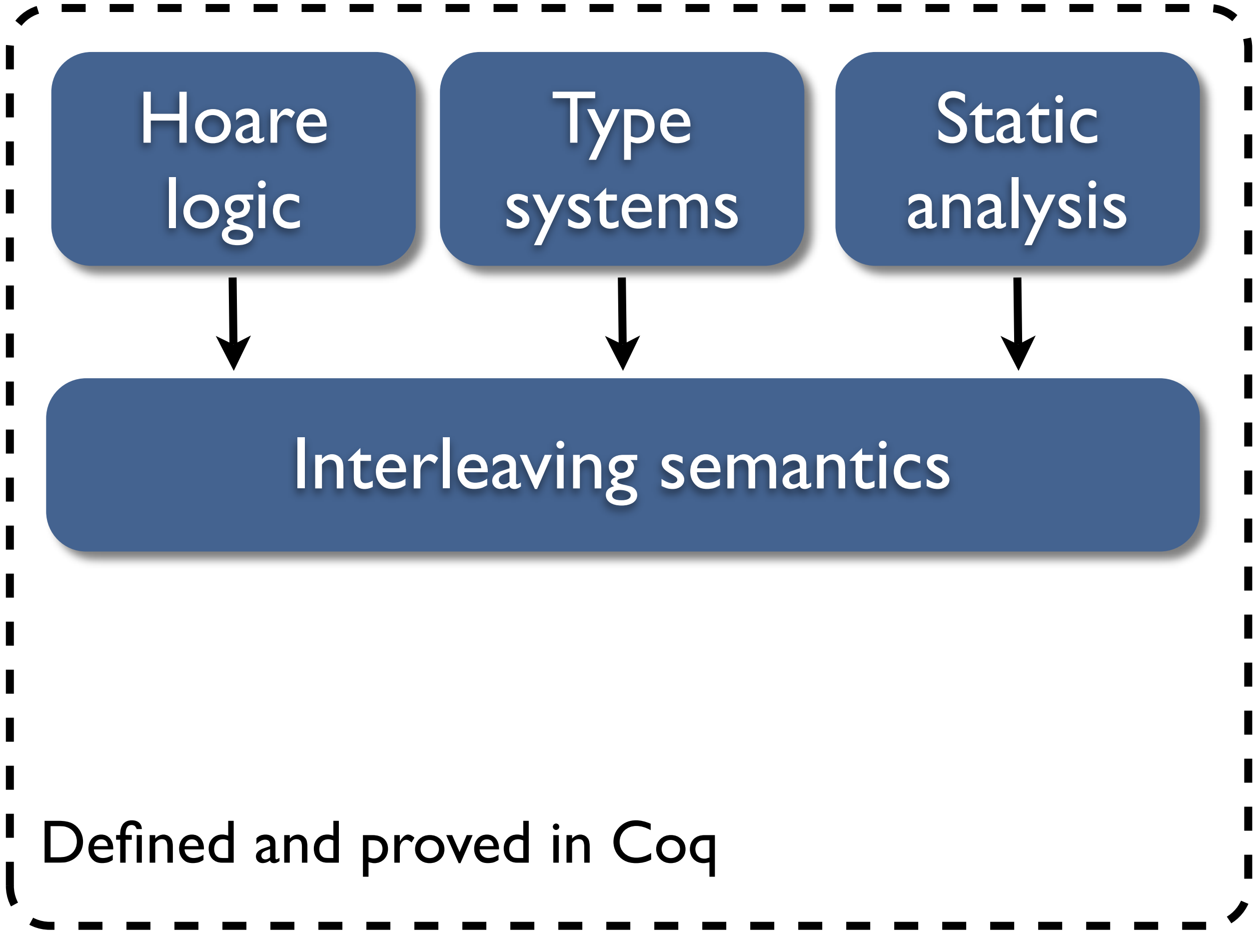
Certified program verification

- 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



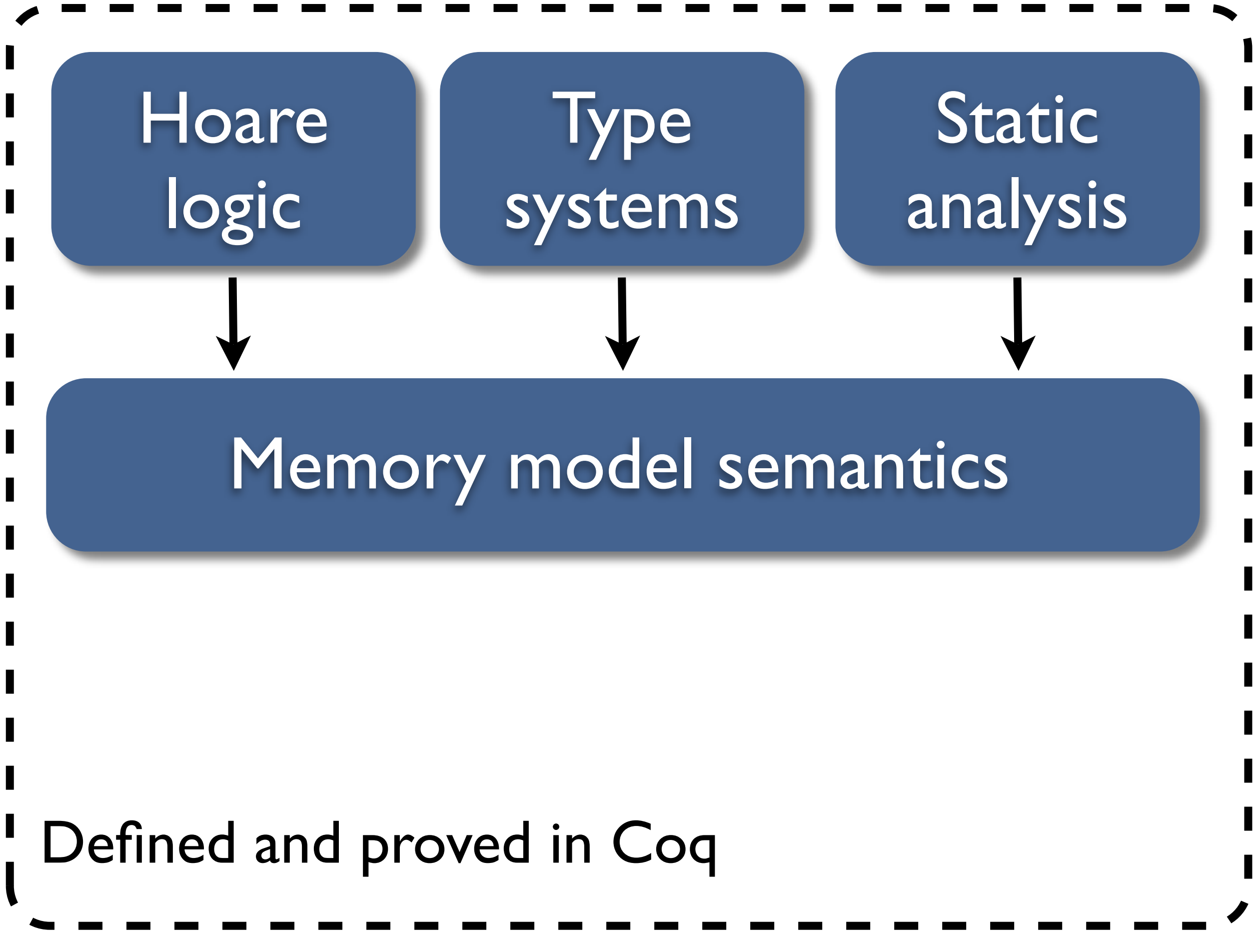
Certified program verification

- 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

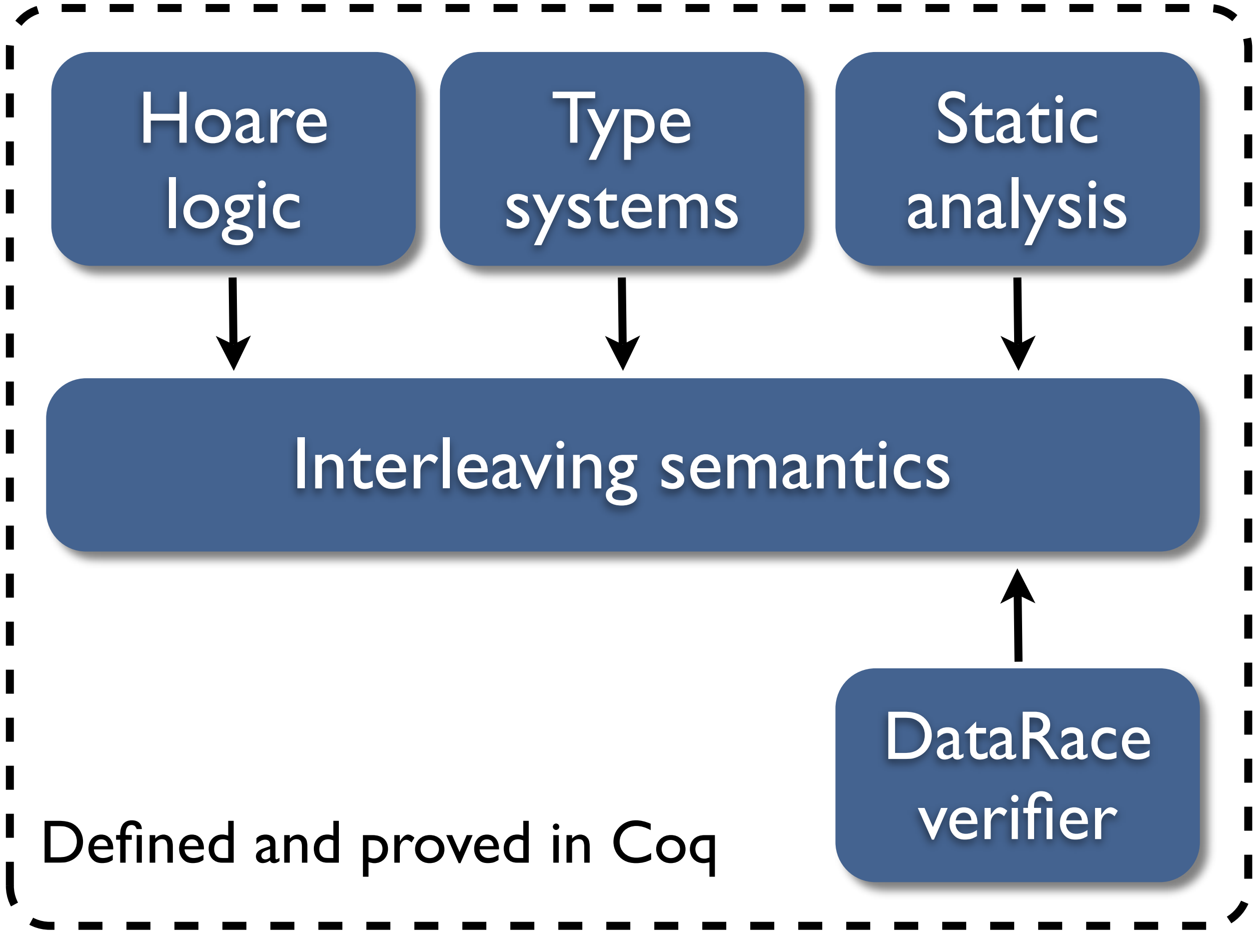
Static analysis

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



Hoare logic

Type systems

Static analysis

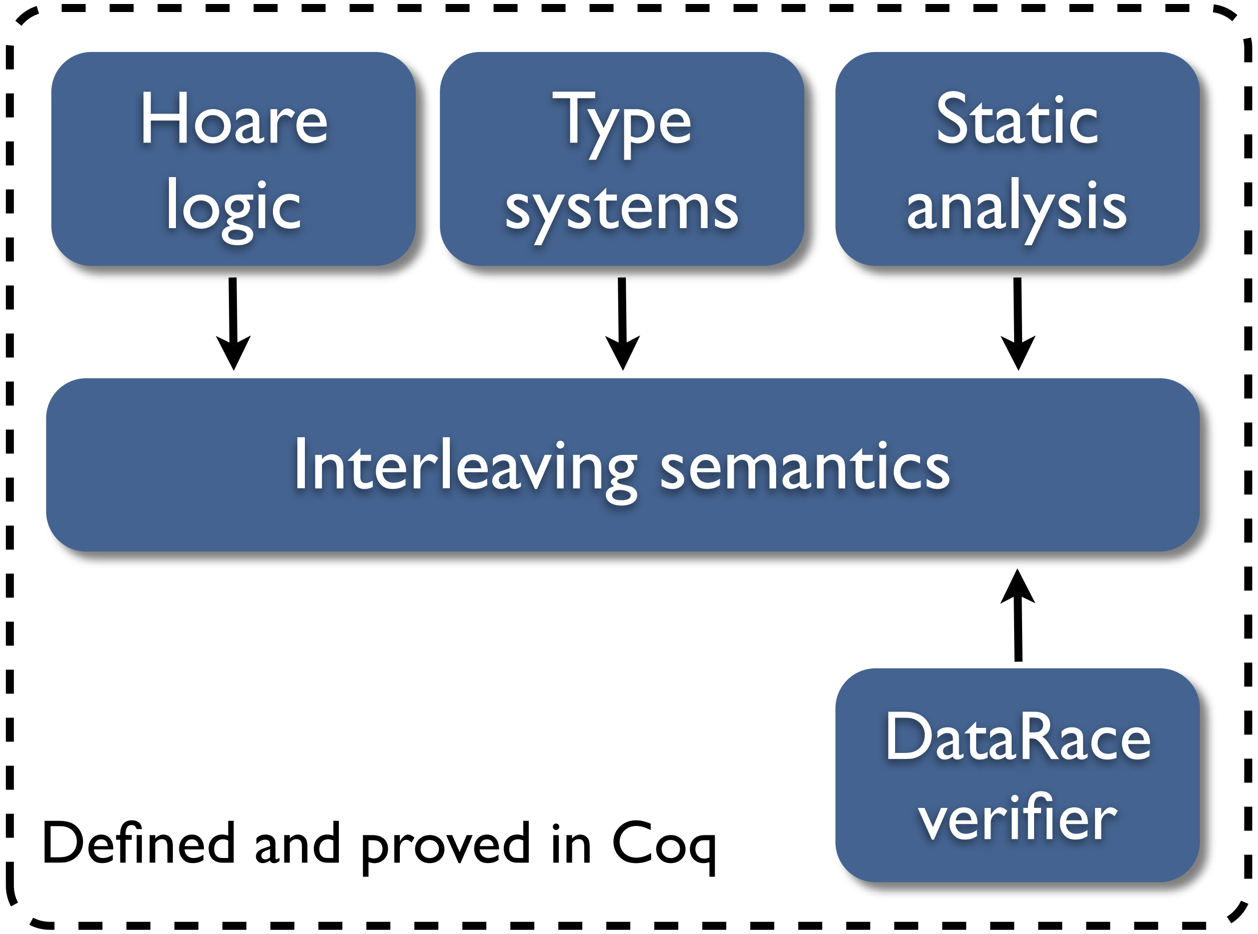
Interleaving semantics

DataRace verifier

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



Hoare logic

Type systems

Static analysis

Interleaving semantics

DataRace verifier

