# Heap Analysis for Concurrent Programs

## BTP-2 Presentation

Anshul Purohit
Roll No: 110050002

Indian Institute of Technology Bombay

Guide: Prof. Uday Khedker

# Outline

- Introduction to Problem Statement
- Heap Analysis
- Analysis technique of Concurrent Programs
- Problems with the technique
- Improvements

# Introduction

- Designing an technique for carrying out heap reference analysis of concurrent programs.
- Reference expression like *x.lptr.rptr.data* are primarily used to access the objects in the heap.
- Will primarily focus on determining the liveness of access links to objects on the heap.
- Java model: The root variables, which are stored on stack, represent references to memory in heap.

# Introduction

Model of threads used to refer to concurrent programs in the problem.

- For accessing shared data, critical sections need to be guarded by the lock and unlock statements.
- Also, we would work under the assumption that program would be data-race free.
- Will present a technique to perform analysis for concurrent programs.

# Heap Analysis

Analyzing properties of heap data is not very trivial.

- The structure of stack and static data is simple to understand since stack variables have a compile-time name(alias) associated with it.
- However, heap data has no compile time alias associated. Also the mapping of access expressions to memory location can change during program execution.
- Objects are referred based on their allocation site.

# Heap Analysis

Heap analysis tries to find out the answer to the questions:

- Can an access expression $a_1$ at program point $p_1$ have the same l-value as access expression $a_2$ at program point $p_2$.
- Can there exist objects in the heap that will not be reachable from the access expressions in the program?
- Which of the access links will be live at a particular point?

We will focus on the liveness analysis part of the heap reference analysis.

# Heap Reference Analysis

A reference can be represented by an access path. In order to perform liveness analysis of heap and identify the set of live links, naming of links is necessary.

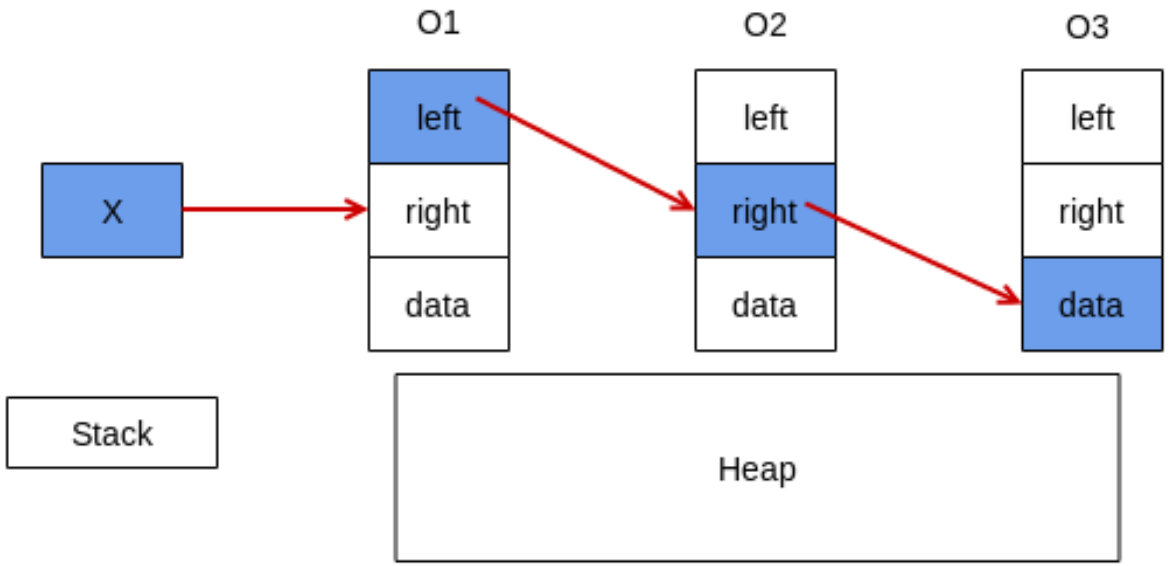



Figure: access path for the expression *x.left.right.data*

# Heap Liveness Analysis

An access path can be unbounded in the case of loops. We need to set a bound on the representation of access paths for liveness information. This is achieved using access graphs. Summarization would also require including program points.

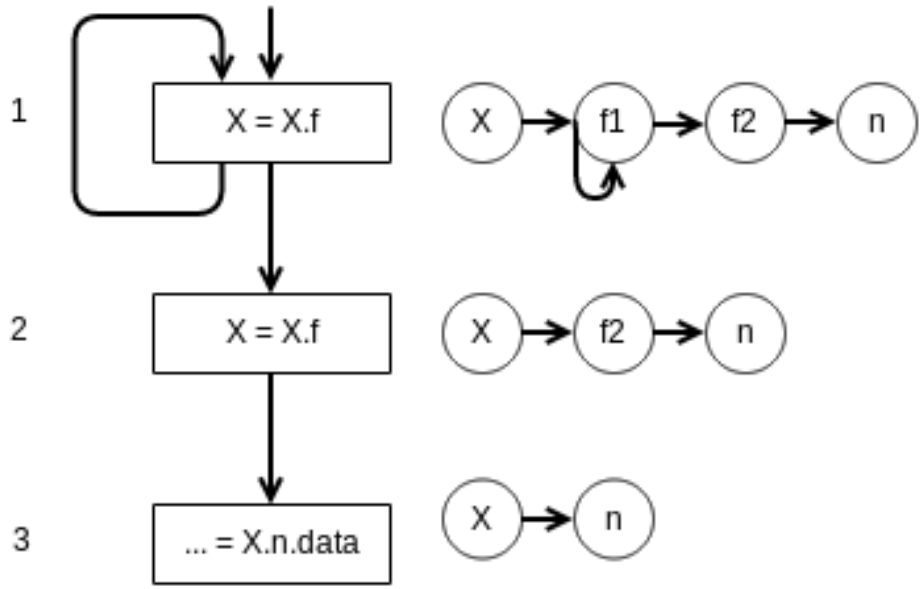



Figure: Use of access graph and liveness data flow values

# Analysis for concurrent programs

- Using the technique mentioned in the paper Dataflow Analysis for Datarace-Free Programs.
- Produces an analysis for concurrent programs, given a sequential data-flow analysis
- Criteria to apply this : The program should be free of data races. Data flow facts should be dependent on the contents of the memory access path.

# Analysis for concurrent programs

Main challenge $\rightarrow$ converting the analysis for sequential programs to concurrent programs. How to propagate data-flow values to handle all possible thread execution orders?

Synchronization structure of the program is made use of to propagate data-flow values

The insight is that data-flow values are only propagated between threads at the lock and unlock points in threads. The relevant statements would usually be present inside the critical section.

# Memory Model

- Specifies the interactions of threads with memory and its shared use.
- Constraints on data access
- Conditions of how data written by one thread is accessible to other threads

Happens-Before Order: Statement a happens before statement b if one of the following hold

- a appears before b in the program order
- b synchronizes-with a
- b can be reached transitively using happens-before relation from a.

