# Static Analysis of Race-Free Interrupt-Driven Programs

Deepak D'Souza

Department of Computer Science and Automation
Indian Institute of Science, Bangalore.

FM Update, BITS Goa, 19 July 2018.

*Joint work with Nikita Chopra and Rekha Pai*

## Outline

1. Data Flow Analysis
2. Concurrent Programs
3. Race-Free Programs
4. Sync-CFG Analysis
5. Analysis

# Data-Flow Analysis / Abstract Interpretation

- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

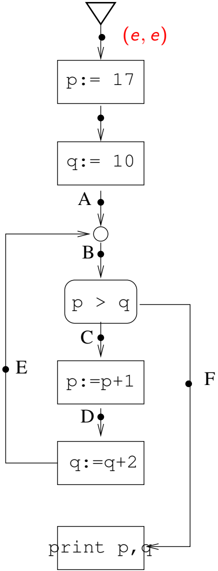
  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10 \rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e \rangle$.
- Interpret execution along a path by transforming the abstract state.

```
p := 17;
q := 10;
while (p > q) {
  p := p + 1;
  q := q + 2;
}
print p, q;
```

# Data-Flow Analysis / Abstract Interpretation

- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

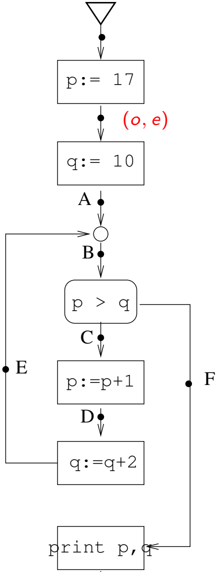
  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10 \rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e \rangle$.
- Interpret execution along a path by transforming the abstract state.



# Data-Flow Analysis / Abstract Interpretation

- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

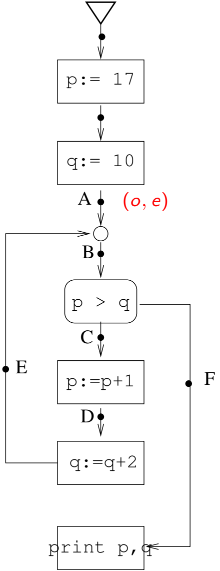
  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10 \rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e \rangle$.
- Interpret execution along a path by transforming the abstract state.



# Data-Flow Analysis / Abstract Interpretation

- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

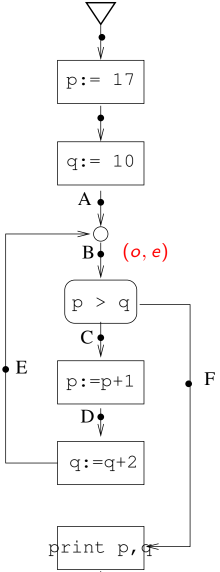
  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10 \rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e \rangle$.
- Interpret execution along a path by transforming the abstract state.



## Data-Flow Analysis / Abstract Interpretation

- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

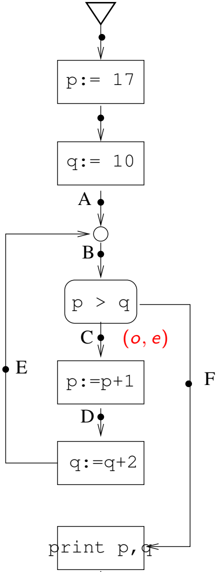
  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10\rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e\rangle$.
- Interpret execution along a path by transforming the abstract state.



# Data-Flow Analysis / Abstract Interpretation

- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

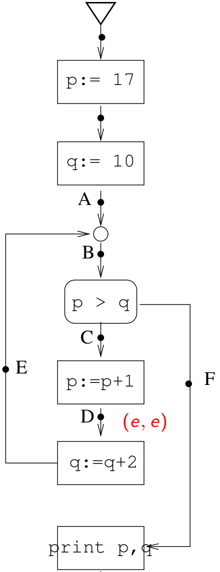
  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10\rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e\rangle$.
- Interpret execution along a path by transforming the abstract state.



# Data-Flow Analysis / Abstract Interpretation

- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

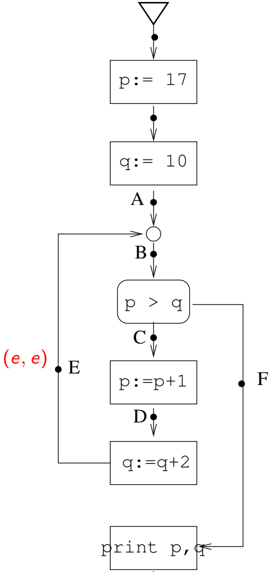
  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10\rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e\rangle$.
- Interpret execution along a path by transforming the abstract state.