Defined and proved in Coq

- 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 analysis

Running example

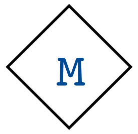
```
class List{ T val; List next; }

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 = new T();
4:    t.f = ...;
5:    t.data = l;
      t.start() }
    return;
   }
}

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

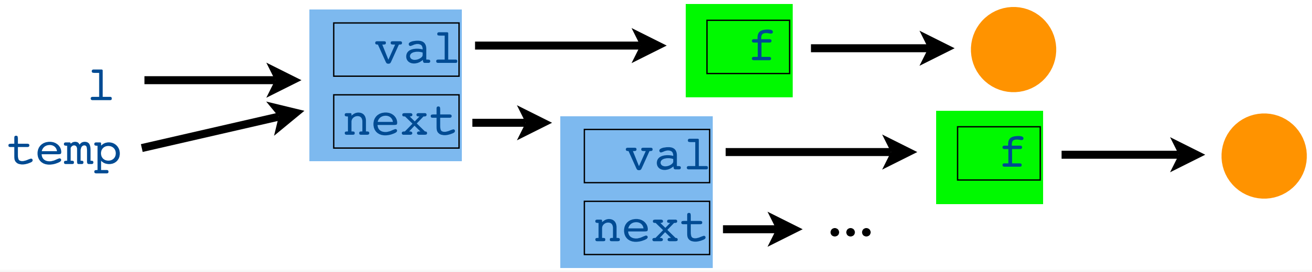
Running example

```
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 = new T();
4:    t.f = ...;
5:    t.data = l;
      t.start() }
    return;
   }
}

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

1. We create a link list `l`

Threads:

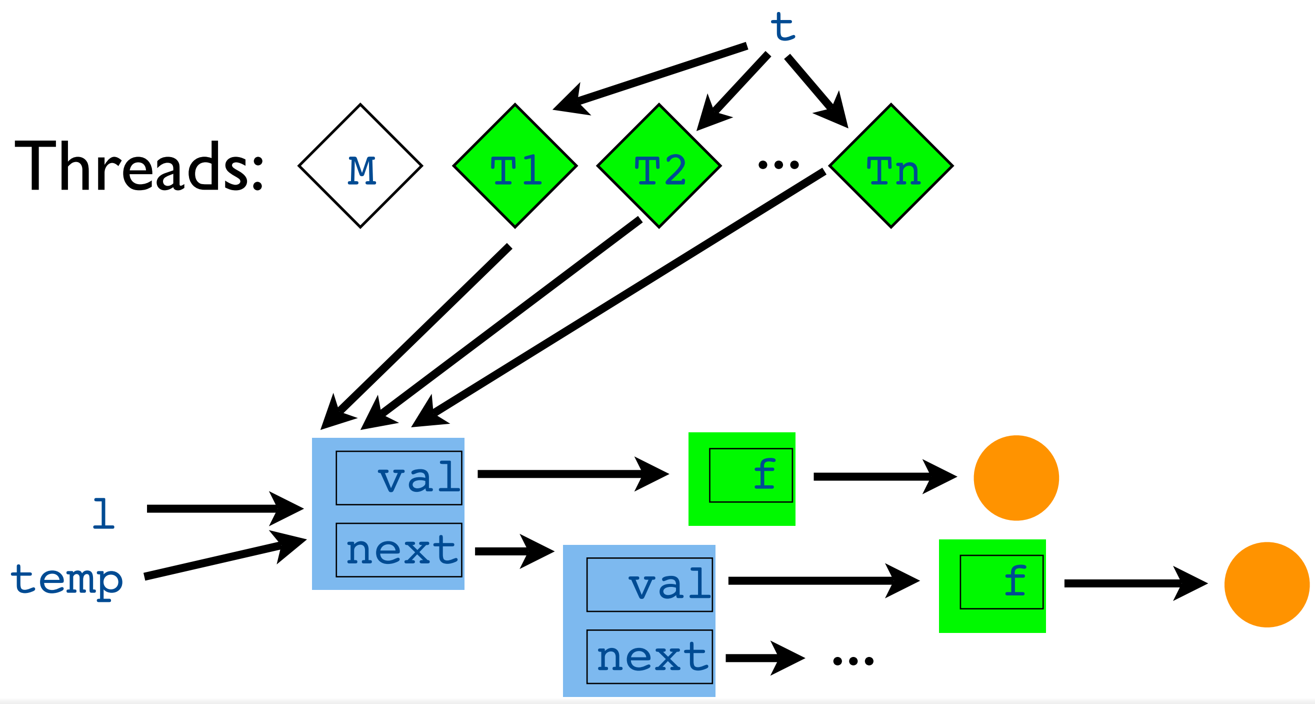




Running example

```
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 = new T();
4:    t.f = ...;
5:    t.data = l;
      t.start() }
    return;
   }
}

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

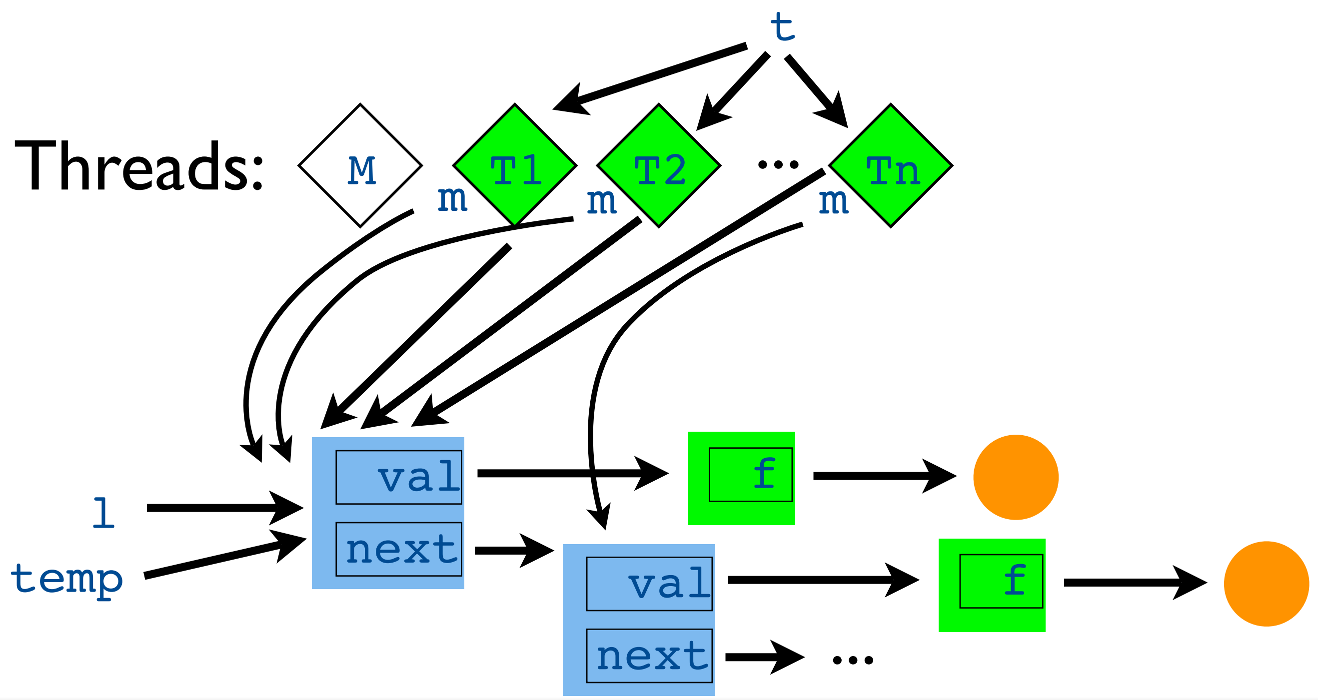
1. We create a link list `l`
2. We create a bunch of thread that all share the list `l`



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 = new T();
4:    t.f = ...;
5:    t.data = l;
      t.start() }
    return;
   }
}

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

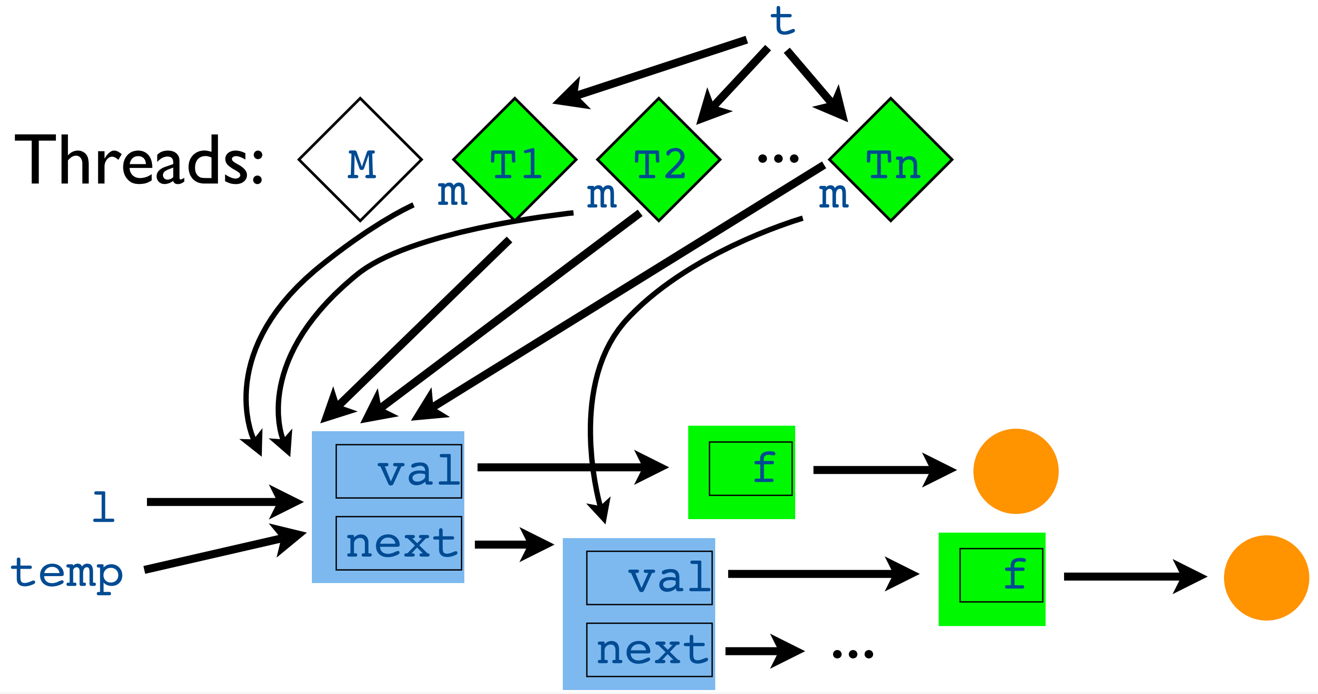
1. We create a link list `l`
2. We create a bunch of thread that all share the list `l`
3. Each thread chooses a cell, takes a lock on it and updates it.



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 = new T();
4:    t.f = ...;
5:    t.data = l;
      t.start() }
    return;
   }
}

class T extends java.lang.Thread {
  A f;
  List data;
  void start(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronize(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.

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

Semantics

- Semantic domains

$$
\begin{array}{rcll}
\mathbb{O} \ni \ell & & & \text{(memory location)} \\
\mathbb{O}_\perp \ni v & ::= & \ell \mid \texttt{Null} & \text{(value)} \\
s & ::= & v :: s \mid \varepsilon & \text{(operand stack)} \\
Var \to \mathbb{O}_\perp \ni \rho & & & \text{(local variables)} \\
\mathbb{O} \rightharpoonup \mathbb{F} \to \mathbb{O}_\perp \ni \sigma & & & \text{(heap)} \\
CS \ni cs & ::= & (m, i, s, \rho) :: cs \mid \Box & \text{(call stack)} \\
\mathbb{O} \rightharpoonup CS \ni L & & & \text{(threads)} \\