# Memory Model

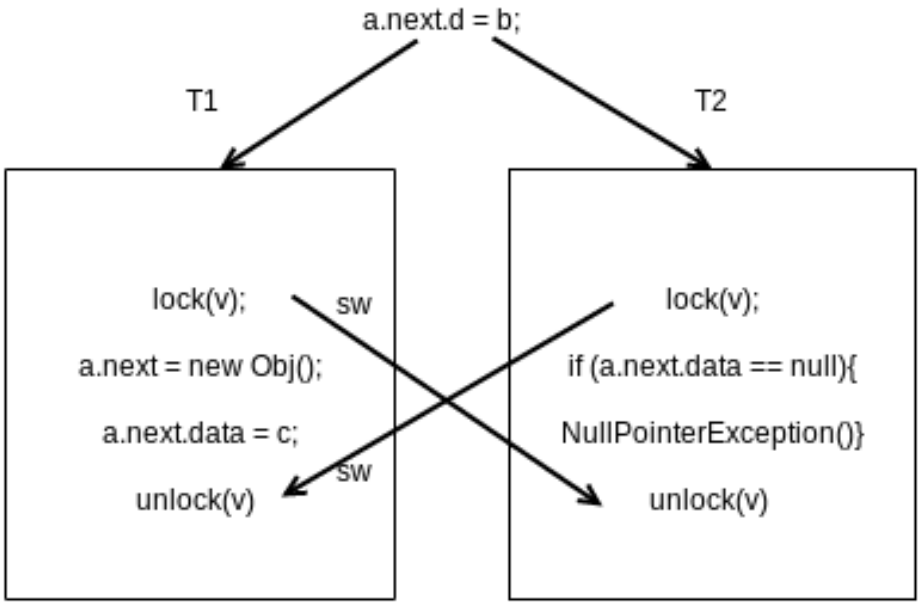



Figure: Happens Before memory model with thread synchronization

The *NullPointerException* in $T_2$ cannot be raised because of the synchronizes-with relation between the lock and unlock statement.

# Memory Model

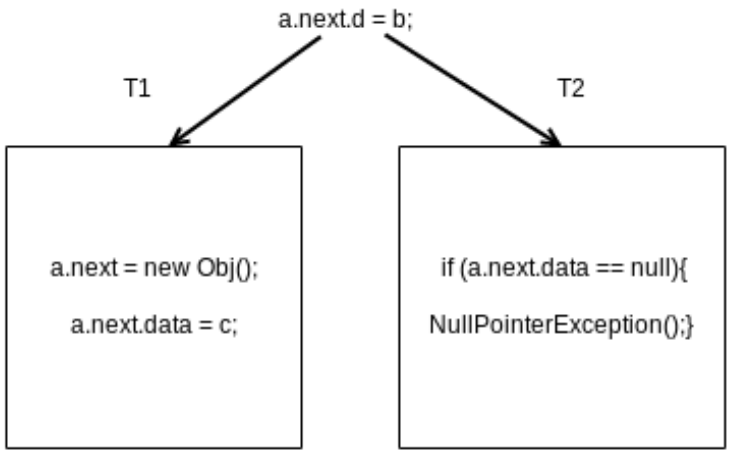



Figure: Happens Before memory model without thread synchronization

There is no synchronization relation between any statement across $T_1$ and $T_2$. There is no happens before order defined for statements across $T_1$ and $T_2$. So, NullPointerException() can be raised.

# Concurrent Null-Pointer Analysis

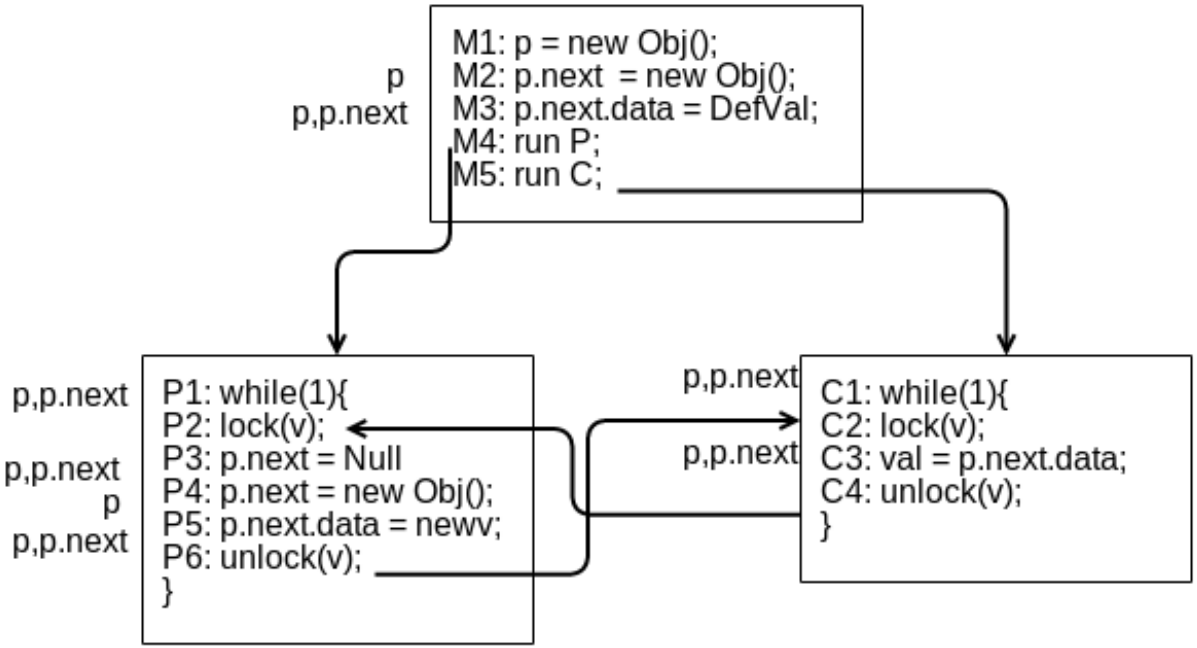



Figure: Heap Access path based null pointer analysis

- Construction of sync-cfg by adding synchronization edges.
- Approximation of concurrent analysis to sequential analysis. Imprecise data flow values are obtained only at irrelevant statements.

