Modular verification of concurrent programs with heap

Alexey Gotsman
University of Cambridge

Joint work with Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv

Concurrent programs with heap

```
void t1394Diag_CancelIrp(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) {
    KIRQL Irql; PBUS_RESET_IRP BusResetIrp; PDEVICE_EXTENSI

    deviceExtension = DeviceObject->DeviceExtension;
    KeAcquireSpinLock(&deviceExtension->ResetSpinLock,

    BusResetIrp = (PBUS_RESET_IRP)deviceExtension->BusReset
    while (BusResetIrp) {
      if (BusResetIrp->Irp == Irp) {
        RemoveEntryList(&BusResetIrp->BusResetIrpList);
        ExFreePool(BusResetIrp);
        break;
      }
      else if (BusResetIrp->BusResetIrpList.Flink == &deviceExtension->BusResetIrps)
        break;
      else
        BusResetIrp = (PBUS_RESET_IRP)BusResetIrp->BusResetIrpList.Flink;
    }

    KeReleaseSpinLock(&deviceExtension->ResetSpinLock, Irql);
    IoReleaseCancelSpinLock(Irp->CancelIrql);
    Irp->IoStatus.Status = STATUS_CANCELLED;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
}
```

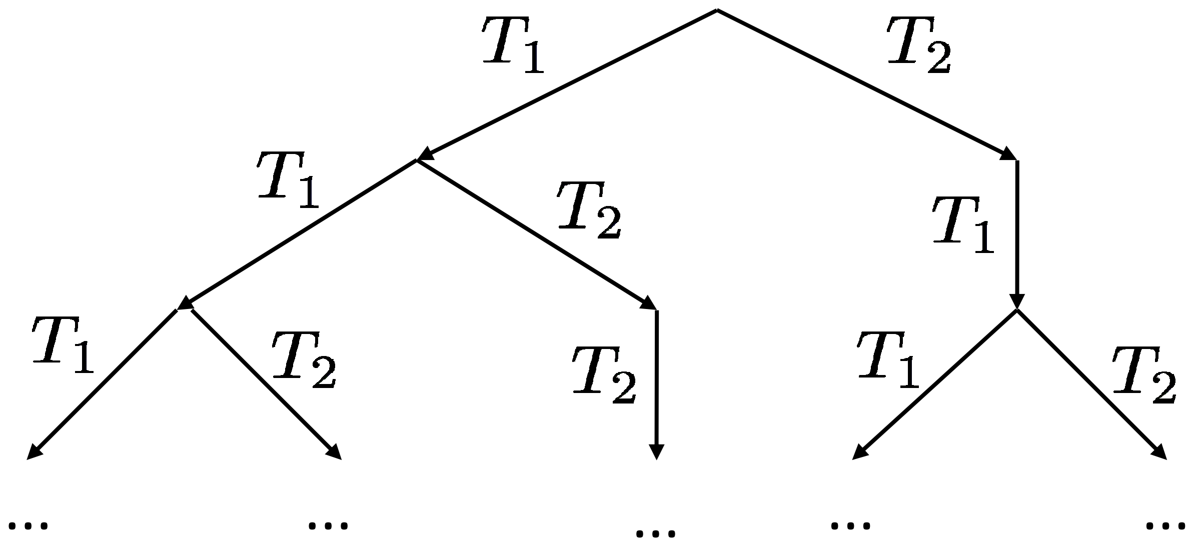
Is this a well-formed cyclic doubly-linked list?