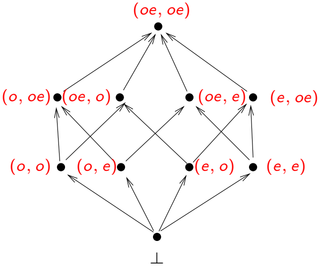
# Data-Flow Analysis / Abstract Interpretation

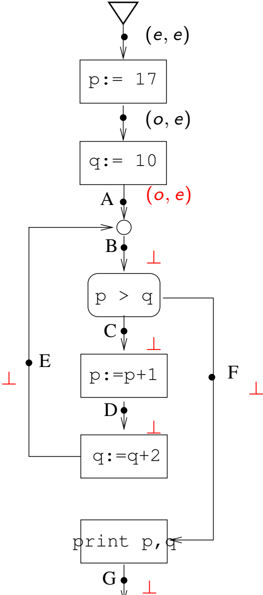
- Aim: To obtain conservative facts about the program state at each program point.
- Use abstract states to represent the concrete state.

  Example:
  Concrete state: $\langle p \mapsto 17, q \mapsto 10 \rangle$
  Abstract state: $\langle p \mapsto o, q \mapsto e \rangle$.
- Interpret execution along a path by transforming the abstract state.



# Computing JOP/LFP

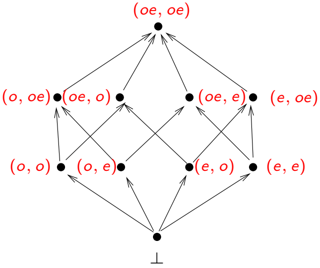




- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

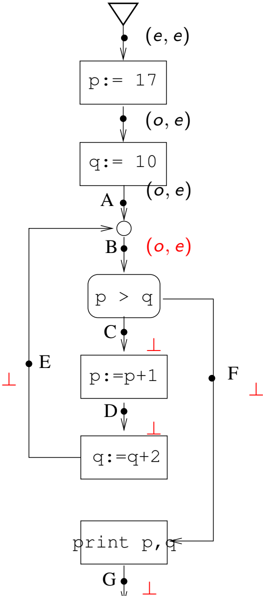
  *number of program points* × *height of abstract lattice.*

# Computing JOP/LFP

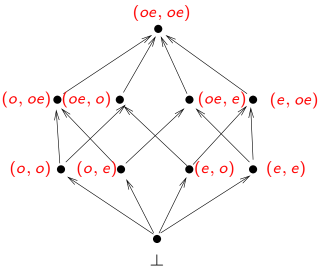


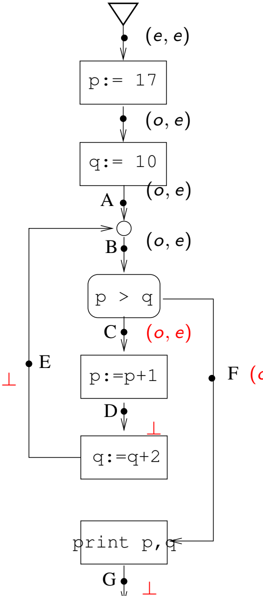
- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points* × *height of abstract lattice.*



# Computing JOP/LFP

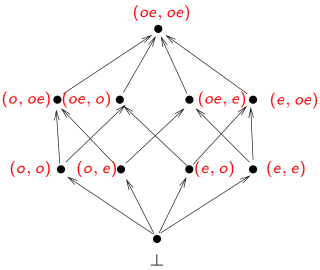


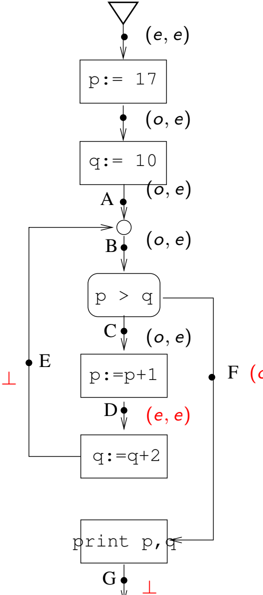


- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points* × *height of abstract lattice.*

# Computing JOP/LFP

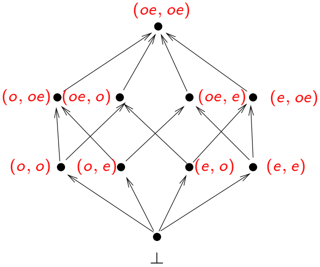




- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

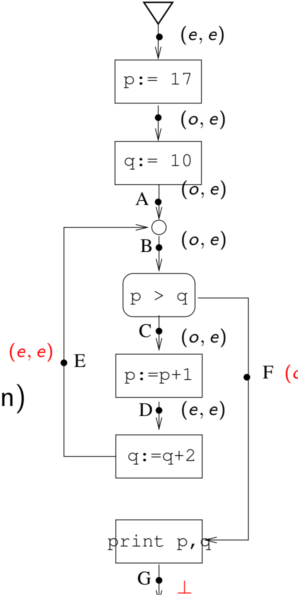
  *number of program points* × *height of abstract lattice.*