# Concurrent Null-Pointer Analysis

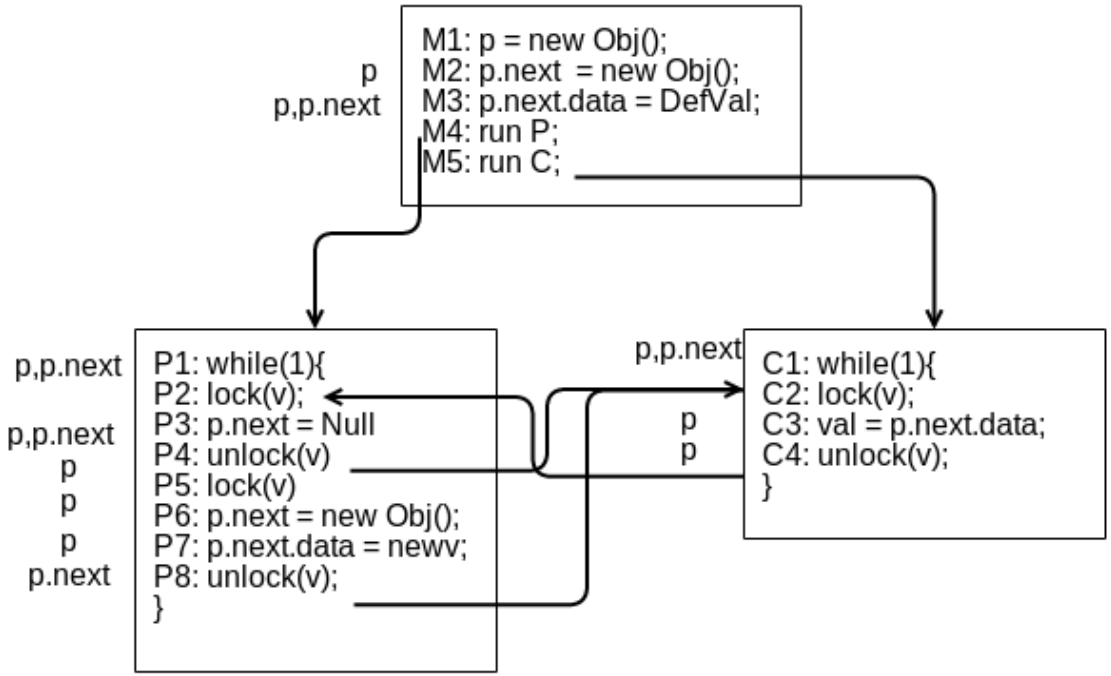



Figure: Heap Access path based null pointer analysis

Only $p$ is not-null at the statement C3, because of merging of values from P4 and P8.

# Concurrent Heap Liveness Analysis

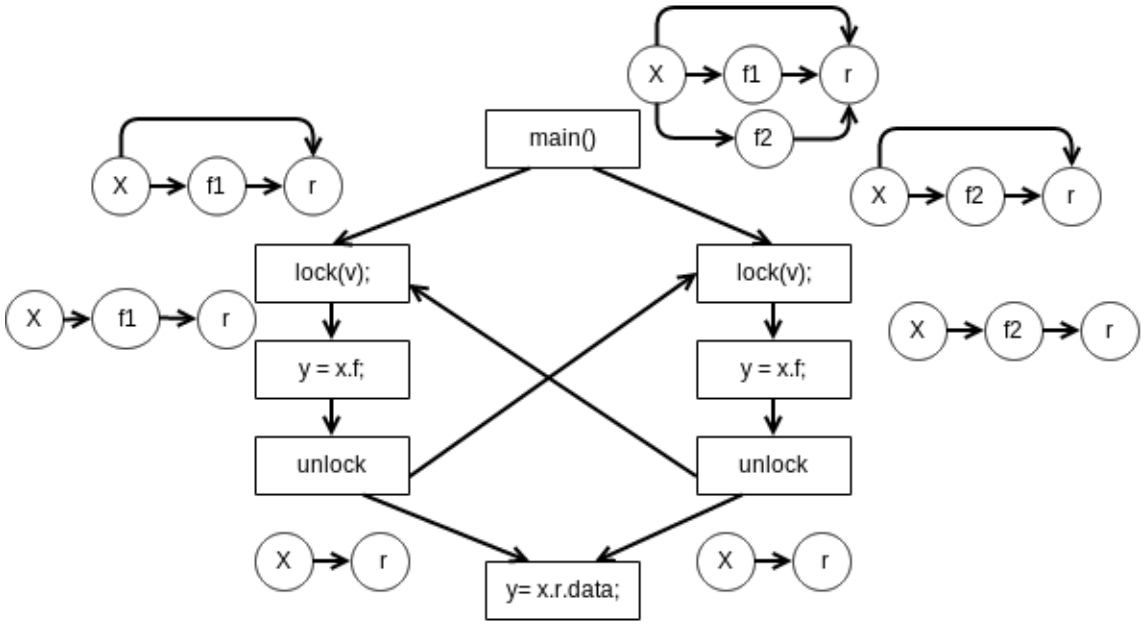



Figure: Concurrent Heap Liveness Analysis

The analysis is guaranteed to return correct data flow values at relevant program statements.