Discovered by shape analyses

→ Properties: memory safety, absence of memory leaks

Verification of (asynchronous) concurrent programs

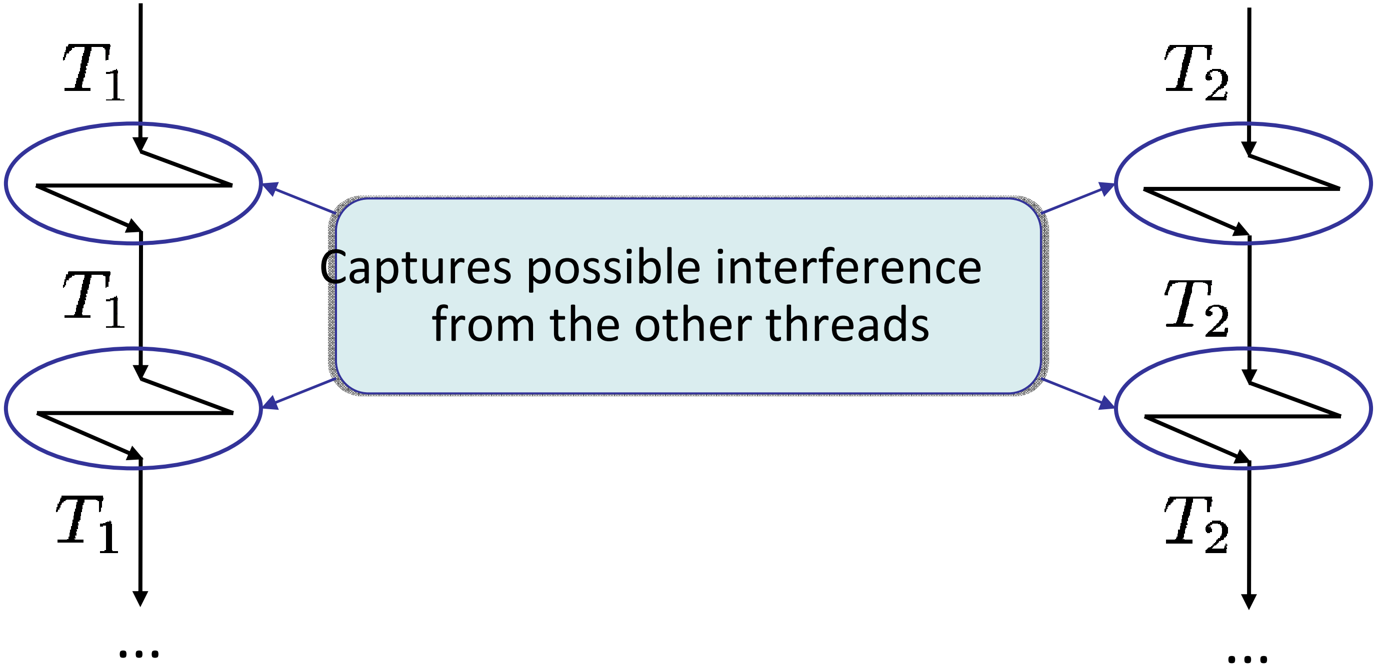
- Have to consider all possible interleavings/schedules:



- State-space explosion
- Heap-manipulation expands the set of possible thread interactions
- Multicores mean more concurrency

Thread-modular reasoning

- Consider every thread in isolation under some assumption on its environment



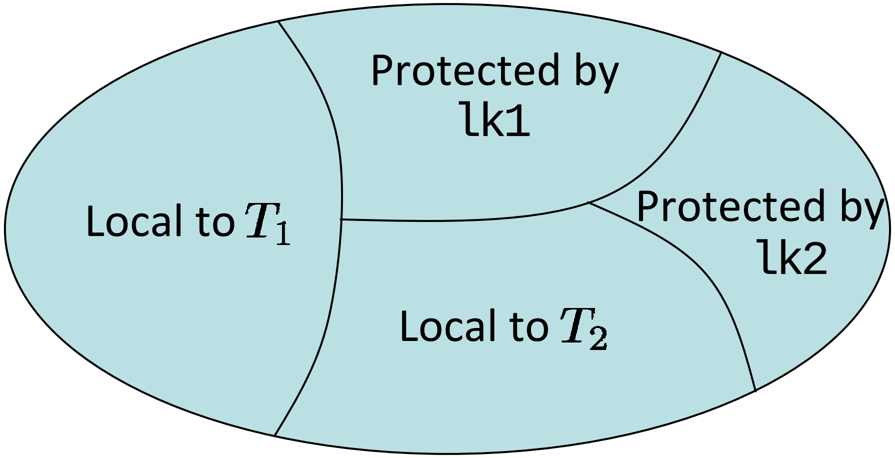
Captures possible interference from the other threads

- No direct enumeration of interleavings
- Existing methods focus on programs without dynamically-allocated memory
- This talk: a thread-modular shape analysis for concurrent programs based on concurrent separation logic

Concurrent separation logic [O'Hearn 2002]

Heap-manipulating programs with static locks and threads:

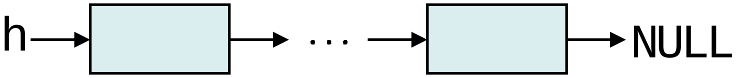
```
          LOCK lk1, lk2;

T1() {           || T2() {
  ...            ||   ...
  lock(lk2);     ||   lock(lk2);
  ...            ||   ...
  unlock(lk2);   ||   unlock(lk2);
  ...            ||   ...
}                || }
```



- → Allocated address space is partitioned into several disjoint parts:
  - thread-local parts: can be accessed only by the corresponding thread
  - parts protected by free locks
- → View enforced by the logic: not true of all programs
- → Benefit: never have to consider local states of other threads

Concurrent separation logic [O'Hearn 2002]

- → Every lock `lk` annotated with a resource invariant $I_{\mathtt{lk}}$ – a predicate on heaps:

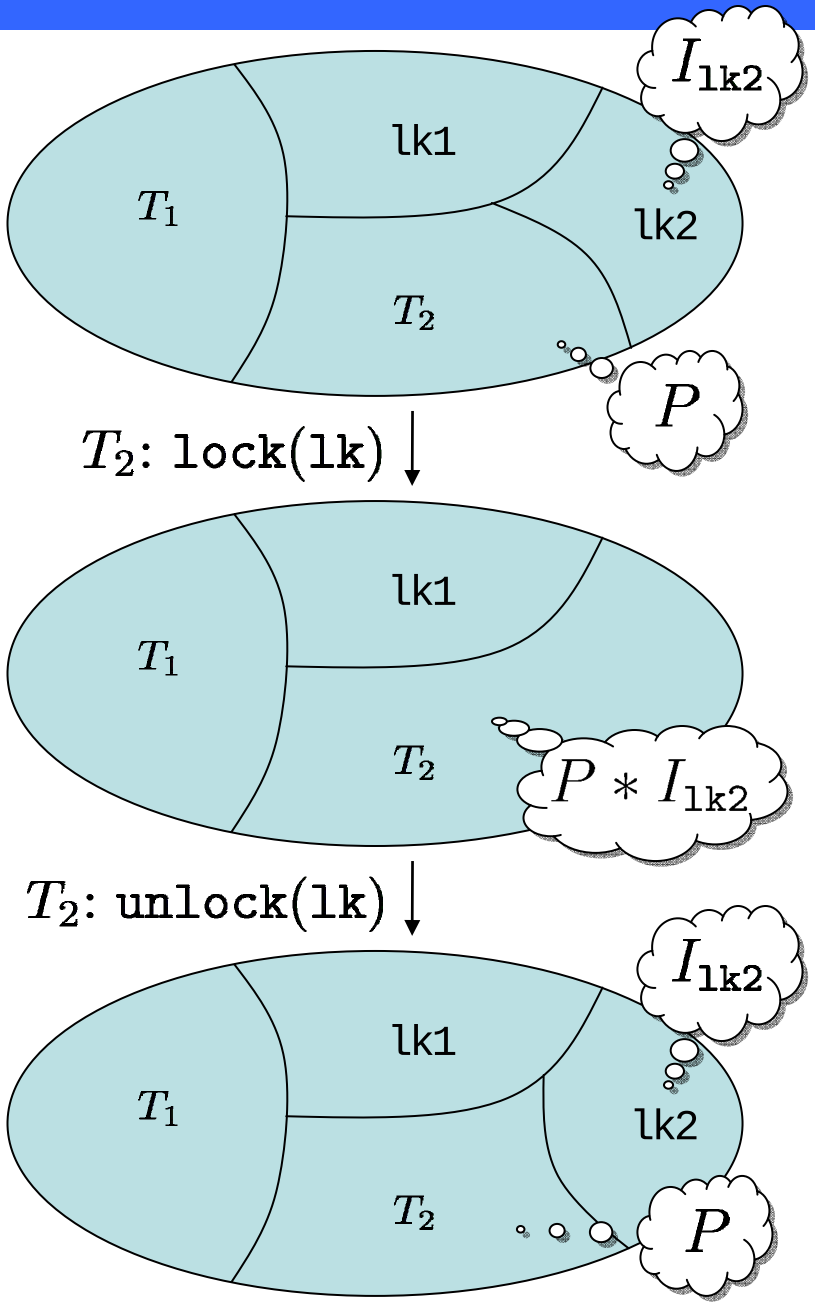


- → Hoare logic: $\{P\}\ C\ \{Q\}$

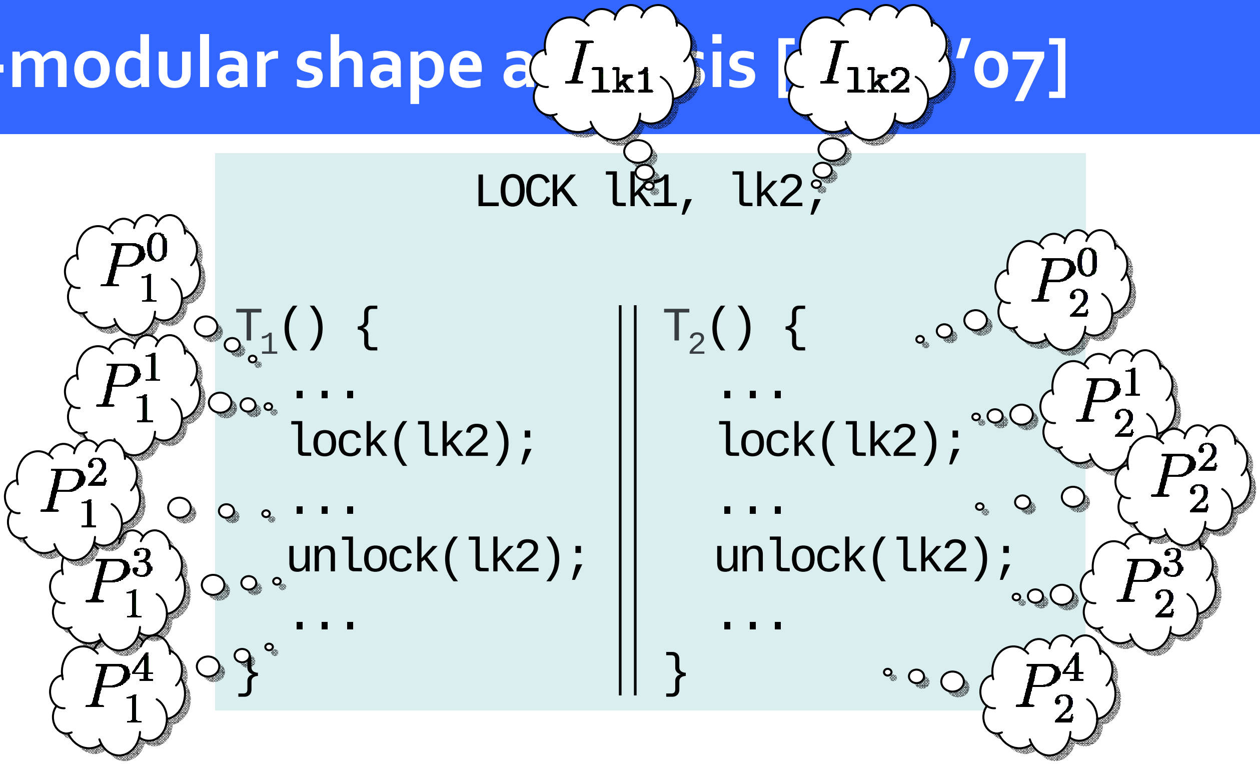
- → Axioms for `lock` and `unlock`:

$$\{P\}\ \mathtt{lock(lk)}\ \{P * I_{\mathtt{lk}}\}$$

$$\{P * I_{\mathtt{lk}}\}\ \mathtt{unlock(lk)}\ \{P\}$$



Thread-modular shape a sis [’07]



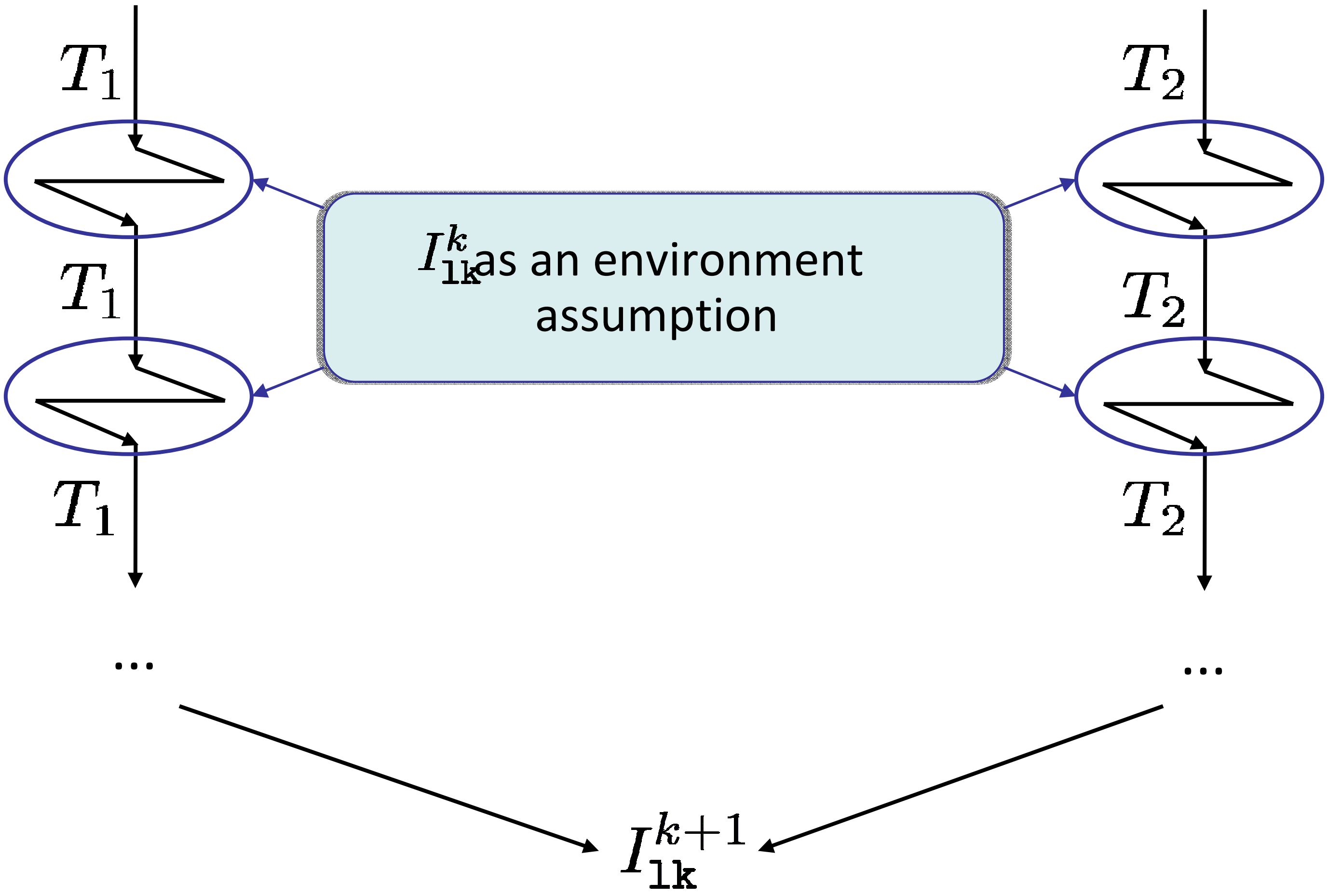
```
LOCK lk1, lk2;

T1() {              ||  T2() {
  ...               ||    ...
  lock(lk2);        ||    lock(lk2);
  ...               ||    ...
  unlock(lk2);      ||    unlock(lk2);
  ...               ||    ...
}                   ||  }
```