# Computing JOP/LFP

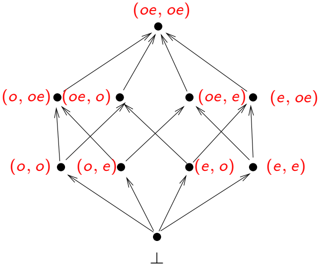


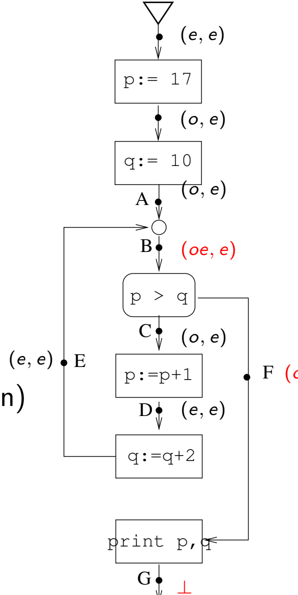
- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points* × *height of abstract lattice.*



# Computing JOP/LFP

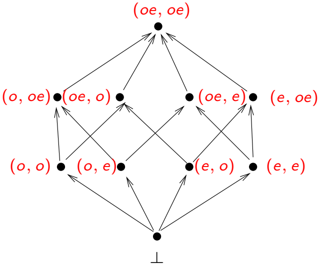


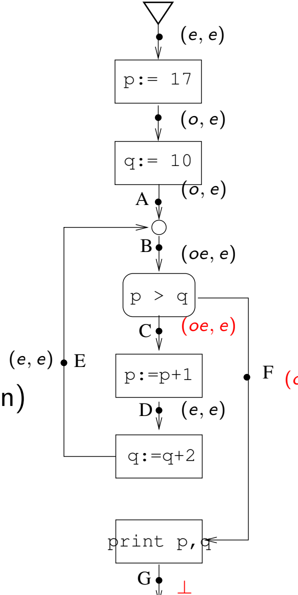


- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points* × *height of abstract lattice.*

# Computing JOP/LFP

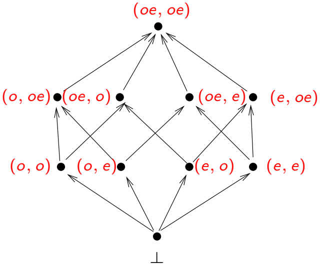


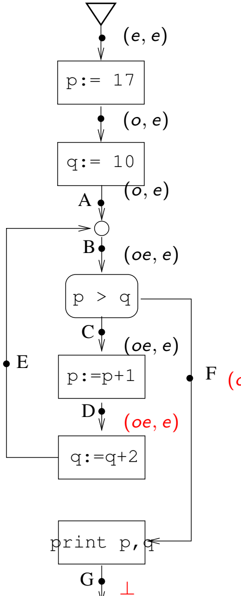


- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points* × *height of abstract lattice.*

# Computing JOP/LFP

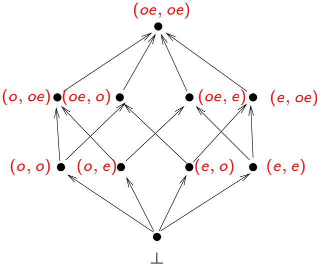


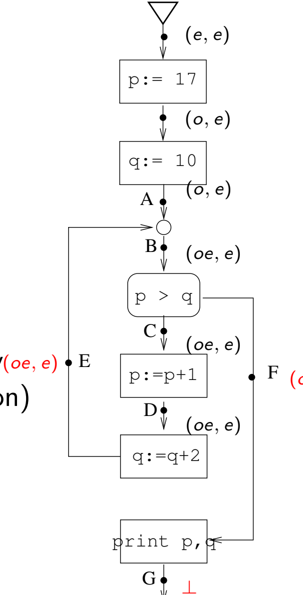


- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points × height of abstract lattice.*

# Computing JOP/LFP

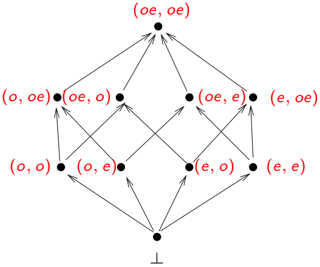


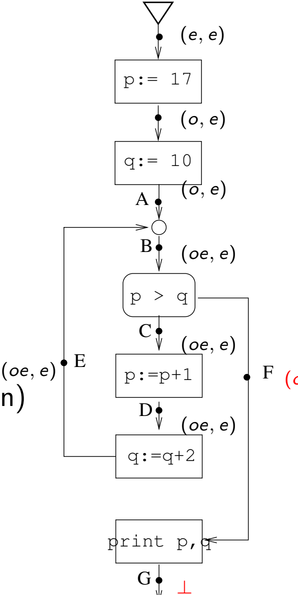


- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points* × *height of abstract lattice.*