# Sync-cfg for Concurrent Heap Liveness

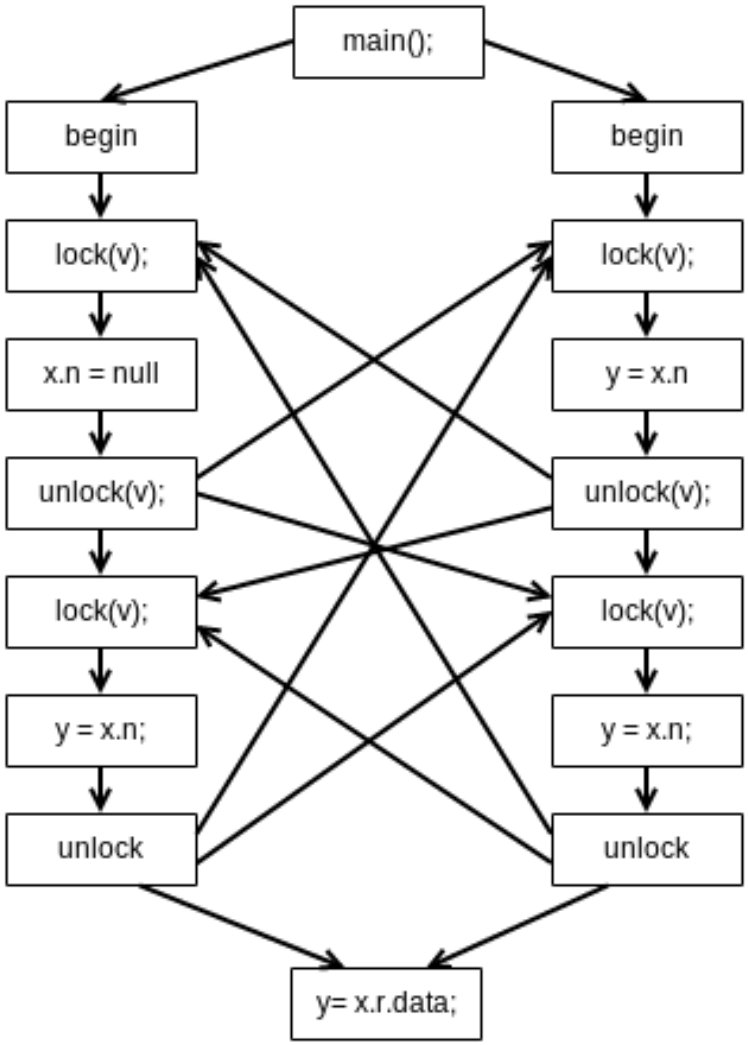



Figure: Concurrent Heap liveness analysis input

# Example for concurrent heap liveness analysis

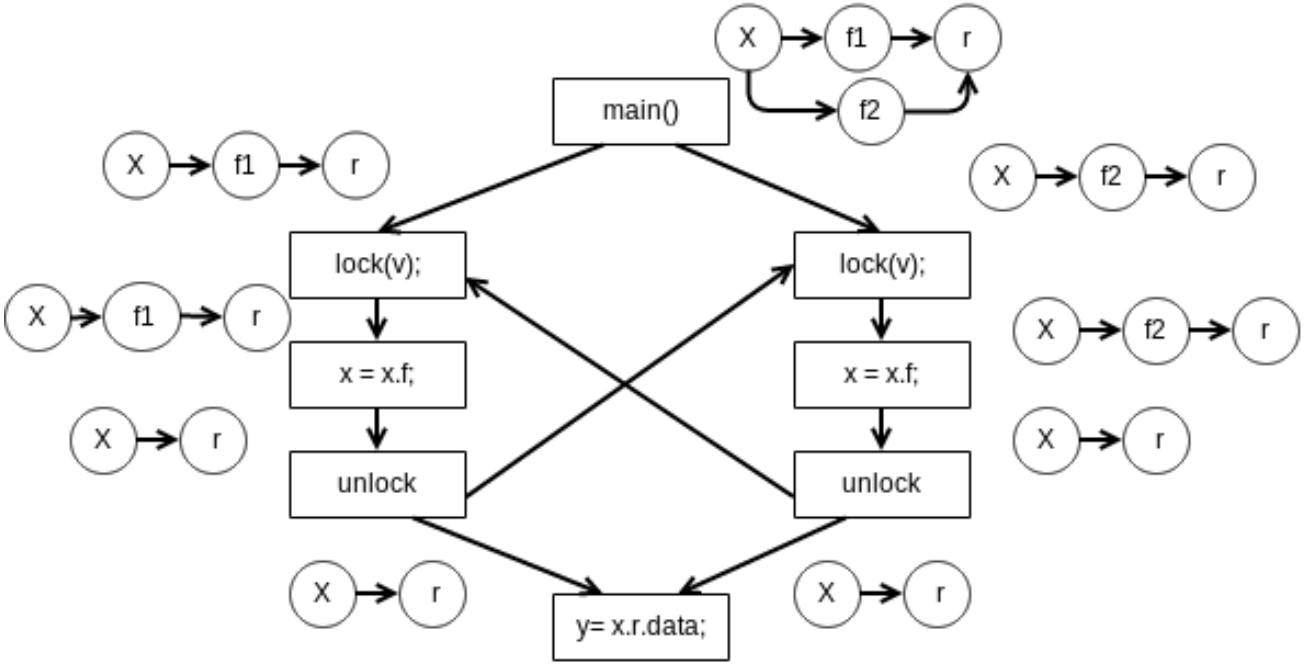



Figure: Concurrent Heap liveness analysis iteration 1

# Example for concurrent heap liveness analysis

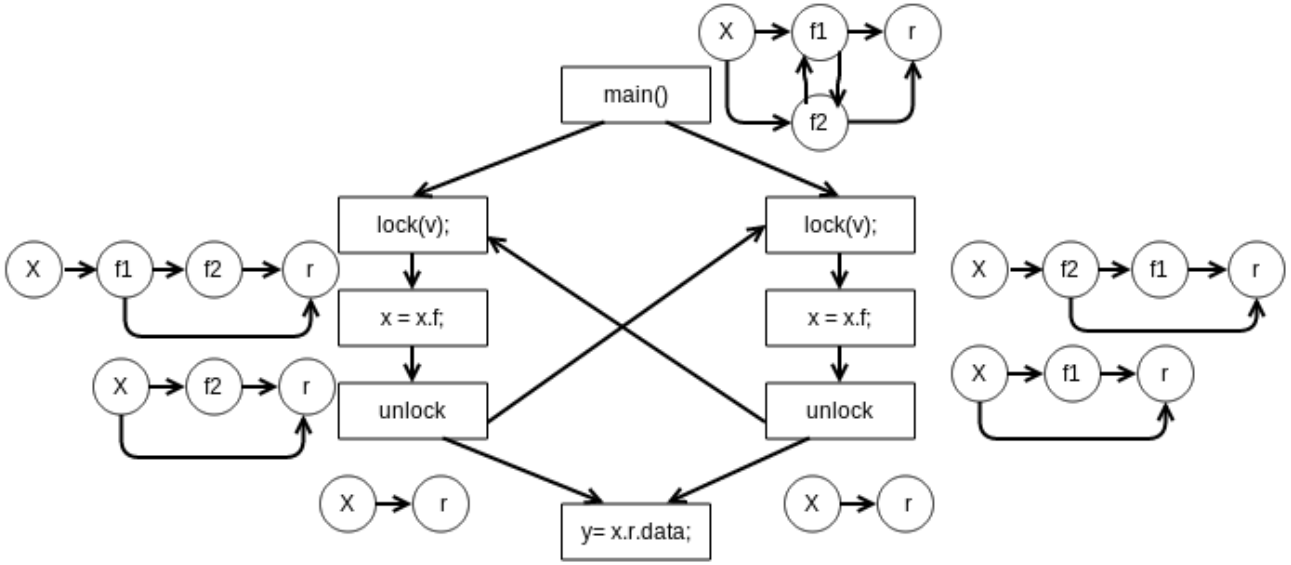



Figure: Concurrent Heap liveness analysis iteration 2

# Example for concurrent heap liveness analysis

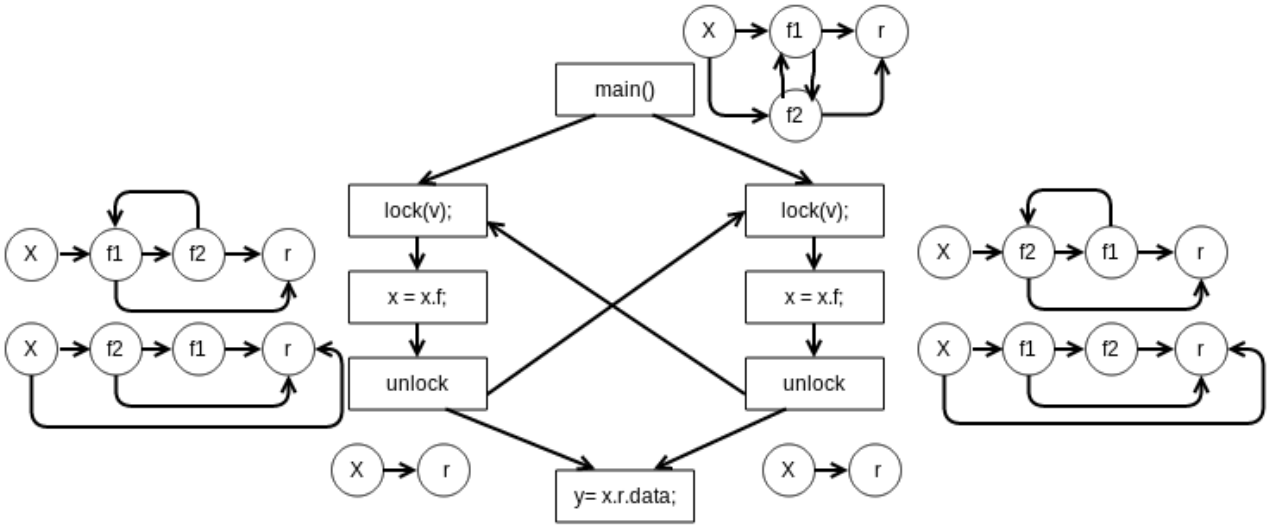



Figure: Concurrent Heap liveness analysis iteration 3

# Example for concurrent heap liveness analysis

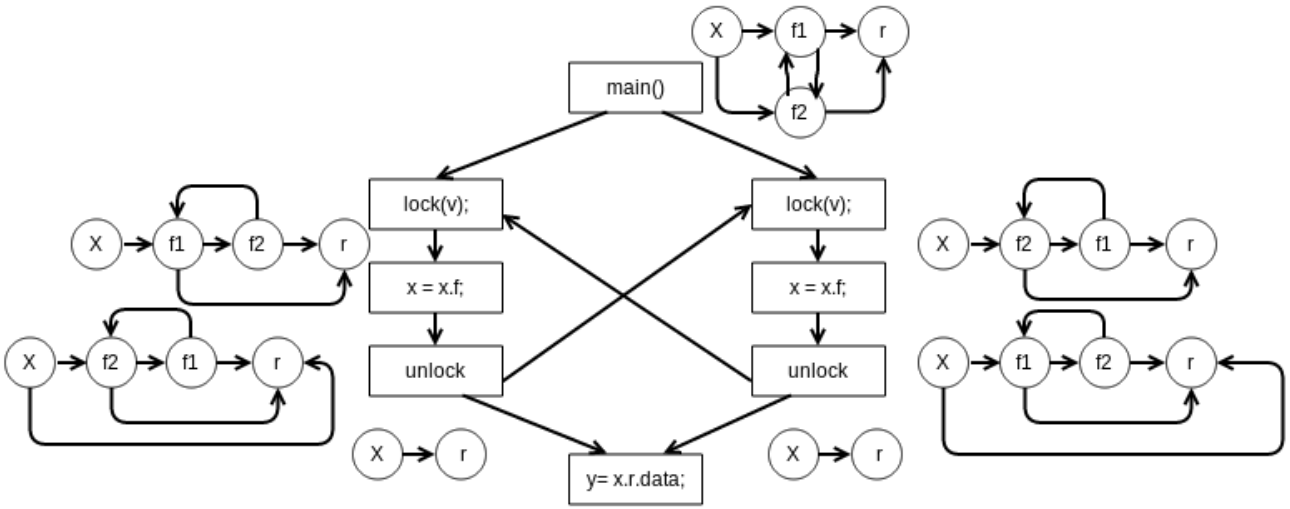



Figure: Concurrent Heap liveness analysis iteration 4

# Example for concurrent heap liveness analysis

Example of the same program without synchronization edges.

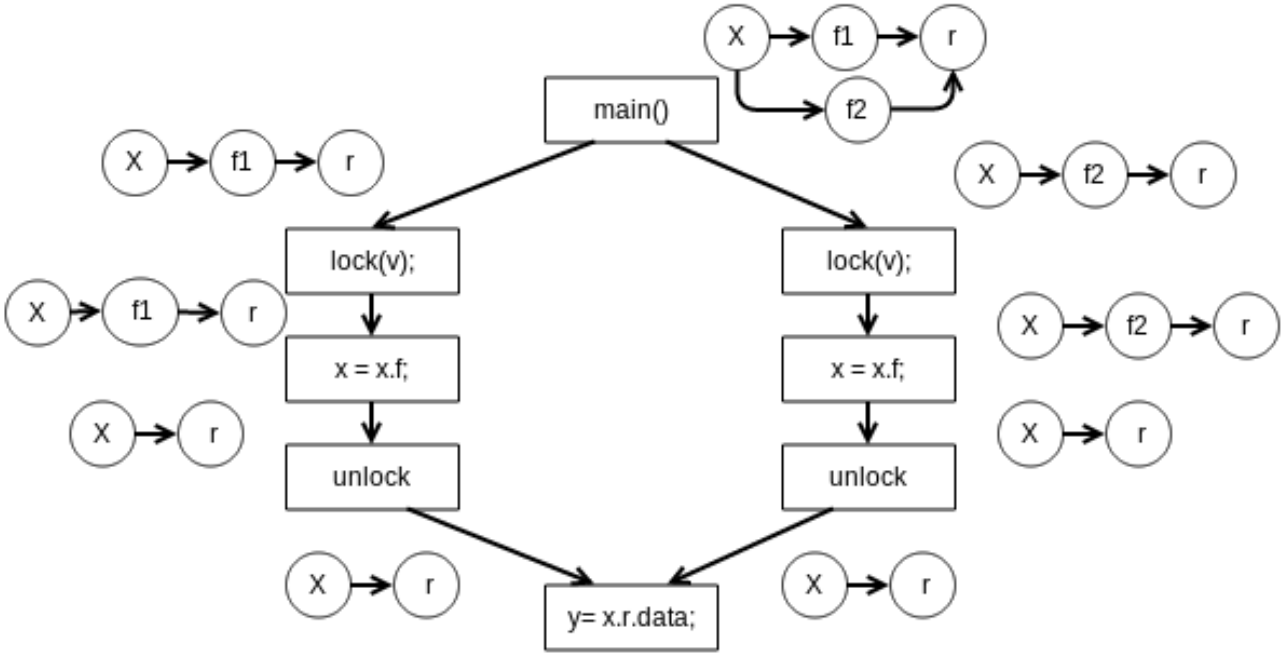



Figure: Concurrent Heap liveness analysis without synchronization edges

An analysis without adding synchronization edges provides better results in this example. This highlights that there are certain problems with this analysis technique

# Problems with the analysis technique

- Propagation of data flow values across inter-thread edges is treated same as along intra-thread edges.
- At program point containing the main statement, the possible live links can be be $x.f1.f2.r$ or $x.f2.f1.r$.
- Final data flow value obtained after analysis includes imprecise access links $x.f1.r$, $x.f2.r$ , $x.\{f2.f1\}^{+}.r$, $x.\{f1.f2\}^{+}.r$

# Problems with the analysis technique

- No bound on the number of transitions from node $f_1$ to $f_2$ and from $f_2$ to $f_1$ in the access graph.
- Thread synchronization edges introduce loops in the program graph
- Cause of imprecise data flow values is due to taking into account execution of critical sections more than once

# Problems with the analysis technique

Execution of multi-threaded programs

- Interleaving of statements of the multiple threads
- Addition of synchronization edges leads to formation of loop in the control flow graph
- Data flow value is transferred to every critical section multiple times
- Thus there is a need to identify the critical sections that only execute once

# Analysis of Critical Section Execution

Intra-thread Analysis to identify if critical section is executed once.

- If there is no loop within and across a critical section, it can only be executed once.
- If a loop is present within a critical section, even then the critical section can be executed only once.
- It a loop is present across a critical section in a thread, then the critical section can be executed zero or more number of times.

# Adding Thread Info in Access Graphs

Once we know about the number of executions of each critical section there is a need to:

- Need to figure out how to store the thread switchings in an access graph
- Store thread id corresponding to every edge in the access graph
- Identify which paths are possible with respect to the execution semantics.
- For example paths with multiple thread switches into a critical section can't be allowed.

# Examples 1

Imposing the restrictions that concurrent sections can only be executed once for $T_1$ and $T_2$. Can discard access paths containing more than 1 edges labeled 1 or 2.