\mathbb{O} \to ((\mathbb{O} \times \mathbb{N}^*) \cup \{\texttt{free}\}) \ni \mu & & & \text{(locking state)} \\
st & ::= & (L, \sigma, \mu) & \text{(state)}
\end{array}
$$

- Actions

$$
\begin{array}{rcl}
PPT = \mathbb{M} \times \mathbb{N} \ni ppt & & \\
e & ::= & * \mid (\ell, ?_f^{ppt}, \ell') \mid (\ell, !_f^{ppt}, \ell')
\end{array}
$$

- 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)

$$\frac{\begin{array}{c}(m.\texttt{body})\ i = \texttt{new}\ cid \qquad \neg(l' \in dom(\sigma)) \\ L' = L[\ell \mapsto (m, i+1, \ell' :: s, \rho) :: cs]\end{array}}{L; \ell \vdash ((m, i, s, \rho) :: cs, \sigma, \mu) \to (L', \sigma[\ell' \leftarrow], \mu)}$$

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

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

- Races

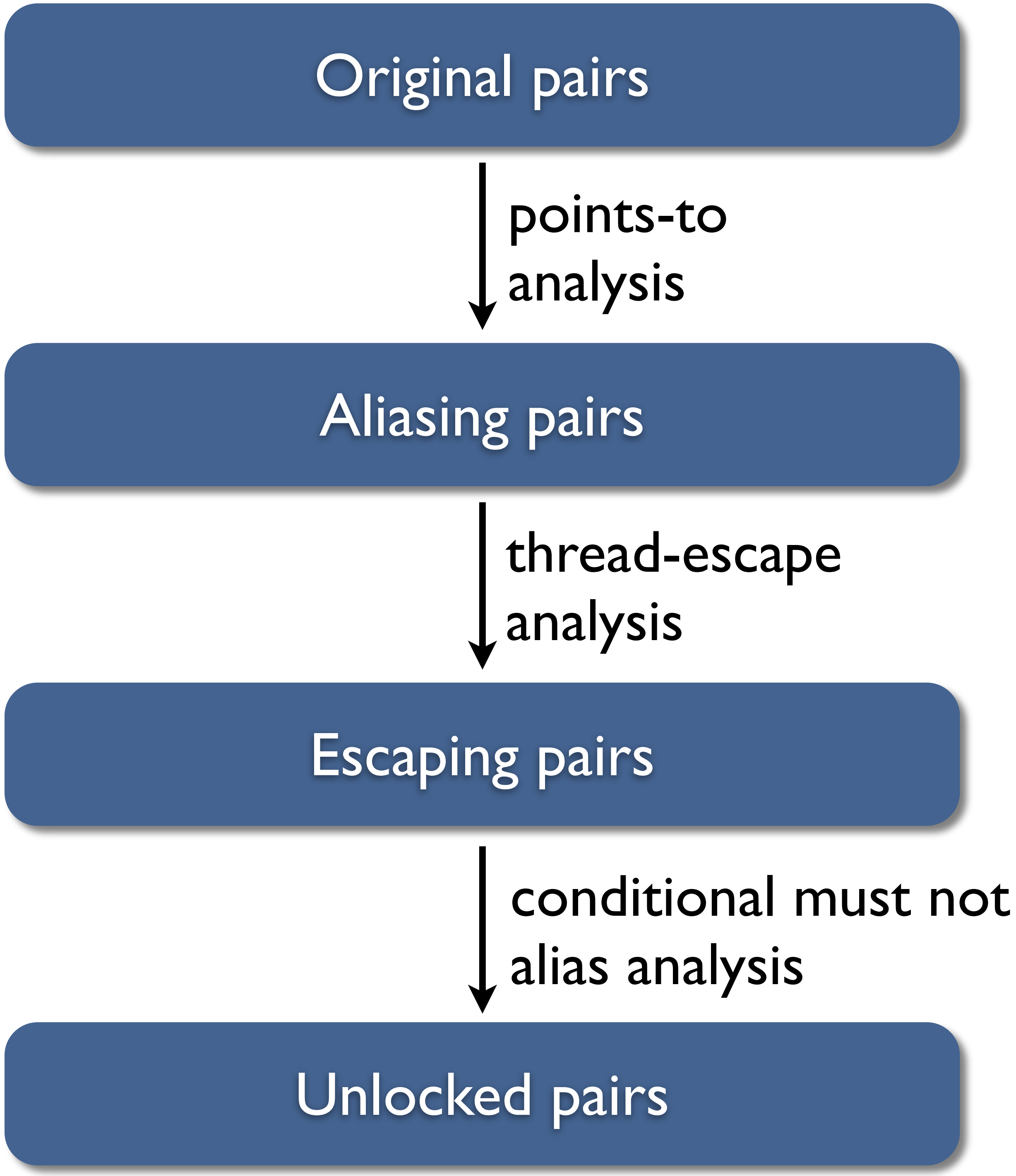
$$\frac{st \in \mathrm{ReachableStates}(P) \qquad st \overset{\ell_1 !_f^{ppt_1} \ell_0}{\longrightarrow} st_1 \qquad st \overset{\ell_2 \mathcal{R} \ell_0}{\longrightarrow} st_2 \qquad \mathcal{R} \in \{?_f^{ppt_2}, !_f^{ppt_2}\} \qquad \ell_1 \neq \ell_2}{Race(P, ppt_1, f, ppt_2)}$$

Data Race Analysis

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

Data Race Analysis

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



Original pairs

points-to analysis

Aliasing pairs

thread-escape analysis

Escaping pairs

conditional must not alias analysis

Unlocked pairs

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 = new T();
4:    t.f = ...;
5:    t.data = l;
      t.start() }
    return;
   }
}

class T {
  A f;
  List data;
  void start(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronize(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 :

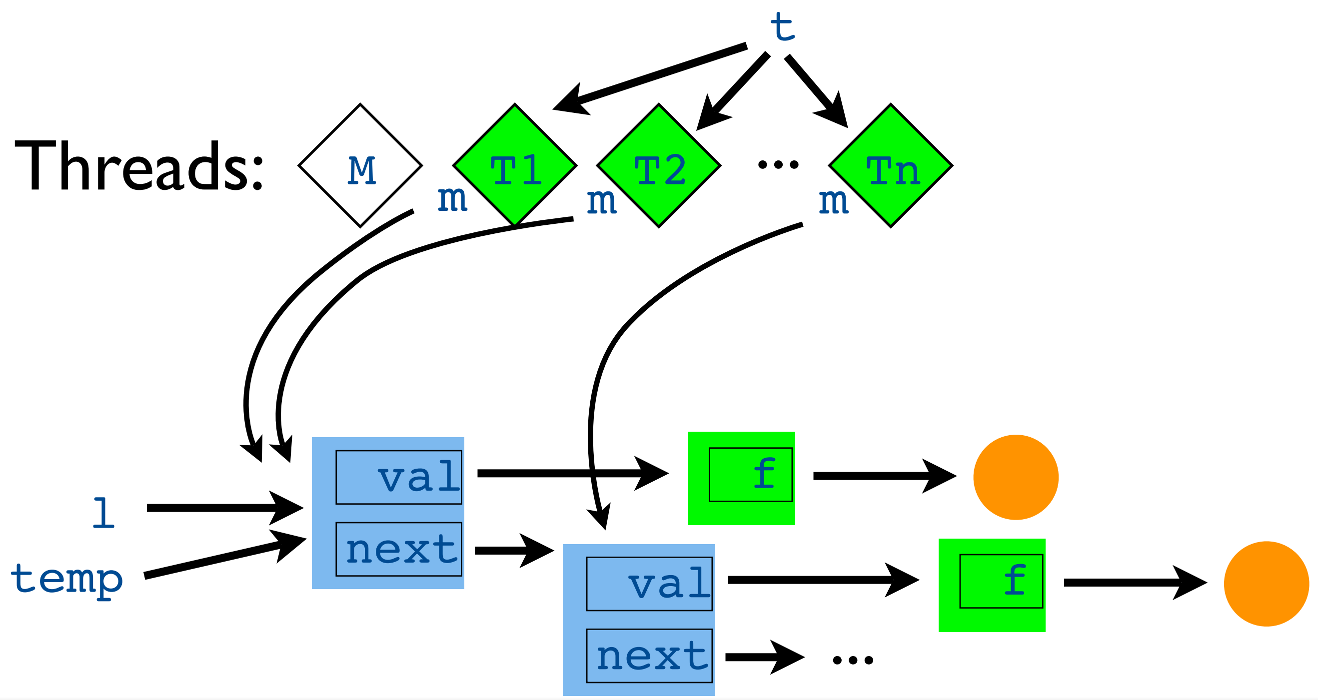
```
(1,val,1),(1,val,2),(2, f, 2), (3, next, 3),
(5,data,5),(4,f,4),