- Input:
  - Program with lock-based synchronisation (for now: static locks and threads)
  - Sequential abstract interpretation-based shape analysis (terms and conditions apply)
- Output:
  - Resource invariants for all locks
  - Local states of threads at all program points
  - Proves memory safety and data-race freedom
- Complexity:
  - Linear in the number of threads

Thread-modular shape analysis

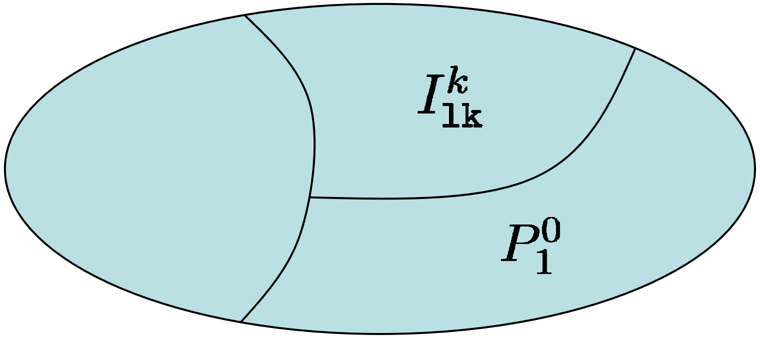
$$I^0_{\mathtt{1k}} \sqsubseteq I^1_{\mathtt{1k}} \sqsubseteq I^2_{\mathtt{1k}} \sqsubseteq \ldots$$

$$I^k_{\mathtt{1k}} \longrightarrow I^{k+1}_{\mathtt{1k}}:$$



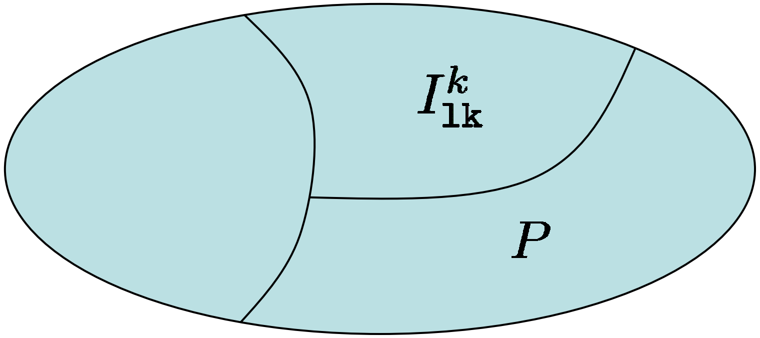
Analysing a thread

```
LOCK lk;   // $I^k_{\mathtt{lk}}$

T_1() {
  {$P^0_1$}
  ...

  lock(lk);


  ...
  ...
  ...

  unlock(lk);


  ...
}
```



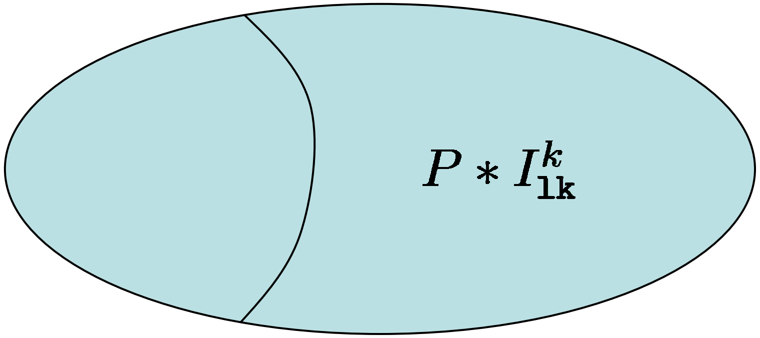
Analysing a thread

```
LOCK lk;   // $I_{\mathtt{lk}}^{k}$

T_1() {
  {$P_1^0$}
  ...
  {$P$}
  lock(lk);


  ...
  ...
  ...

  unlock(lk);


  ...
}
```