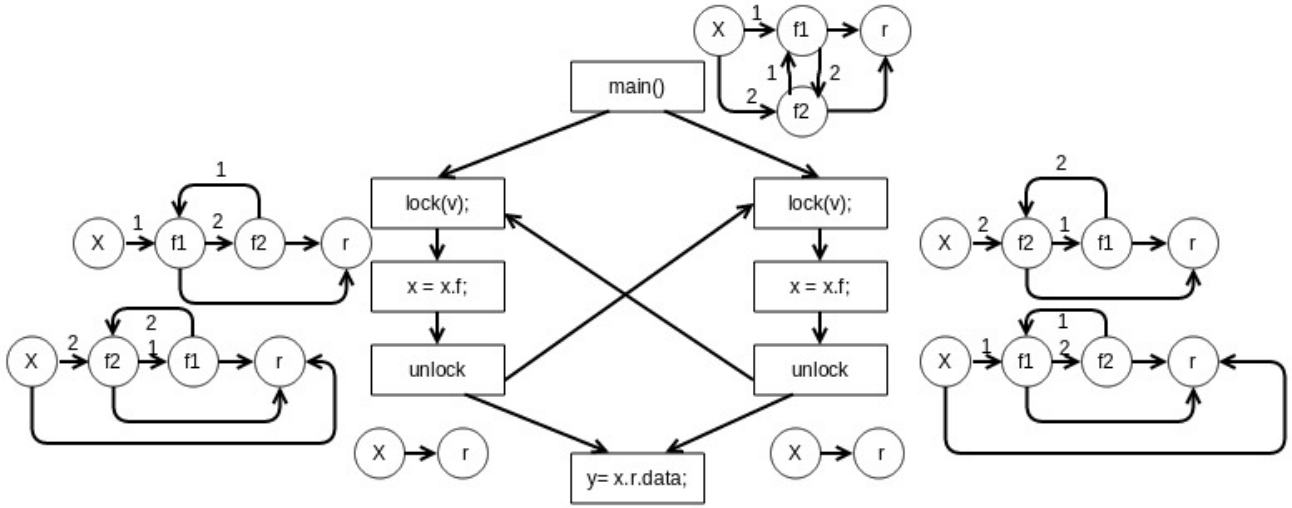



Figure: Concurrent Heap liveness analysis example. Note that edges are marked by thread id

# Examples 2

Critical section of thread 1 and thread 2 can only be executed once. Can allow 0 or more occurrence of edges labeled 2 but the number of edges labeled 1 must strictly be 1 in a access graph path.

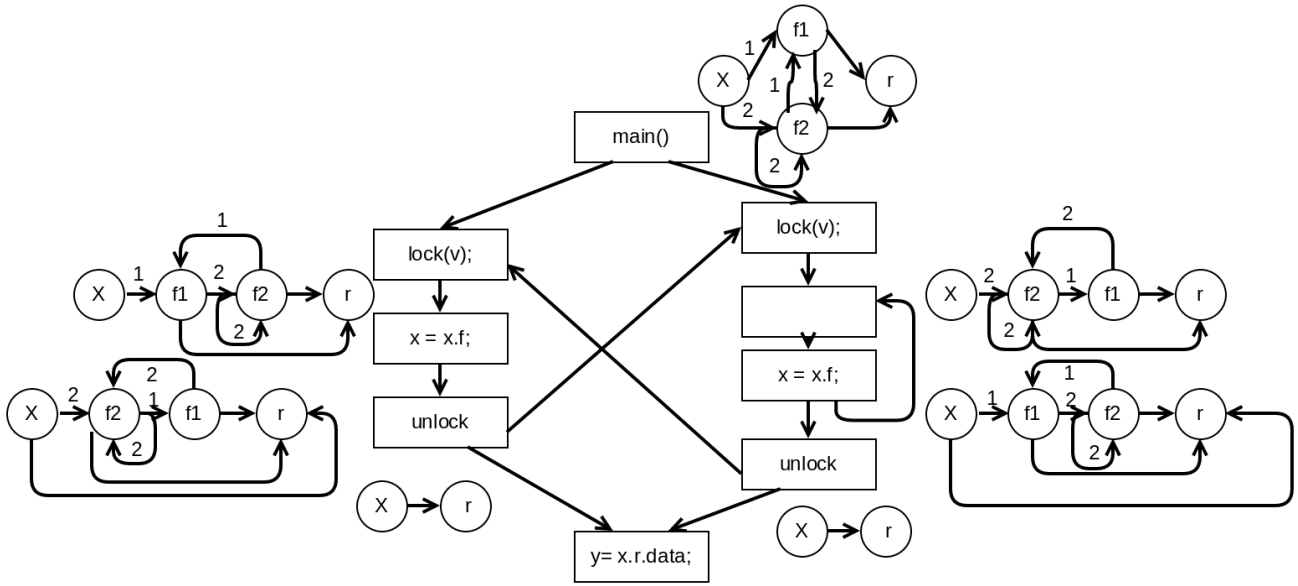



Figure: Concurrent Heap liveness analysis example (on sync cfg). Note that edges in access graphs are marked by thread id

# Example 3

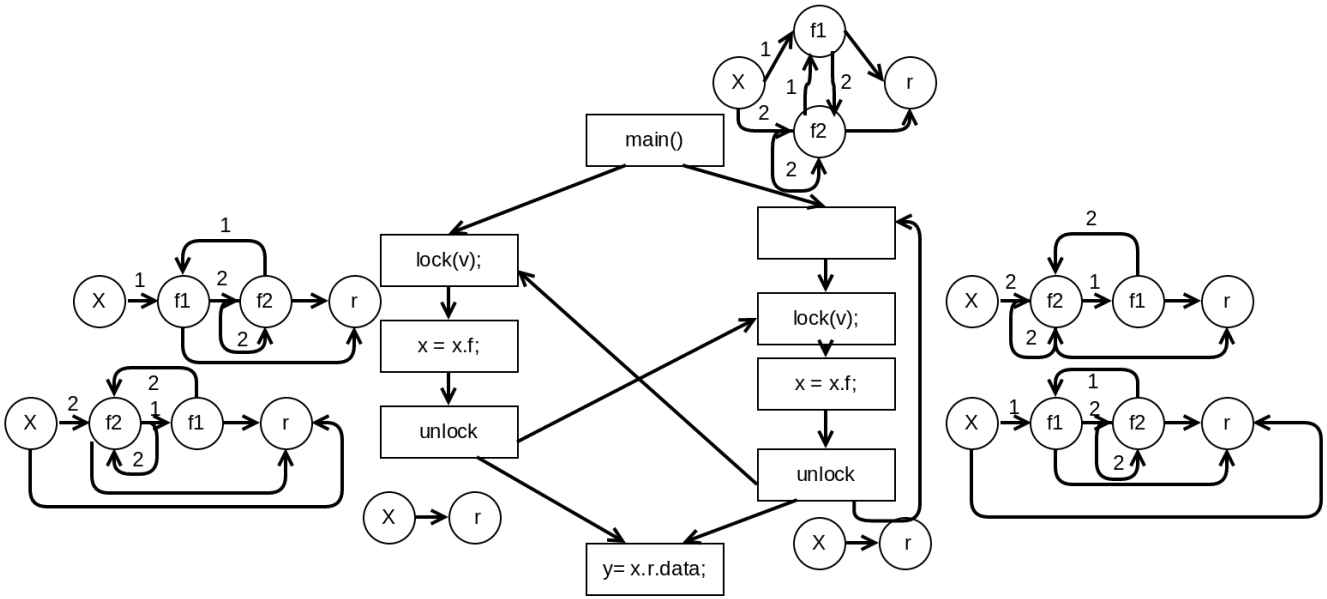



Figure: Concurrent Heap liveness analysis example (on sync-cfg).