# Computing JOP/LFP

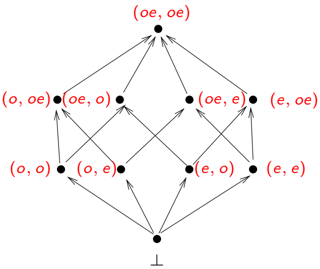


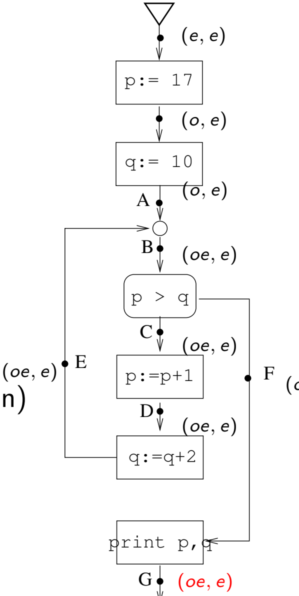


- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points* × *height of abstract lattice.*

# Computing JOP/LFP

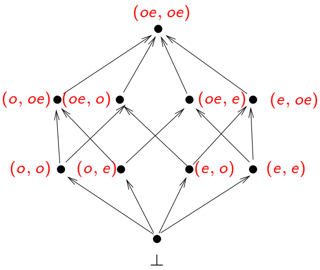


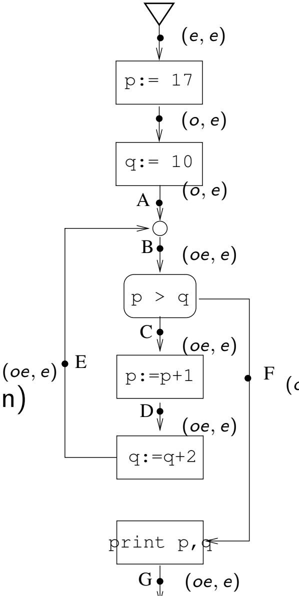


- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points × height of abstract lattice.*

# Computing JOP/LFP





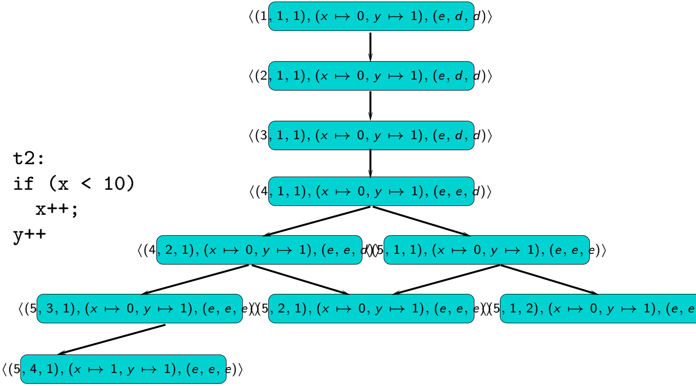
- We usually further over-approximate the JOP by computing the least fixpoint (LFP) (least solution) of data-flow equations.
- The number of steps in the LFP computation is bounded by

  *number of program points × height of abstract lattice.*

# Multi-Threaded Programs

## Standard interleaving semantics

```
   main:
1. x := 0;
2. y := 0;
3. spawn(t1);
4. spawn(t2);
5.

   t1:                 t2:
1. if (x < 10)      1. if (x < 10)
2.   x++;           2.   x++;
3. y++              3. y++
4.                  4.
```



$\langle(1, 1, 1), (x \mapsto 0, y \mapsto 1), (e, d, d)\rangle$

$\langle(2, 1, 1), (x \mapsto 0, y \mapsto 1), (e, d, d)\rangle$

$\langle(3, 1, 1), (x \mapsto 0, y \mapsto 1), (e, d, d)\rangle$

$\langle(4, 1, 1), (x \mapsto 0, y \mapsto 1), (e, e, d)\rangle$

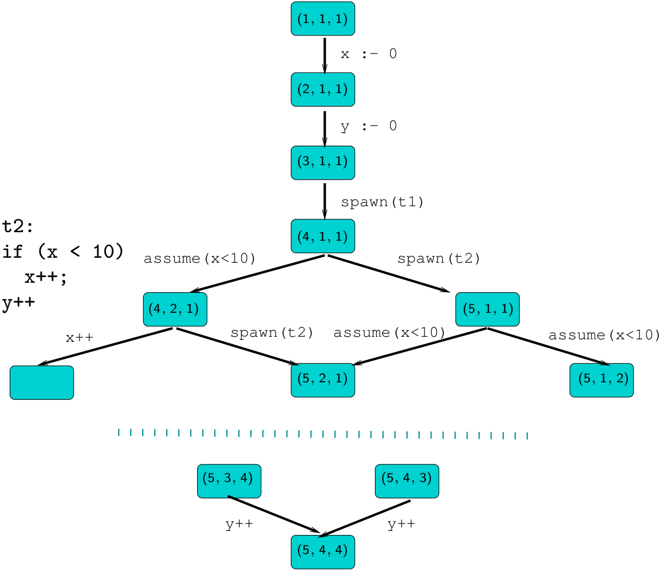
$\langle(4, 2, 1), (x \mapsto 0, y \mapsto 1), (e, e, d)\rangle$ $\langle(5, 1, 1), (x \mapsto 0, y \mapsto 1), (e, e, e)\rangle$