Analysing a thread

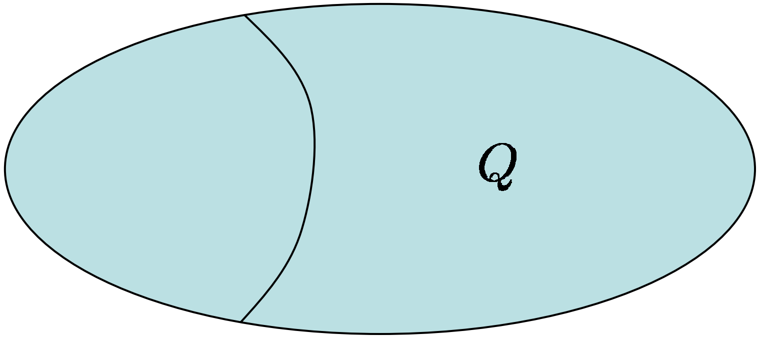
```
LOCK lk;  // $I^k_{\mathtt{lk}}$

T$_1$() {
  $\{P^0_1\}$
  ...
  $\{P\}$
  lock(lk);
  $\{P * I^k_{\mathtt{lk}}\}$
  ...
  ...
  ...

  unlock(lk);


  ...
}
```



→ lock: conjoin the current approximation $I^k_{\mathtt{lk}}$ of the resource invariant to the local state

Analysing a thread

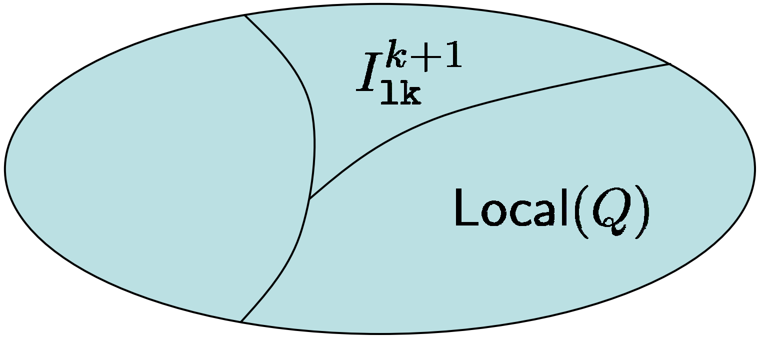
```
LOCK lk;   // $I^k_{\texttt{lk}}$

T₁() {
   {$P^0_1$}
   ...
   {$P$}
   lock(lk);
   {$P * I^k_{\texttt{lk}}$}
   ...
   ...
   ...
   {$Q$}
   unlock(lk);


   ...
}
```



→ lock: conjoin the current approximation $I^k_{\texttt{lk}}$ of the resource invariant to the local state

Analysing a thread

```
LOCK lk;   // $I^{k}_{\mathtt{lk}}$

$T_1$() {
  {$P^0_1$}
  ...
  {$P$}
  lock(lk);
  {$P * I^{k}_{\mathtt{lk}}$}
  ...
  ...
  ...
  {$Q = \mathsf{Local}(Q) * \mathsf{Protected}(Q)$}
  unlock(lk);
  {$\mathsf{Local}(Q)$}
  ...
}
```



→ `lock`: conjoin the current approximation $I^{k}_{\mathtt{lk}}$ of the resource invariant to the local state

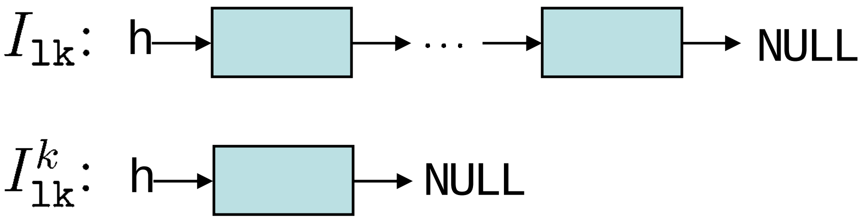
→ `unlock`: split the local state Q into two parts

- Local(Q): the new local state
- Protected(Q): the new approximation of the resource invariant
- Defined by application-specific heuristics

$$I^{k+1}_{\mathtt{lk}} = I^{k}_{\mathtt{lk}} \vee \mathsf{Protected}(Q)$$

Analysing a thread

```
LOCK lk;  // $I^k_{\mathtt{lk}}$

T₁() {
  $\{P^0_1\}$
  ...
  $\{P\}$
  lock(lk);
  $\{P * I^k_{\mathtt{lk}}\}$
  ...
  ...
  ...
  $\{Q = \mathsf{Local}(Q) * \mathsf{Protected}(Q)\}$
  unlock(lk);
  $\{\mathsf{Local}(Q)\}$
  ...
}
```