(2,f,4),(4,f,8),
(5,data,6),(3,next,7),(1,val,8),(2,f,8),
(8,f,8)
```

Points-to analysis

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

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

class T {
  A f;
  List data;
  void start(){
    while(*){
6:    List m = this.data;
7:    while (*) { m = m.next; }
8:    synchronize(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),
(5,data,5),(4,f,4),
(2,f,4),(4,f,8),
(5,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 = new T();
4:     t.f = ...;
5:     t.data = l;
       t.start() }
    return;
   }
}

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

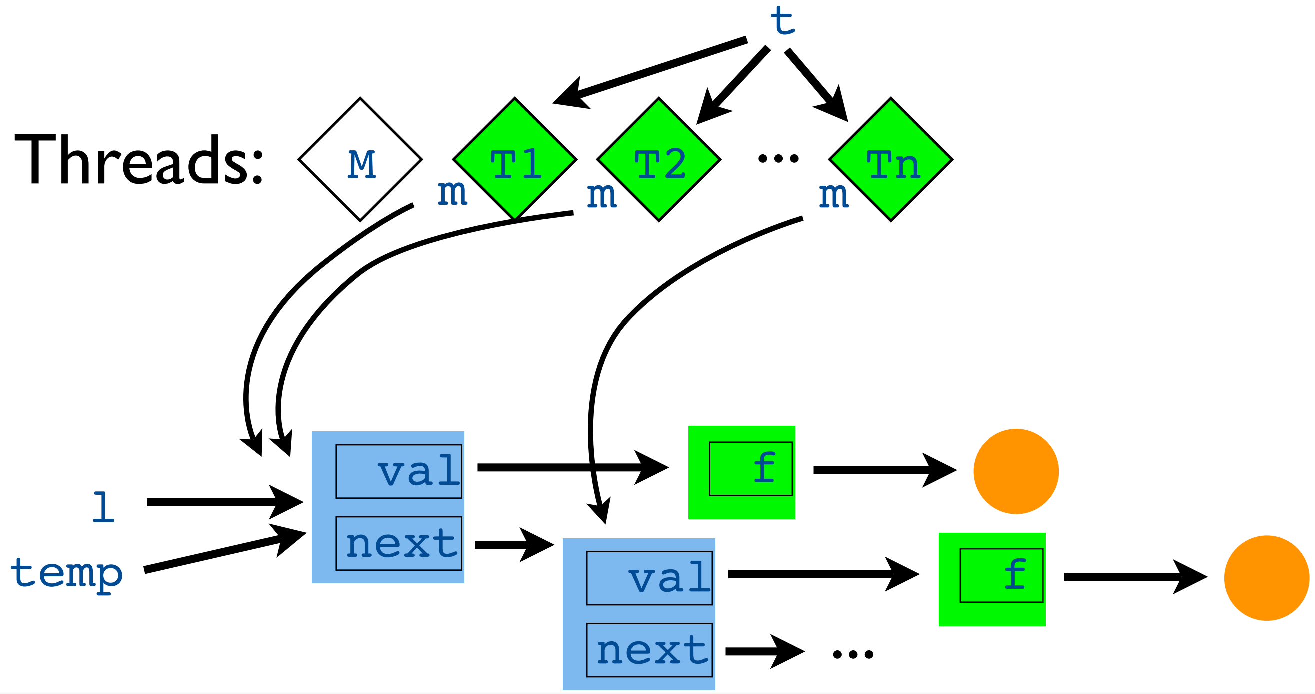
Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site

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

`(2,f,4),(4,f,8),`

`(5,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 = new T();
4:     t.f = ...;
5:     t.data = l;
       t.start() }
    return;
  }
}

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

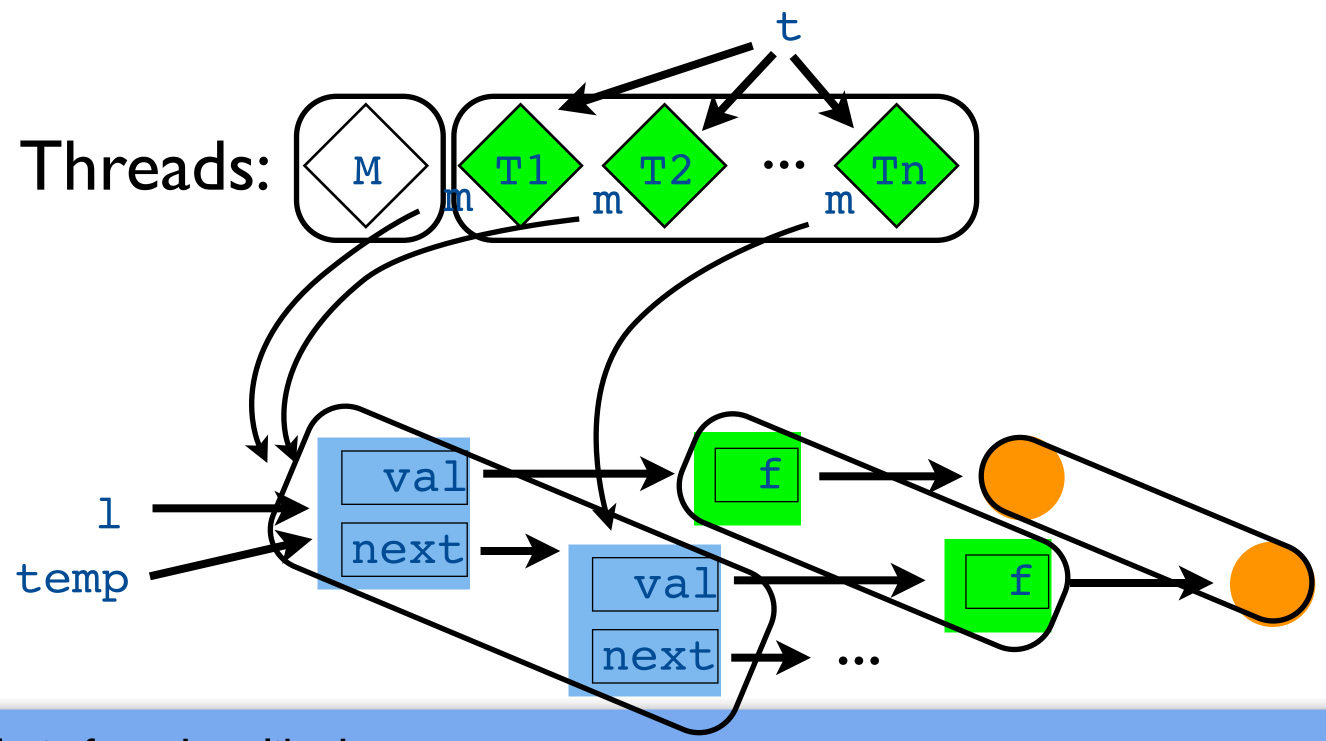
Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site

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

`(2,f,4),(4,f,8),`

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

`(8,f,8)`

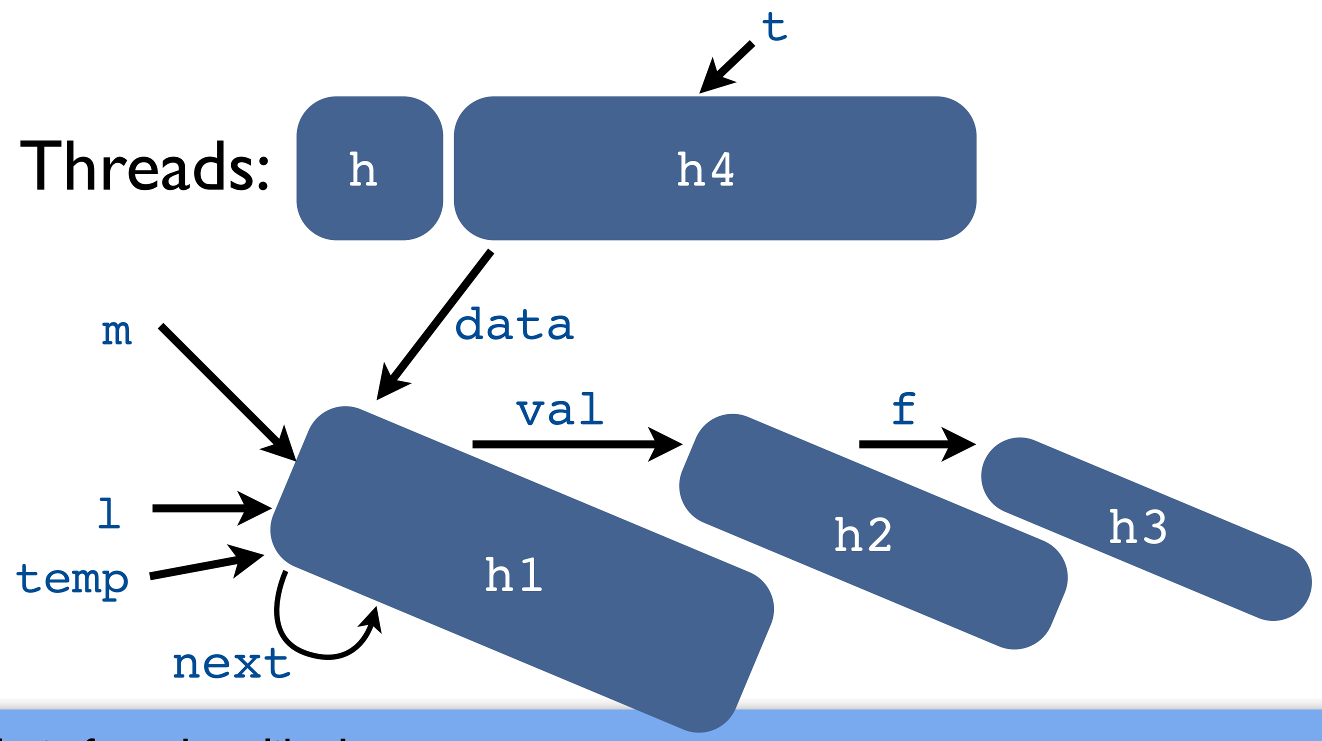


Points-to analysis

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

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 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 = new T();
4:     t.f = ...;
5:     t.data = l;
       t.start() }
    return;
   }
}

class T {
  A f;
  List data;
  void start(){
    while(*){
6:     List m = this.data;
7:     while (*) { m = m.next; }
8:     synchronize(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),