Access graphs returned are the same for both Example 2 and Example 3. There is a need to ensure that the edge from $f_1$ to $f_2$ in the access graph can be traversed any number of times.

# Changes in the analysis

For ensuring inter-thread edges are treated according to the critical section execution semantics, impose conditions on them. Label the edges with the number of times the transition is possible.
Considering the examples, these are the desired access graphs

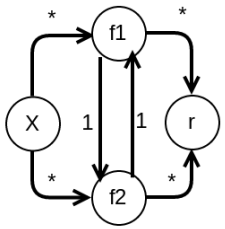



Figure 5.2

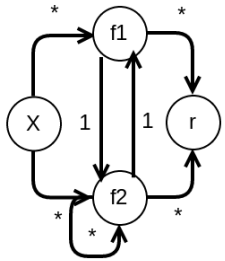



Figure 5.3

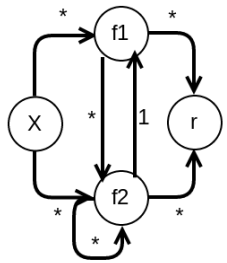



Figure 5.4

Figure: Desired Access graphs for the three examples

# Concurrent Heap Access Examples

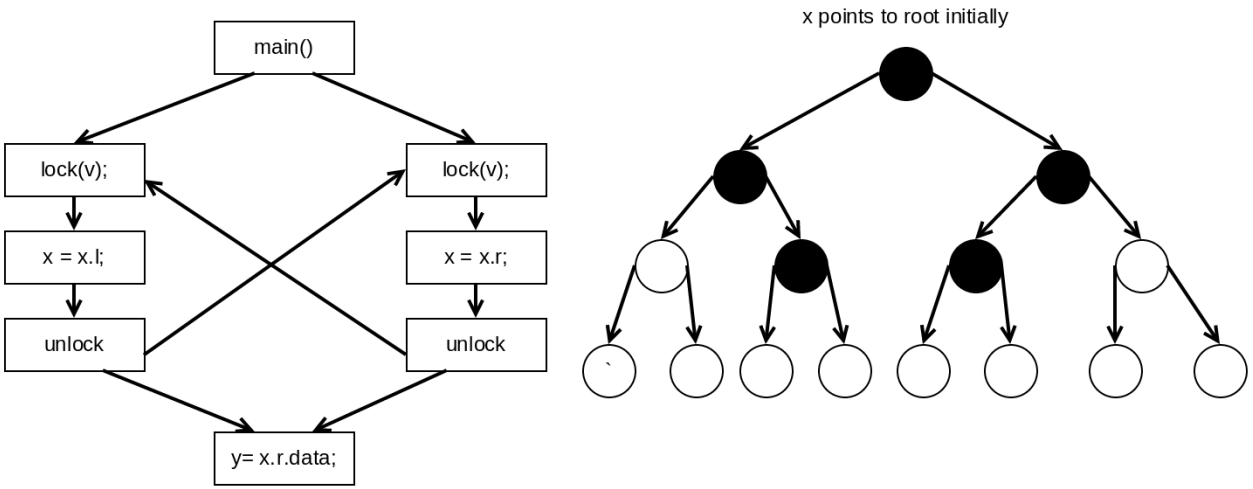



Figure: Concurrent access example in a tree

If $C_1$ is executed first then the object $x.l.r$ is accessed otherwise $x.r.l$ is accessed. Corresponds to the first example

# Concurrent Heap Access Examples

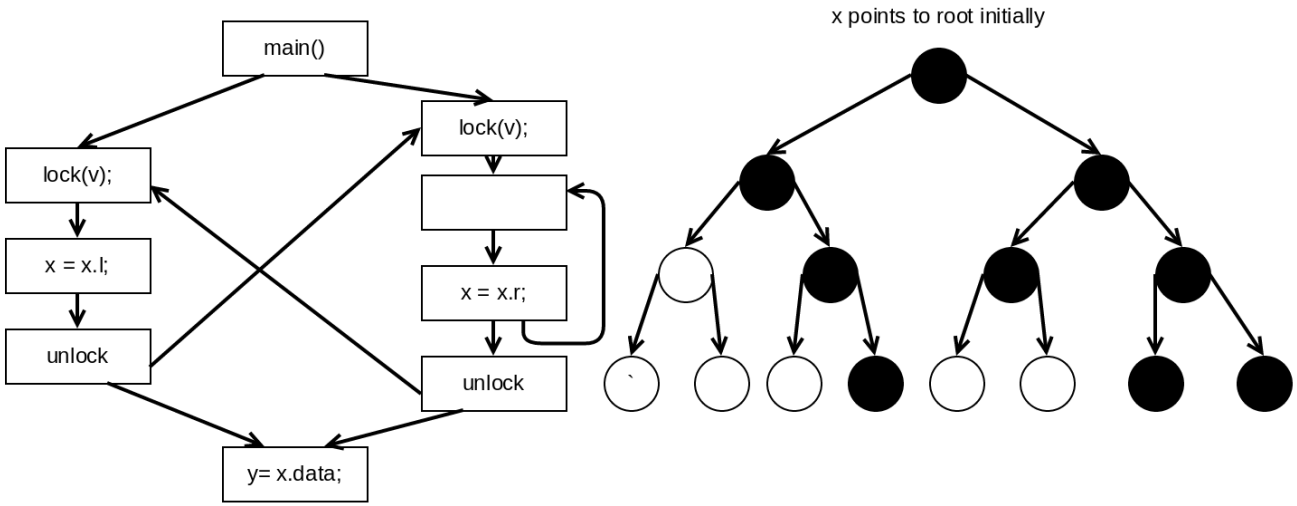



Figure: Concurrent access example in a tree

Objects accessed will satisfy the regular expression $x.l.r^*$ or $x.r^*.l$ depending on the starting critical section.