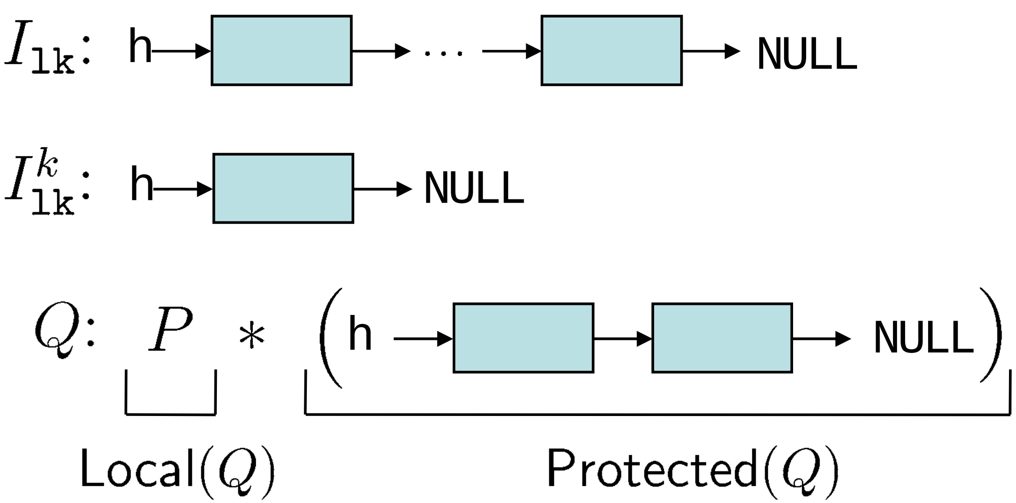


$$I^{k+1}_{\mathtt{lk}} = I^k_{\mathtt{lk}} \vee \mathsf{Protected}(Q)$$

Analysing a thread

```
LOCK lk;  // $I^k_{\mathtt{lk}}$

$T_1$() {
  {$P^0_1$}
  ...
  {$P$}
  lock(lk);
  {$P * I^k_{\mathtt{lk}}$}
  ...
  // Insert an entry
  ...
  {$Q = \mathsf{Local}(Q) * \mathsf{Protected}(Q)$}
  unlock(lk);
  {$\mathsf{Local}(Q)$}
  ...
}
```

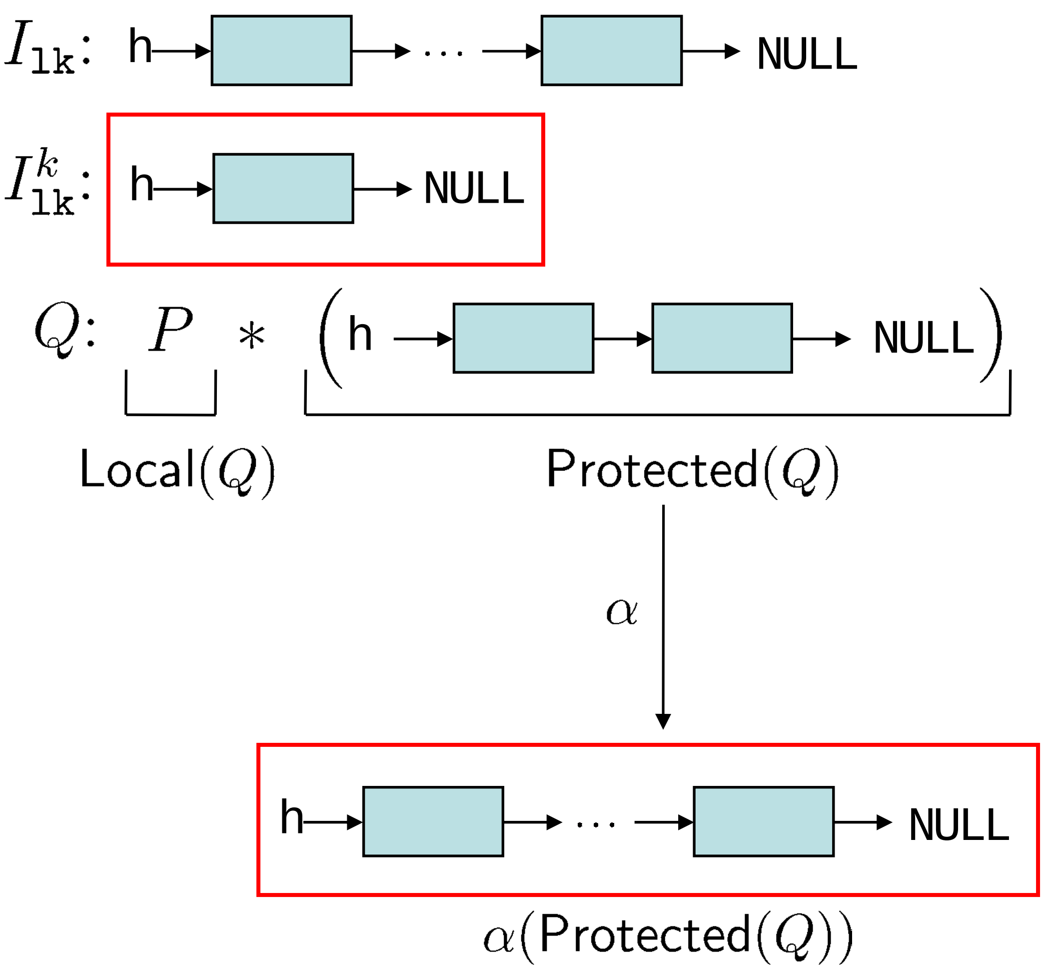
$$I^{k+1}_{\mathtt{lk}} = I^k_{\mathtt{lk}} \vee \mathsf{Protected}(Q)$$