(5,data,5),(4,f,4),
(2,f,4),(4,f,8),
(5,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 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 = new T();
4:       t.f = ...;
5:       t.data = l;
         t.start() }
    return;
   }
}

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

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

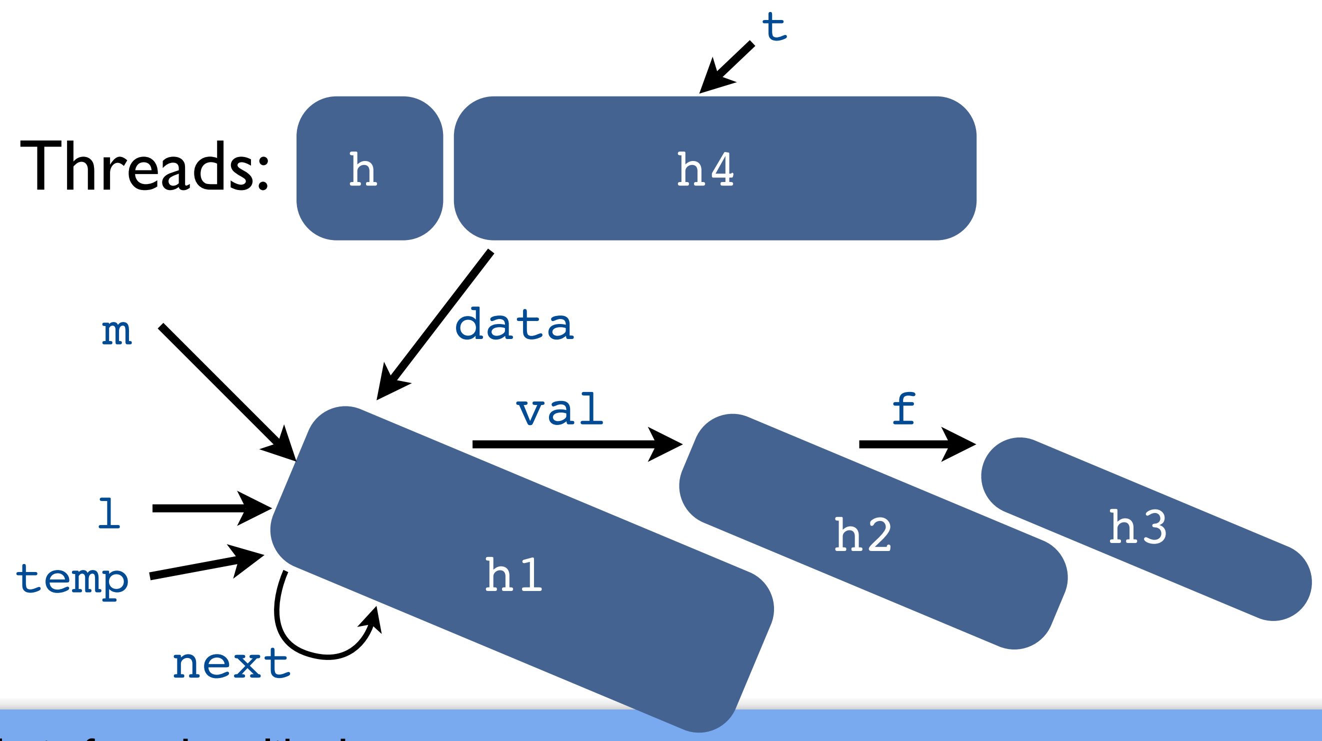
- h is a *single-instance* allocation site

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

(2,f,4),(4,f,8),

(5,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 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 = new T();
4:     t.f = ...;
5:     t.data = l;
       t.start() }
    return;
   }
}

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

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

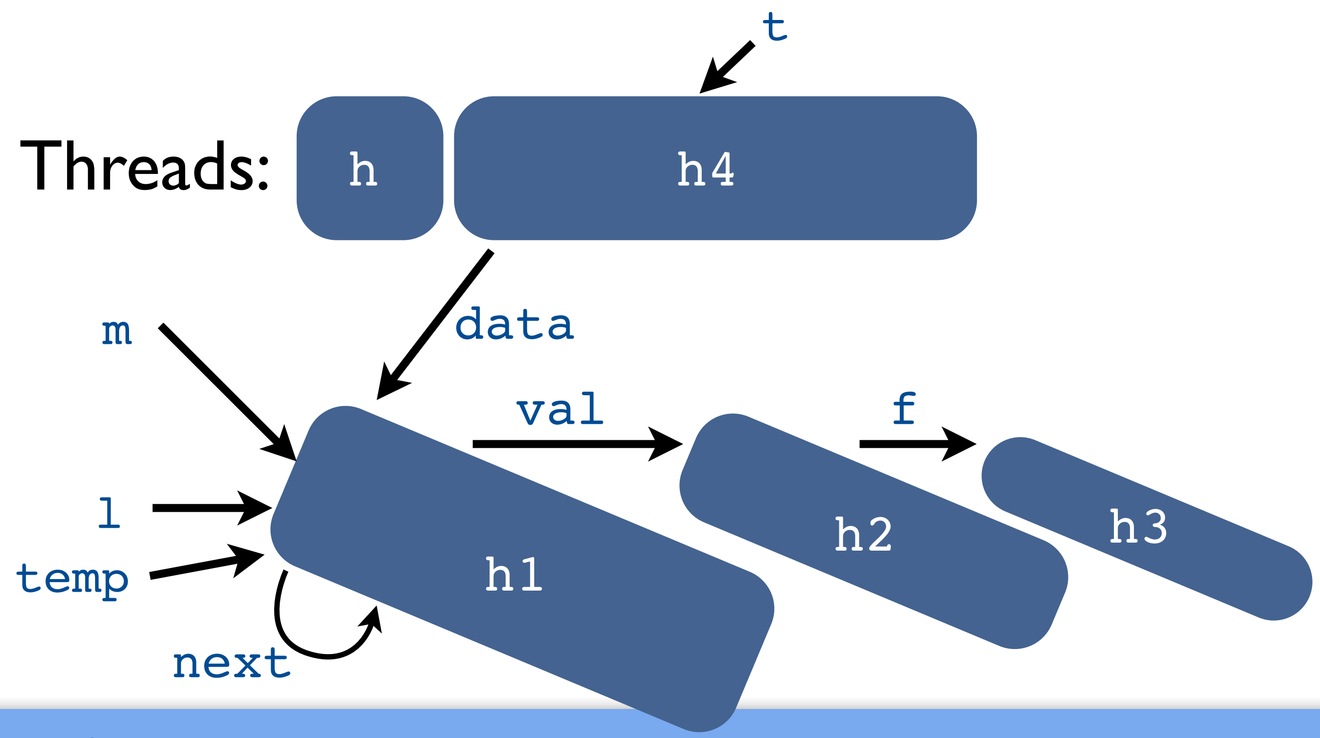
- `h` is a *single-instance* allocation site

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

(2,f,4),(4,f,8),

(5,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 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 = new T();
4:     t.f = ...;
5:     t.data = l;
       t.start() }
    return;
   }
}

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

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

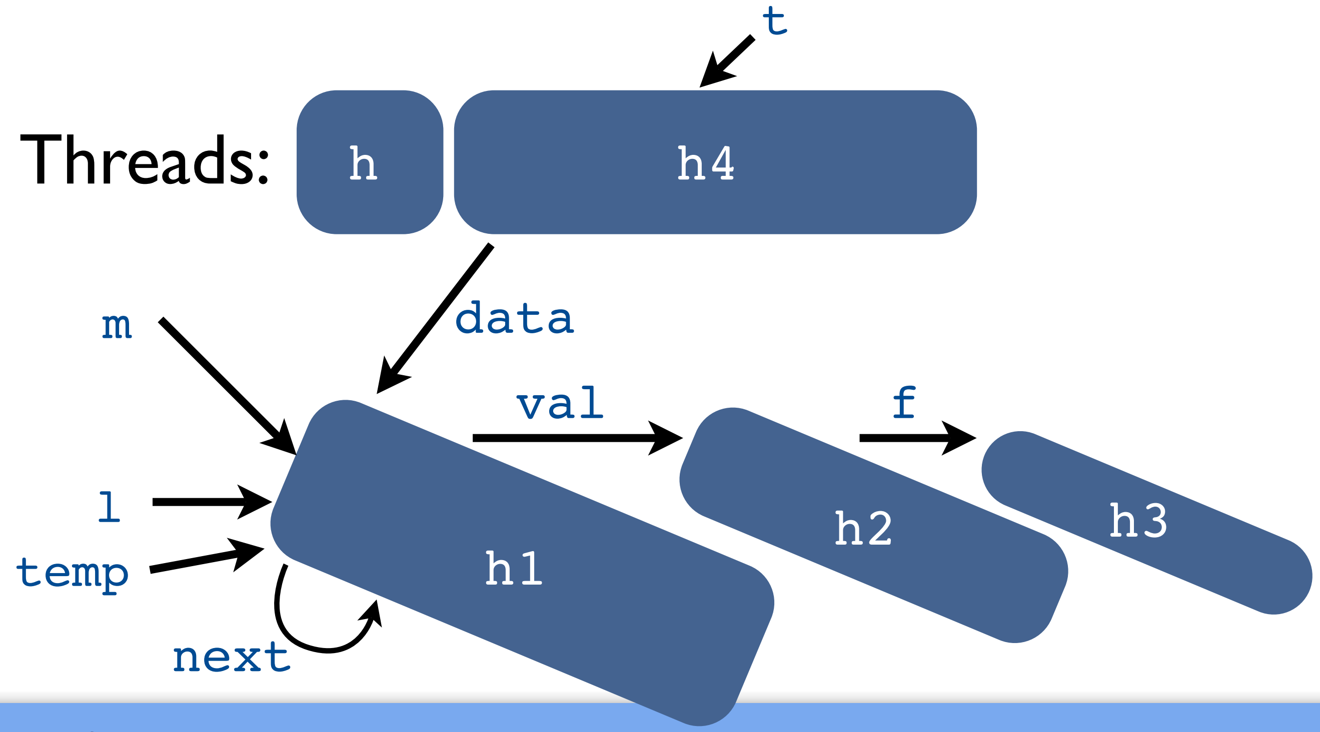
- `t` points-to `h4`
- `temp.val` and `m.val` points-to `h2`

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

`(2,f,4),(4,f,8),`

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

`(8,f,8)`



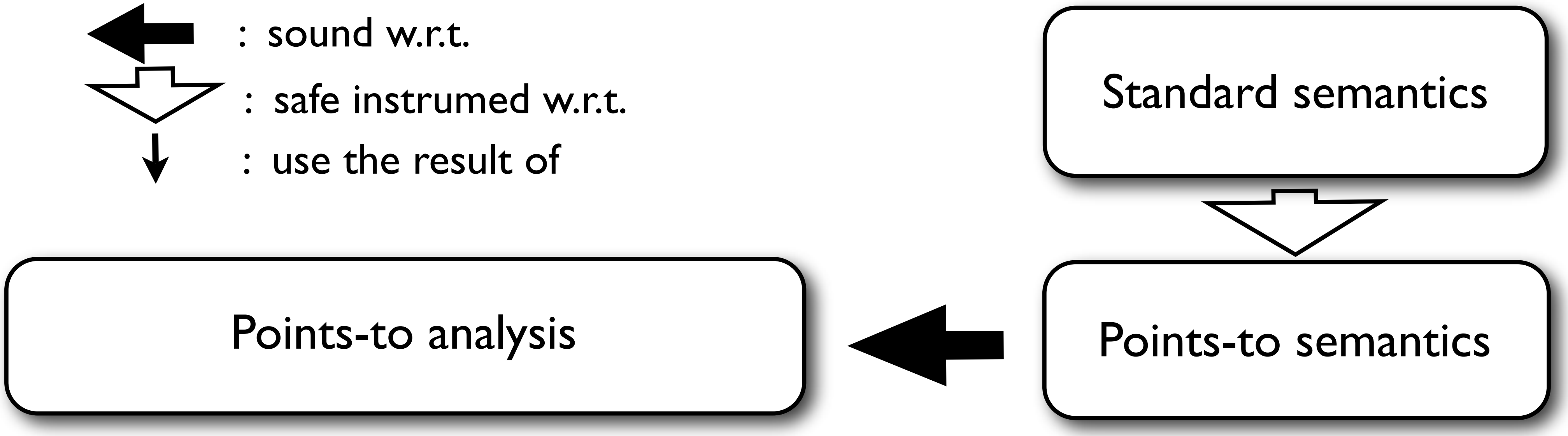
Points-to analysis in Coq

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*.



Thread-escape analysis

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