$\langle(5, 3, 1), (x \mapsto 0, y \mapsto 1), (e, e, e)\rangle$ $\langle(5, 2, 1), (x \mapsto 0, y \mapsto 1), (e, e, e)\rangle$ $\langle(5, 1, 2), (x \mapsto 0, y \mapsto 1), (e, e$

$\langle(5, 4, 1), (x \mapsto 1, y \mapsto 1), (e, e, e)\rangle$

# Product Control Flow Graph

```
   main:
1. x := 0;
2. y := 0;
3. spawn(t1);
4. spawn(t2);
5.

   t1:              t2:
1. if (x < 10)   1. if (x < 10)
2.   x++;        2.   x++;
3. y++           3. y++
4.               4.
```



## Data Flow Analysis for a Concurrent Program

Naive approach:

- Construct Product CFG
- Carry out analysis on this graph

Approach is precise, but too expensive! Problem: If number of threads is $k$, height of lattice is $h$, and number of program points in a thread is $n$, then

- Number of program points in product CFG is $n^k$.
- Number of iterations is bounded by

$$h \times n^k$$

- Time taken can be exponential in number of threads.

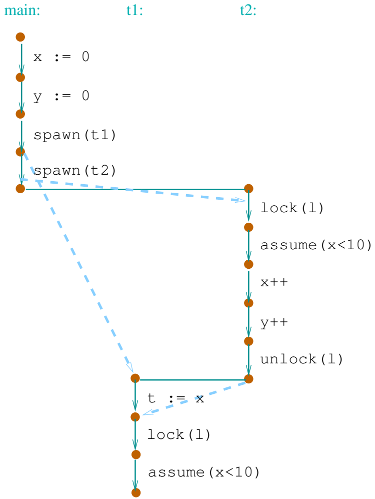
Can we be more efficient for some class of programs, maybe at the cost of precision?

## Happens-Before Race

- Happens-Before ordering on instructions in an execution:
    - synchronizes-with relation: Two instructions $I$ and $J$ in an execution are sync-with related if $I$ is a release (like `unlock(l)`) and $J$ is the next corresponding acquire (like `lock(l)`).
    - Program-Order relation.
    - HB order is the reflexive transitive closure of the union of program-order and sync-with relations.
- Two instructions in an execution are involved in a HB-race if they are conflicting accesses and are unordered by the the HB order.

# Illustrating Happens-Before Race

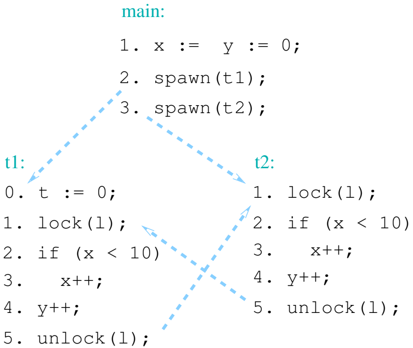
```
main:
x := 0;
y := 0;
spawn(t1);
spawn(t2);

t1:                t2:
t := x;            lock(l);
lock(l);           if (x < 10)
if (x < 10)          x++;
  x++;             y++;
y++;               unlock(l);
unlock(l);
```



# Sync-CFG Analysis for HB-Race-Free Programs [De, D, Nasre 2011]

- Given a HB-Race-Free program
- Build a Sync-CFG for the program
  - Union of CFG's of each thread
  - May-Sync-With edges to conservatively capture sync-with relation.
- Perform a Value-Set analysis.
- LFP values for a variable are guaranteed to be sound at points where the variable is owned by the thread.

# Example Sync-CFG



main:

```
1. x :=  y := 0;
2. spawn(t1);
3. spawn(t2);
```

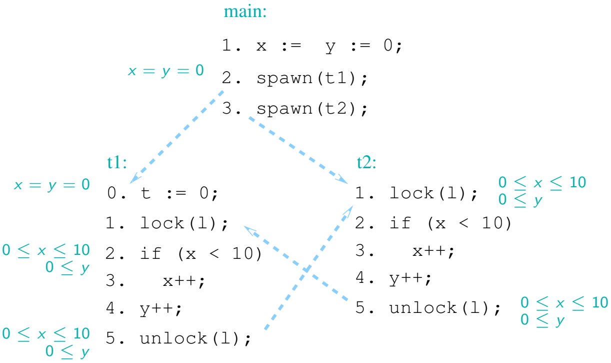
t1:

```
0. t := 0;
1. lock(l);
2. if (x < 10)
3.   x++;
4. y++;
5. unlock(l);
```

t2:

```
1. lock(l);
2. if (x < 10)
3.   x++;
4. y++;
5. unlock(l);
```

# Example Sync-CFG with Value-Set Analysis



main:

```
1. x :=  y := 0;
2. spawn(t1);
3. spawn(t2);
```

$x = y = 0$

t1:

```
0. t := 0;
1. lock(l);
2. if (x < 10)
3.   x++;
4. y++;
5. unlock(l);
```