- → Variables that correlate with the lock: variables accessed only when the lock is held [Pratikakis et al., 2006; Savage et al., 1997]
- → Protected(Q): the part of Q reachable from the variables that correlate with the lock
- → Similar heuristics for determining initial local states and resource invariants

Analysing a thread

```
LOCK lk;   // $I^k_{\mathtt{lk}}$

$T_1$() {
  $\{P^0_1\}$
  ...
  $\{P\}$
  lock(lk);
  $\{P * I^k_{\mathtt{lk}}\}$
  ...
  // Insert an entry
  ...
  $\{Q = \mathsf{Local}(Q) * \mathsf{Protected}(Q)\}$
  unlock(lk);
  $\{\mathsf{Local}(Q)\}$
  ...
}
```



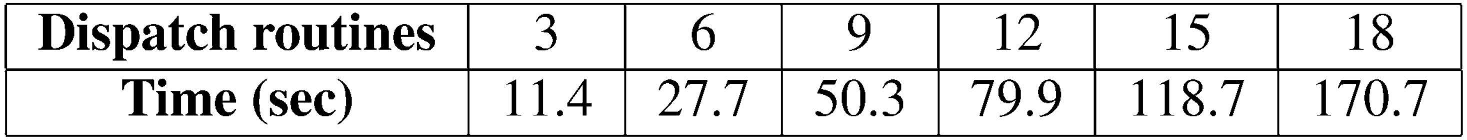
$$I^{k+1}_{\mathtt{lk}} = I^k_{\mathtt{lk}} \vee \alpha(\mathsf{Protected}(Q))$$

α – abstraction function of the sequential shape analysis

Implementation

- A sequential shape analysis based on separation logic for device driver data structures [Berdine et al., 2007]

- Firewire driver:



Dispatch routines	3	6	9	12	15	18
Time (sec)	11.4	27.7	50.3	79.9	118.7	170.7

- Part of the *SLAyer/Terminator* tool (Microsoft Research Cambridge): checks memory safety and liveness properties of device drivers

Back to the logic…

→How can we believe an analysis? Would like it to produce certificates – proofs in a program logic

→Results could be used in proof-carrying code or theorem proving systems

→Does the analysis compute proofs in concurrent separation logic?

→No: not all resource invariants I_{lk} are allowed!

Back to the logic…

$$\frac{\{P\}\ C\ \{Q_1\} \quad \{P\}\ C\ \{Q_2\}}{\{P\}\ C\ \{Q_1 \wedge Q_2\}}$$

→In concurrent separation logic resource invariants have to be precise: in any heap there may be at most one subheap satisfying the invariant

→Resource invariants computed by the analysis aren’t precise

→The underlying logic of the analysis has no conjunction rule and no precision restriction

→The variant of the logic and the analysis proved sound together

What about dynamically-allocated locks?

```
lk = new LOCK;
:
init(lk);
:
lock(lk);
:
unlock(lk);
:
finalize(lk);
:
delete lk;
```

→Unbounded numbers of locks – a finite number of invariants

→Abstract domain extended with elements representing locks with a given invariant

→Concurrent separation logic extended appropriately [APLAS’07]

What about dynamic thread creation?

```
for (i = 0; i < n; i++) {
  t[i] = fork(proc, i);
}
⋮
for (i = 0; i < n; i++) {
  join(t[i]);
}
```

- →Can use algorithms for interprocedural heap analysis [SAS'06]
- →Part of the heap reachable from `fork`'s parameters transferred to the thread
- →Concurrent separation logic extended appropriately [APLAS'07]

What about non-blocking and fine-grained concurrency?

- Thread-modular analysis works well on programs with coarse-grained synchronisation: one lock per data structure

- Fine-grained concurrency: multiple locks per data structure

- Non-blocking concurrency: lower-level synchronisation techniques

- Non-blocking and fine-grained concurrency need relations to describe interference
  - Combination of rely-guarantee and separation logic [Vafeiadis & Parkinson 2007; Feng, Ferreira & Shao 2007]
  - Shape analysis for non-blocking and fine-grained algorithms [Vafeiadis 2009]

Thread-modular shape analysis

→Efficient

unlike enumerating interleavings

→Sound and precise

unlike most race-detection analyses

→Handles ownership transfer

unlike ownership type systems

→Fully automatic

unlike systems based on VC generation