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 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 = new T();
4:     t.f = ...;
5:     t.data = l;
       t.start() }
    return;
   }
}

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

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

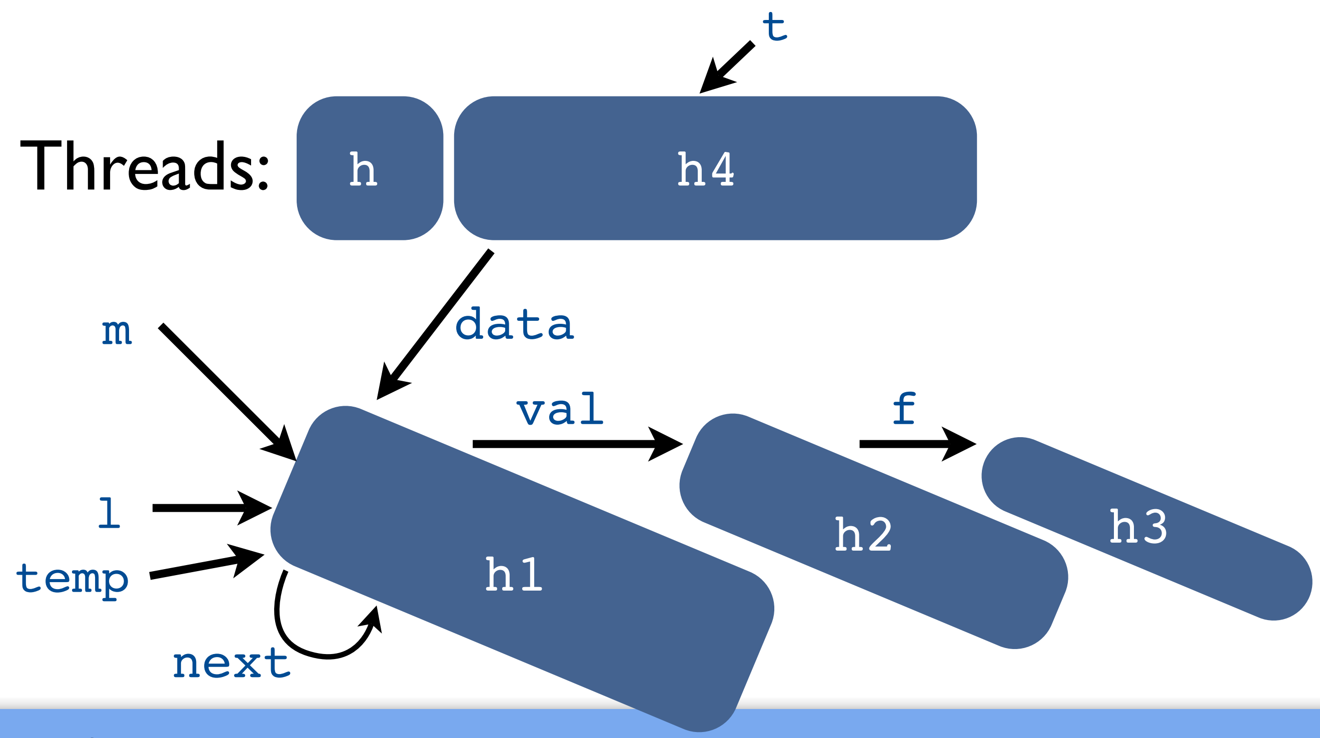
- Naik uses a flow sensitive thread-escape analysis
- We are currently working on its formalisation

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

~~(2,f,4),(4,f,8),~~

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

(8,f,8)



Thread-escape analysis

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

class Main() {
  void main(){
    List l = null;
    while (*) {
    h1 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 = new T();
4:     t.f = ...;
5:     t.data = l;
       t.start() }
    return;
   }
}

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

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

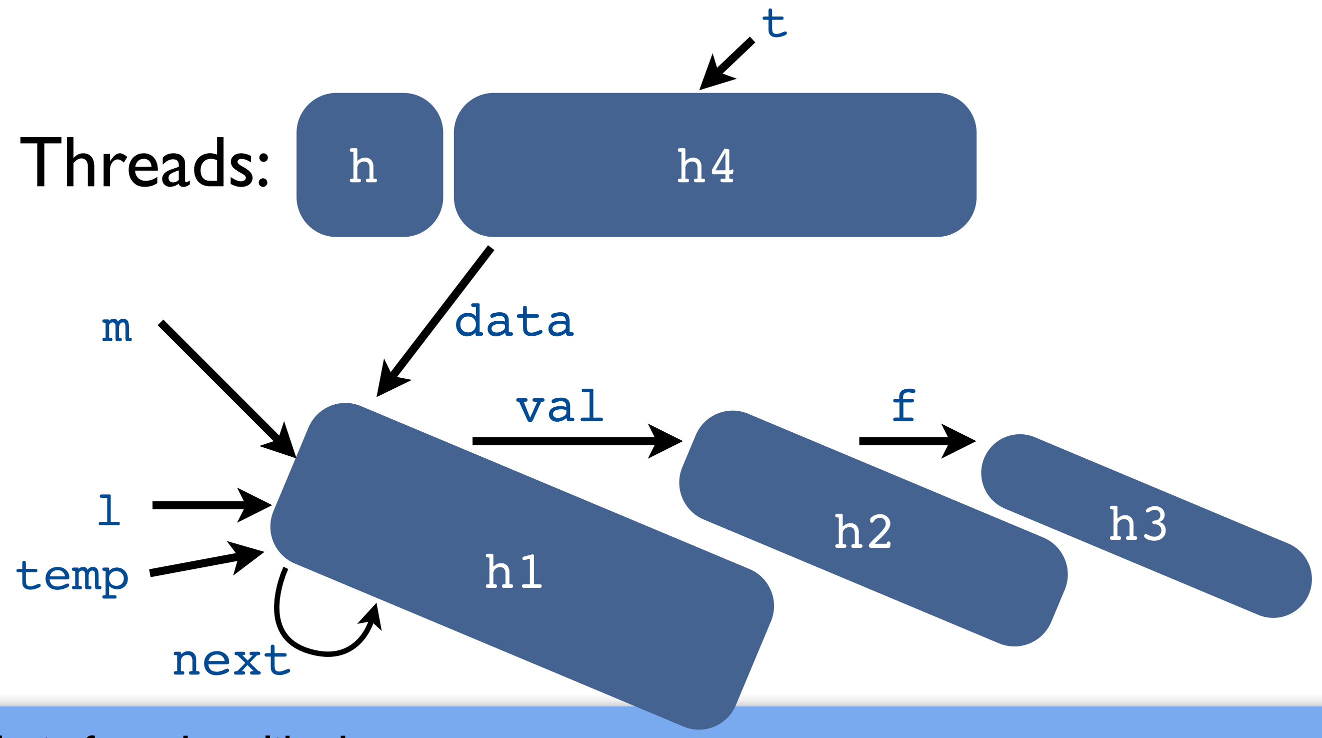
- Naik uses a flow sensitive thread-escape analysis
- We are currently working on its formalisation

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

~~(2,f,4),(4,f,8),~~

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

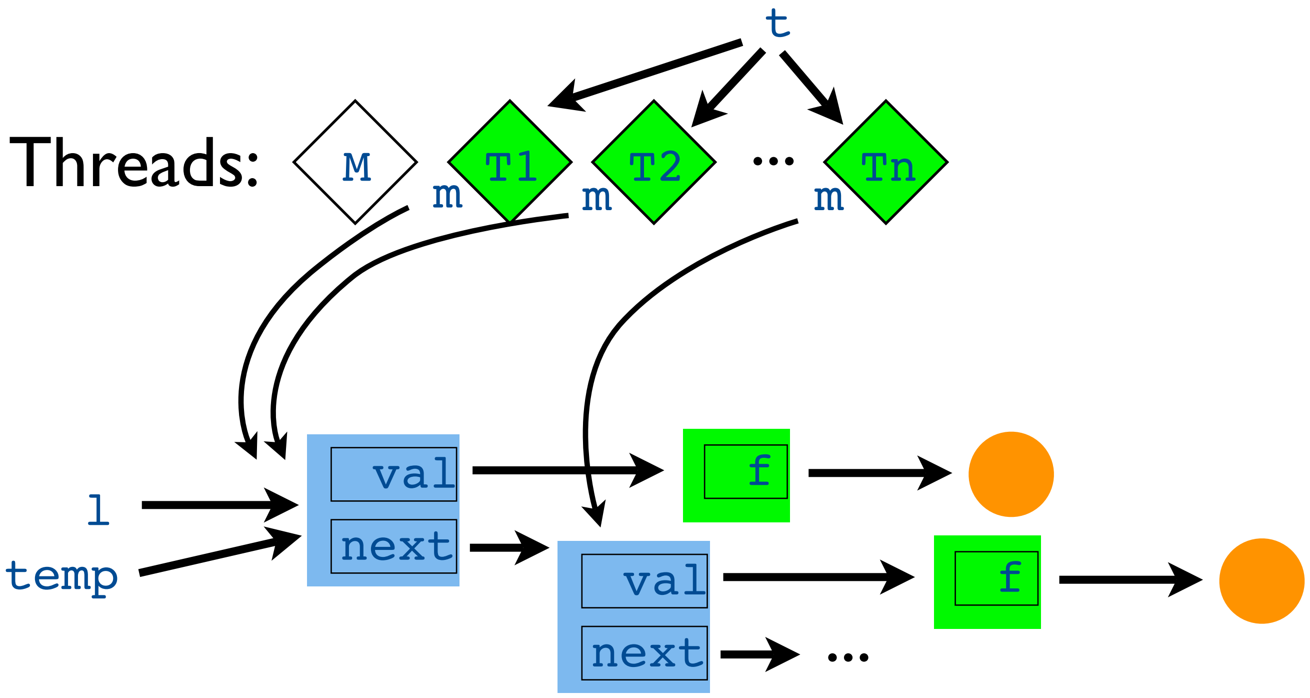
(8,f,8)

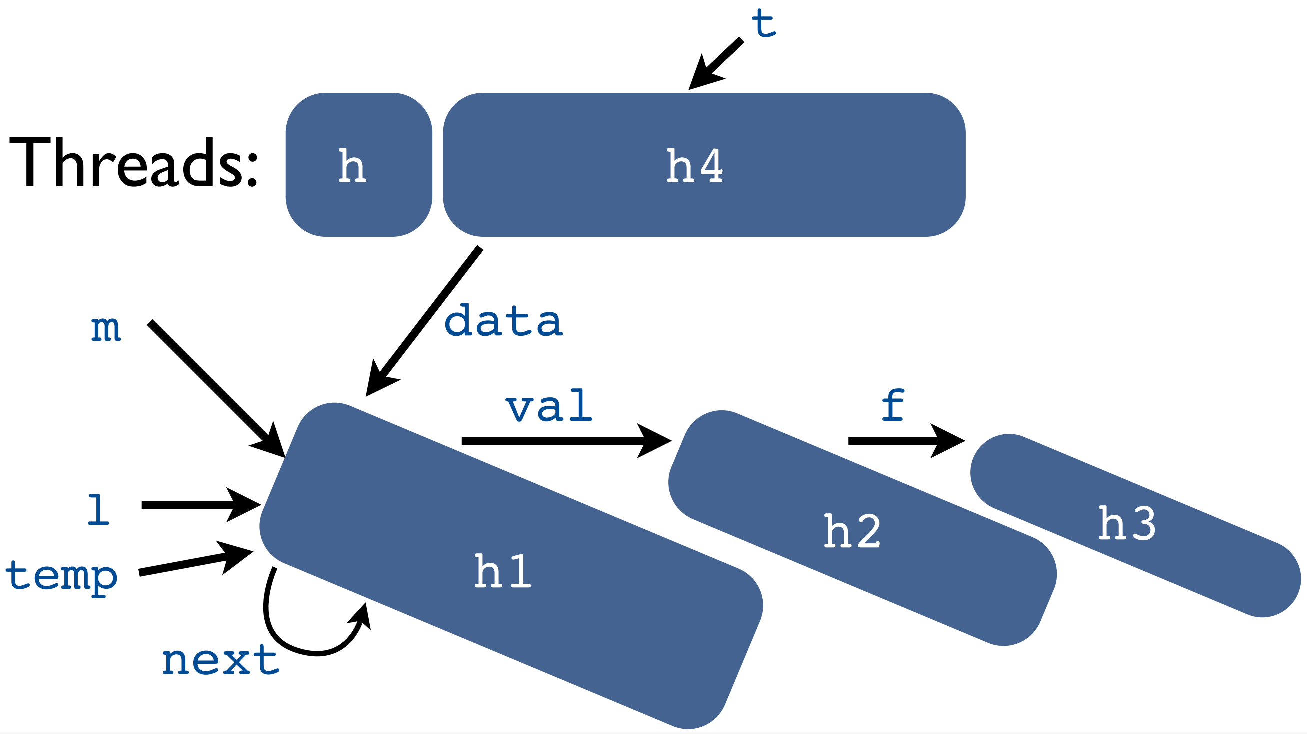


The last one...

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

- 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: $h \in \mathrm{DR}(H)$ *for* H *a set of allocation sites, if and only if whenever an object* o *allocated at site* h *may be reachable by one or more field dereferences from each of 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