$x = y = 0$

$0 \leq x \leq 10$
$0 \leq y$

$0 \leq x \leq 10$
$0 \leq y$

t2:

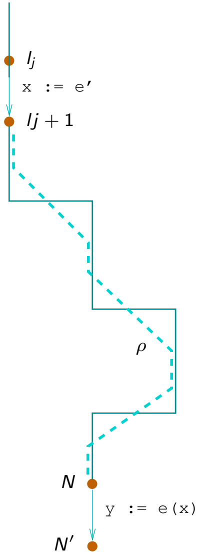
```
1. lock(l);
2. if (x < 10)
3.   x++;
4. y++;
5. unlock(l);
```

$0 \leq x \leq 10$
$0 \leq y$

$0 \leq x \leq 10$
$0 \leq y$

## Soundess Claim and Proof

Claim: Let $P$ be a HB-race-free program. Consider the final data-flow facts in the Value-Set analysis for $P$. Suppose variable $x$ is owned by thread $t$ at point $N$. Consider an execution reaching $N$ with $x$ having value $v$. Then $v$ belongs to the value set of $x$ at $N$.



## Shortcomings and Extensions

- Can be imprecise due to following reasons:
  - No relational information (like $x \leq y$).
  - Spurious loops ($y$ is unbounded).
- Some extensions
  - Use regions of variables (like $\{x, y\}$) which are similarly protected, and compute a value-set for the region (can get $x \leq y$).
  - Define a relational sync-cfg based semantics which is sound and complete (Mukherjee et al 2017). This gives us a variety of relational analyses.
  - Can handle programs with races (havoc reads of variables involved in a race)

## Shortcomings and Extensions

- Can be imprecise due to following reasons:
    - No relational information (like $x \leq y$).
    - Spurious loops ($y$ is unbounded).
- Some extensions
    - Use regions of variables (like $\{x, y\}$) which are similarly protected, and compute a value-set for the region (can get $x \leq y$).
    - Define a relational sync-cfg based semantics which is sound and complete (Mukherjee et al 2017). This gives us a variety of relational analyses.
    - Can handle programs with races (havoc reads of variables involved in a race)

How do we extend this Sync-CFG based analysis to programs with non-standard concurrency? What is the notion of a race, sync-with relation, HB order, etc?

# Abstracted version of Send/ReceiveISR Methods

main:

```
msgw := 0;
len := 10;
wtosend := 0;
wtorec := 0;
RxLock := 0;
create(qsend);
create(qrec_ISR);
```

qsend:

```
disableint;
if(msgw < len) {
  msgw++;
  if(wtorec > 0)
    wtorec--;
  enableint;
}
else {
  enableint;
  suspendsch;
  disableint;
  RxLock++;
  enableint;
  wtosend++;
  disableint;
  while(RxLock > 1) {
    if(wtosend > 0)
      wtosend--;
    RxLock--;
  }
  RxLock := 0;
  enableint;
  resumesch;
}
```

qrec_ISR:

```
if(msgw > 0) {
  msgw--;
  if(RxLock = 0) {
    if(wtosend > 0)
      wtosend--;
  }
  else
    RxLock++;
}
```

# Disjoint blocks with locks

main:

```
1. x :=  y := 0;
2. spawn(t1);
3. spawn(t2);
```

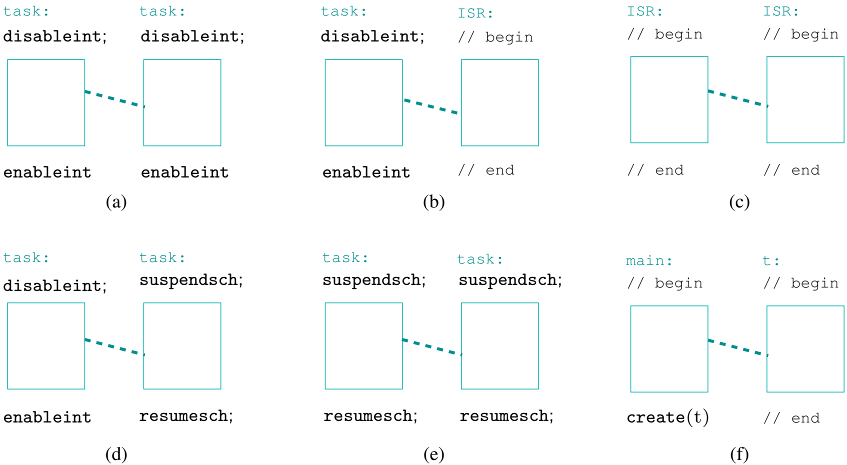
t1:

```
0. t := 0;
1. lock(l);
2. if (x < 10)
3.   x++;
4. y++;
5. unlock(l);
```

t2:

```
1. lock(l);
2. if (x < 10)
3.   x++;
4. y++;
5. unlock(l);
```

# Disjoint Blocks



task: disableint; … enableint task: disableint; … enableint

(a)

task: disableint; … enableint ISR: // begin … // end

(b)