- 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 instrumentation completely identify locations: two location tagged with the same loop counter must be equals
3. Define and prove correct a type and effect system that computes 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 of the disjoint reachability notion, using the previous type system.

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 a standard *may* information: the set of locations that may be targeted by a read or a write
- We mix all these analysis to remove races

$$F_\Sigma(R) = \{(ppt_1, f, ppt_2) \in R \mid \neg \begin{pmatrix} Must_1 \neq \emptyset \wedge Must_2 \neq \emptyset \\ \wedge\ May_1 \cap May_2 \subseteq DR_\Sigma(Must_1 \cup Must_2) \end{pmatrix} \}$$

$$\textit{where}\ \mathcal{P}(ppt_1, f) = (May_1, Must_1) \qquad \mathcal{P}(ppt_2, f) = (May_2, Must_2)$$

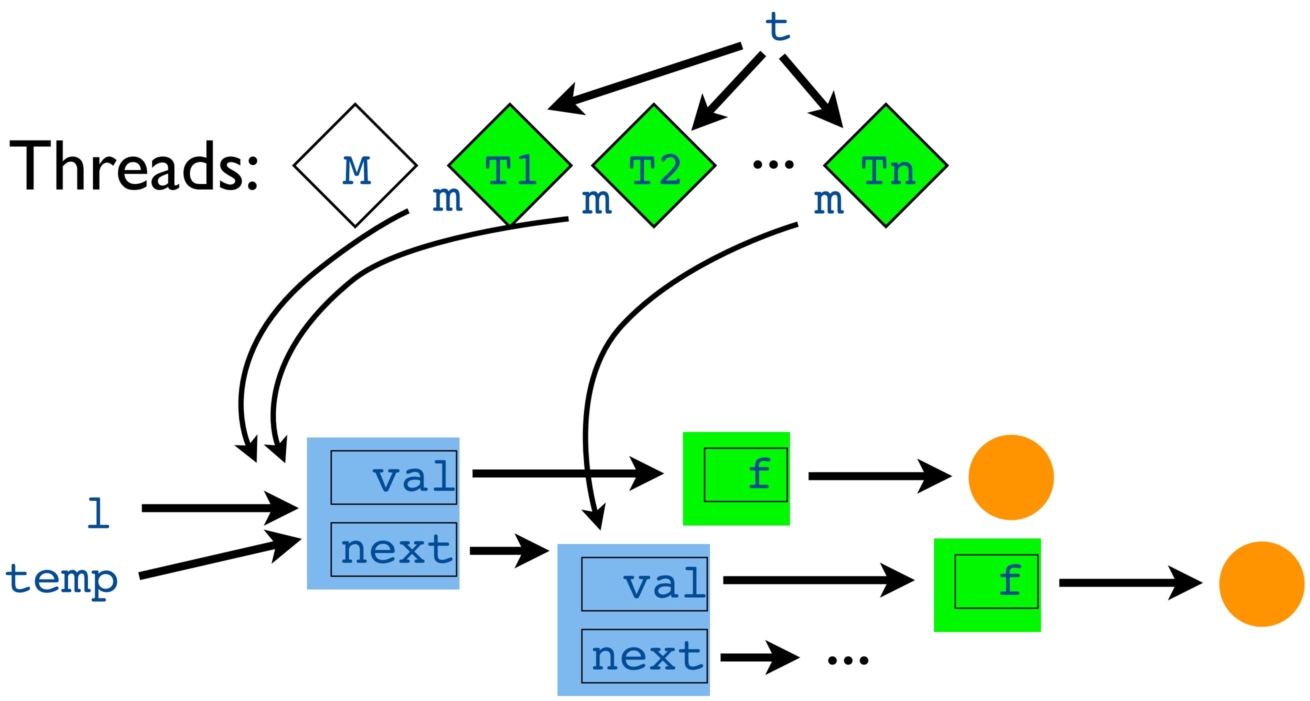
Running example

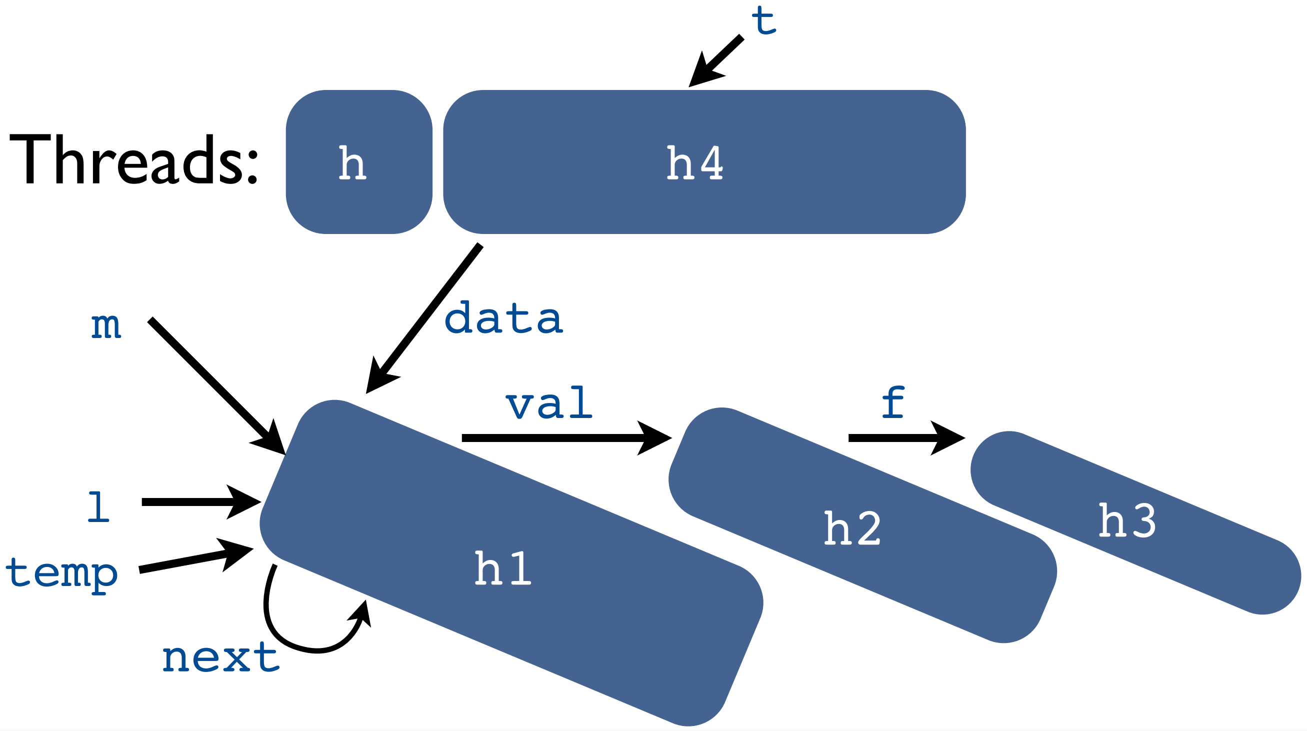
```
synchronize(m){ m.val.f = ...;}  ‖  synchronize(m){ m.val.f = ...;}
```

$$May_1 = May_2 = \{\mathbf{h2}\}$$
$$Must_1 = Must_2 = \{\mathbf{h1}\}$$
$$DR_\Sigma(\{\mathbf{h1}\}) = \{\mathbf{h2}\}$$

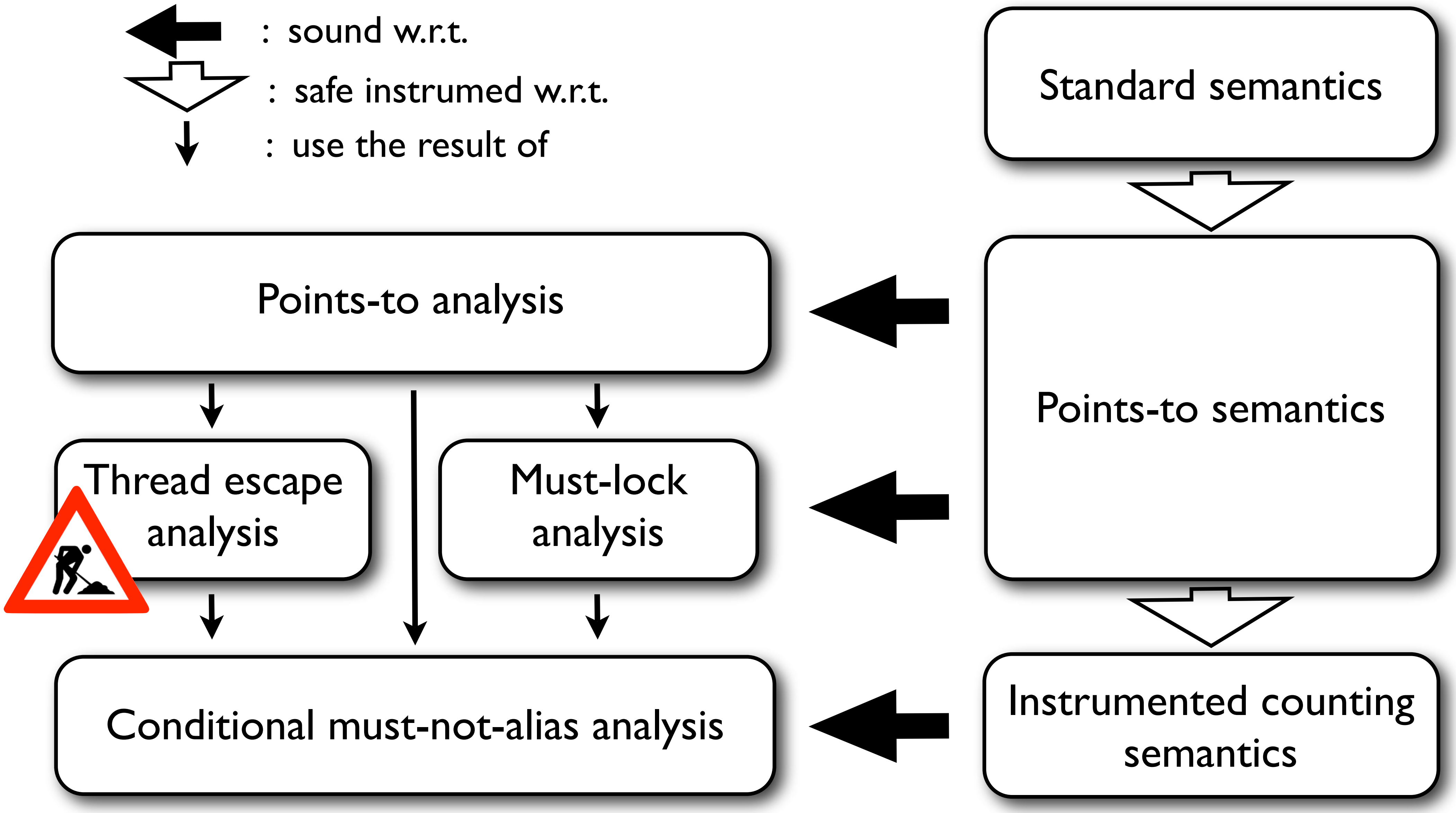
$$\Longrightarrow$$

$$Must_1 \neq \emptyset \wedge Must_2 \neq \emptyset \wedge$$
$$May_1 \cap May_2 \subseteq DR_\Sigma Must_1 \cap Must_2)$$





The big picture



⬅ : sound w.r.t.

⇩ : safe instrumed w.r.t.

↓ : use the result of

Standard semantics

Points-to analysis

Points-to semantics

Thread escape analysis

Must-lock analysis

Conditional must-not-alias analysis

Instrumented counting semantics

Conclusions

- 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 (15.000 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
    - We could refine the current formalisation to something executable