ISR: // begin … // end ISR: // begin … // end

(c)

task: disableint; … enableint task: suspendsch; … resumesch;

(d)

task: suspendsch; … resumesch; task: suspendsch; … resumesch;

(e)

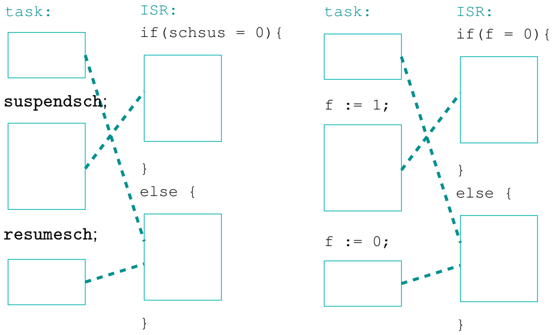
main: // begin … create(t) t: // begin … // end

(f)

enableint resumesch; resumesch; resumesch; create(t) // end

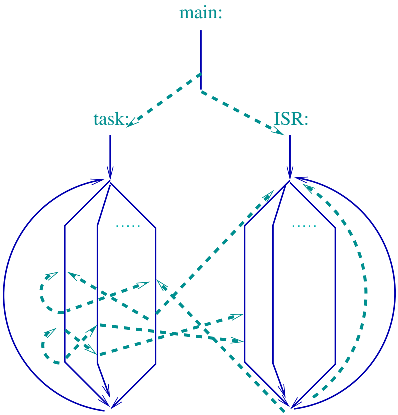
(d) (e) (f)

# Disjoint Blocks

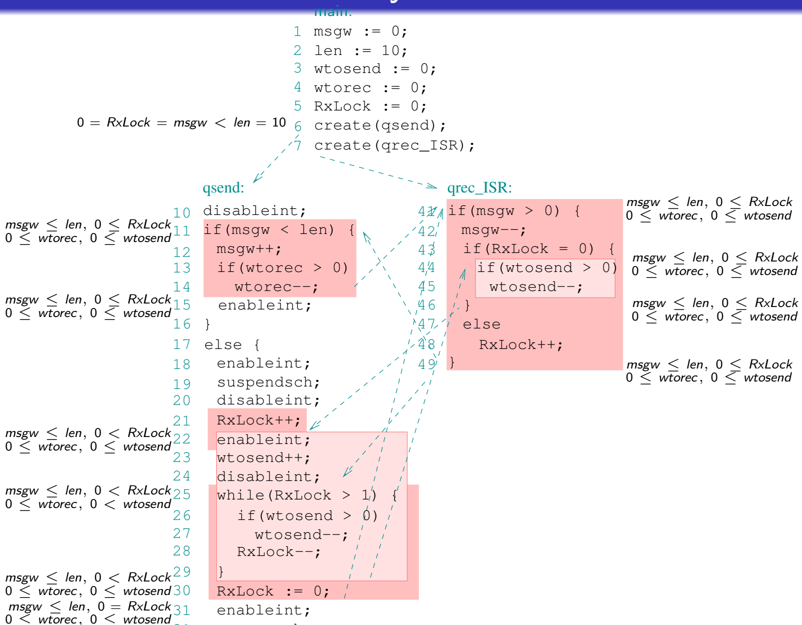


(g) (h)

# Sync-CFG induced by FreeRTOS kernel



# Sync-CFG and the Value-Set analysis on it



main:

```
msgw := 0;
len := 10;
wtosend := 0;
wtorec := 0;
RxLock := 0;
create(qsend);
create(qrec_ISR);
```

$0 = RxLock = msgw < len = 10$

qsend:

```
disableint;
if(msgw < len) {
  msgw++;
  if(wtorec > 0)
    wtorec--;
  enableint;
}
else {
  enableint;
  suspendsch;
  disableint;
  RxLock++;
  enableint;
  wtosend++;
  disableint;
  while(RxLock > 1) {
    if(wtosend > 0)
      wtosend--;
    RxLock--;
  }
  RxLock := 0;
  enableint;
```

$msgw \leq len$, $0 \leq RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

$msgw \leq len$, $0 \leq RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

$msgw \leq len$, $0 < RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

$msgw \leq len$, $0 < RxLock$ $0 \leq wtorec$, $0 < wtosend$

$msgw \leq len$, $0 < RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

$msgw \leq len$, $0 = RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

qrec_ISR:

```
if(msgw > 0) {
  msgw--;
  if(RxLock = 0) {
    if(wtosend > 0)
      wtosend--;
  }
  else
    RxLock++;
}
```

$msgw \leq len$, $0 \leq RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

$msgw \leq len$, $0 \leq RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

$msgw \leq len$, $0 \leq RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

$msgw \leq len$, $0 \leq RxLock$ $0 \leq wtorec$, $0 \leq wtosend$

# Octagon/Polyhedral Analysis on FreeRTOS sync-CFG

| Assertion | Interval Analysis | Region Analysis (Octagon/Polyhedra) |
|---|---|---|
| xTickCount $\leq$ xNextTaskUnblockTime | × | √ |
| head(pxDelayedTaskList) $=$ xNextTaskUnblockTime | × | √ |
| head(pxDelayedTaskList) $\geq$ TickCount | × | √ |
| uxMessagesWaiting $\leq$ uxLength | × | √ |
| uxMessagesWaiting $\geq$ 0 | √ | √ |
| uxCurrentNumberOfTasks $\geq$ 0 | √ | √ |
| lenpxReadyTasksLists $\geq$ 0 | √ | √ |
| uxTopReadyPriority $\geq$ 0 | √ | √ |
| lenpxDelayedTaskList $\geq$ 0 | √ | √ |
| lenxPendingReadyList $\geq$ 0 | √ | √ |
| lenxSuspendedTaskList $\geq$ 0 | √ | √ |
| cRxLock $\geq$ -1 | √ | √ |
| cTxLock $\geq$ -1 | √ | √ |
| lenxTasksWaitingToSend $\geq$ 0 | √ | √ |
| lenxTasksWaitingToReceive $\geq$ 0 | √ | √ |

## Why a lock translation does not work

Why not

- Translate interrupt-driven program $P$ to classical lock-based $P^L$, which captures interleaved executions of $P$.
- Now do race-detection and sync-CFG analysis on $P^L$.

# Races may not be preserved

```
main:
1. x := y := t := 0;
2. create(t1);
3. create(t2);

t1:                  t2:
4. x := x + 1;       8. disableint;
5. disableint;       9. t := x;
6. x := y;          10. enableint;
7. enableint;
```

Program $P$

```
main:
1. x := y := t := 0;
2. spawn(t1);
3. spawn(t2);

t1:                  t2:
4. lock(E)           10. lock(E);
5. x := x + 1;       11. t := x;
6. unlock(E)         12. unlock(E);
7. lock(E)
8. x := y;
9. unlock(E)
```

Execution preserving translation $P^L$

# Sync-CFG may be too imprecise

## Our Translation

Our approach can be viewed as giving a weak lock-based traslation $P$ to $P^W$ which:

- Does not attempt to preserve execution semantics (allows more executions than original program)
- Preserves disjoint blocks, hence race-detection.
- Produces a lean sync-CFG with more precise data-flow facts.

## Our “Weak” Translation

```
main:
1. x := y := t := 0;
2. create(t1);
3. create(t2);

t1:                  t2:
4. x := x + 1;       8. disableint;
5. disableint;       9. t := x;
6. x := y;          10. enableint;
7. enableint;
```

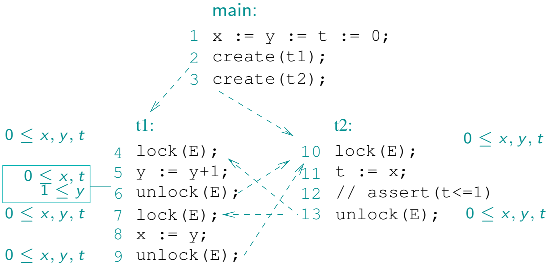
Program $P$

```
main:
1. x := y := t := 0;
2. spawn(t1);
3. spawn(t2);

t1:                  t2:
4. x := x + 1;       8. lock(A);
5. lock(A);          9. t := x;
6. x := y;          10. unlock(A);
7. unlock(A);
```

Lightweight translation $P^W$

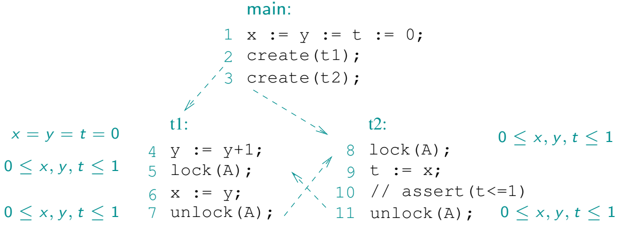
# Sync-CFGs produced by the two translations

Translation
$P^L$



```
main:
1  x := y := t := 0;
2  create(t1);
3  create(t2);

t1:
4  lock(E);
5  y := y+1;
6  unlock(E);
7  lock(E);
8  x := y;
9  unlock(E);

t2:
10 lock(E);
11 t := x;
12 // assert(t<=1)
13 unlock(E);
```

Translation
$P^W$



```
main:
1  x := y := t := 0;
2  create(t1);
3  create(t2);

t1:
4  y := y+1;
5  lock(A);
6  x := y;
7  unlock(A);

t2:
8  lock(A);
9  t := x;
10 // assert(t<=1)
11 unlock(A);
```

## Conclusion and Future Directions

- Sync-CFG based analysis of race-free programs.
- Lays foundation for extending to other non-standard concurrency.
- Future directions:
  - Implement other analyses (Null dereference, points-to, shape analysis).
  - Explore Sync-CFG as a proof technique for concurrent programs.