# Concurrent Heap Access Examples

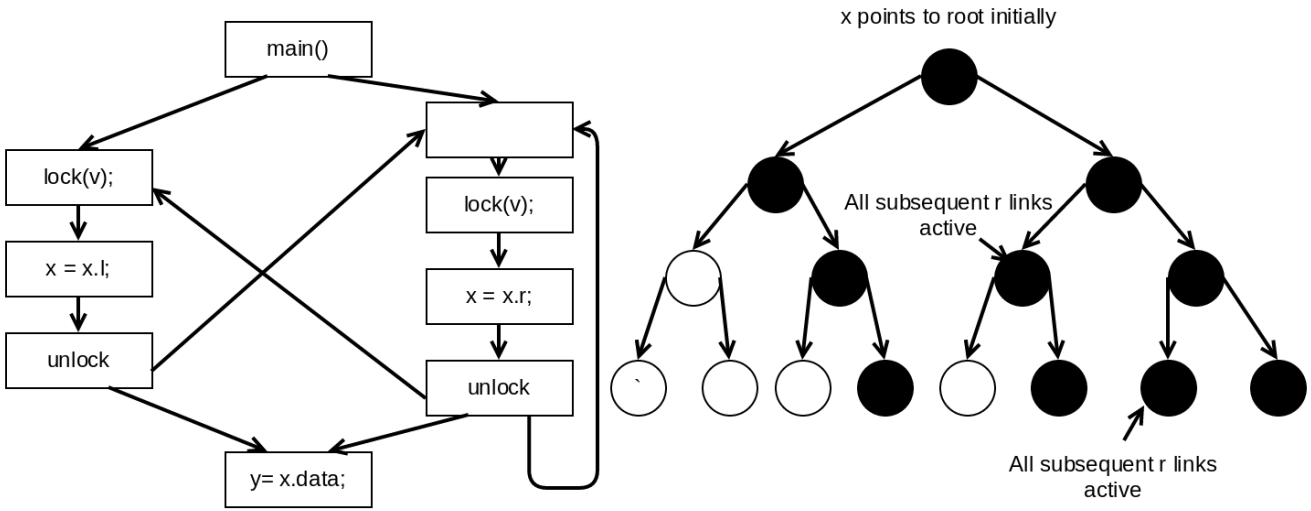



Figure: Another Concurrent access example in a tree

$C_2$ can be executed any number of times. The difference from the last example being that $C_2{}^*$ can be executed after $(C_2, C_1)$.

# Concurrent Heap Access Examples

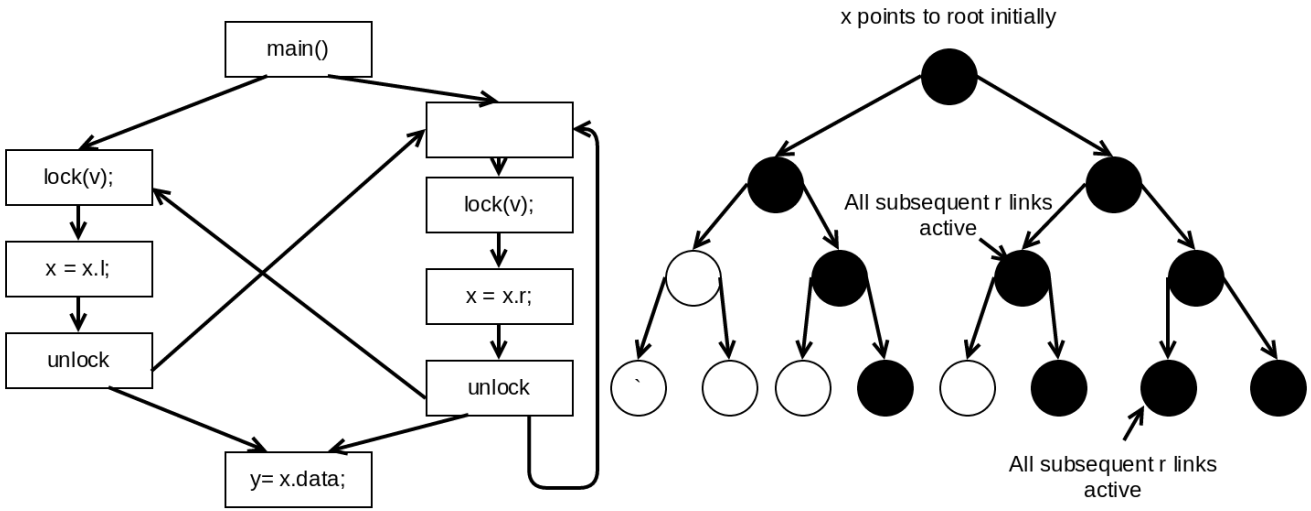



Figure: Another Concurrent access example in a tree

$C_2$ can be executed any number of times. The difference from the last example being that $C_2{}^*$ can be executed after $(C_2, C_1)$.

# Concurrent Heap Access Examples

An example when any number of $C_1$ and $C_2$ executions are possible and there are no restrictions on thread-switching. Possible executions are $(C_1?, C_2?)^+$.

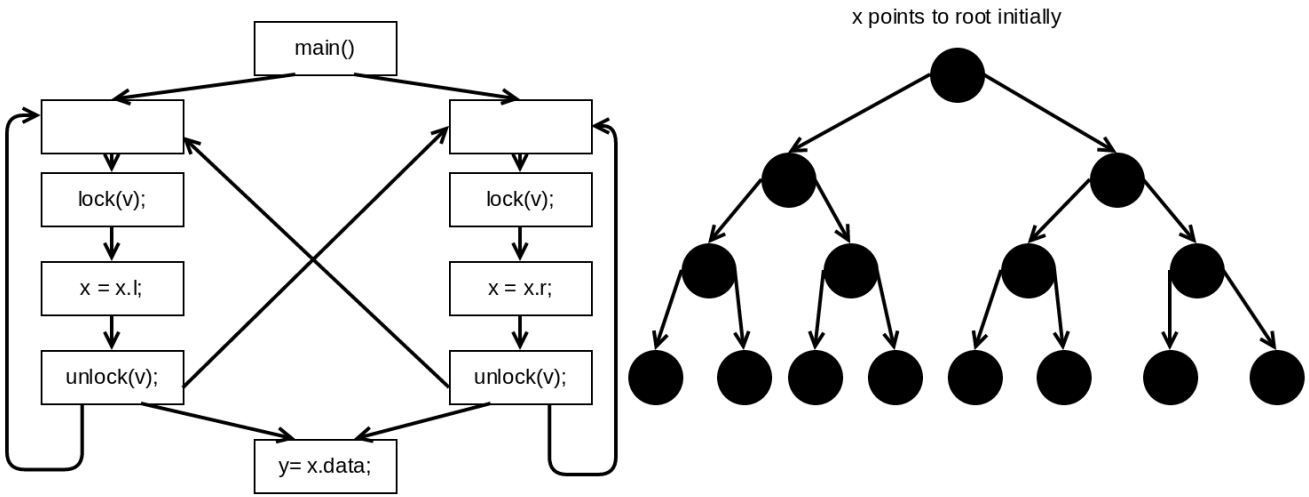
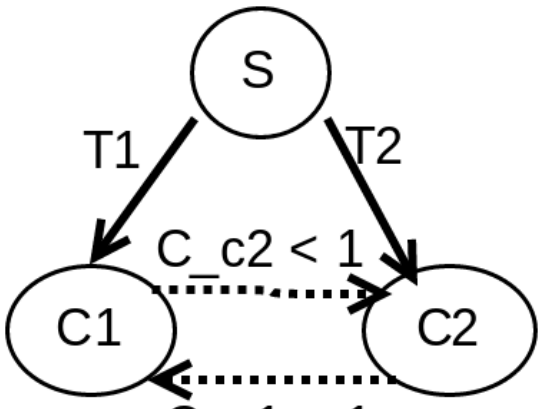



Figure: Another Concurrent access example in a tree

# Construction a transition system

Once we have identified the critical sections and their number of executions possible, we can create a transition system with nodes being the critical section and edges being transitions from a critical section to another based on condition of execution semantics.

This condition can be on count of nodes/critcal section execution and the program order conditions(intra-thread order of statements). In case of 2 critical sections in the same thread, the condition to the incoming edges to the second one must be the predicate $C_1 >= 1$ && $C_2 < 1$



# Conversion of transition system to automata

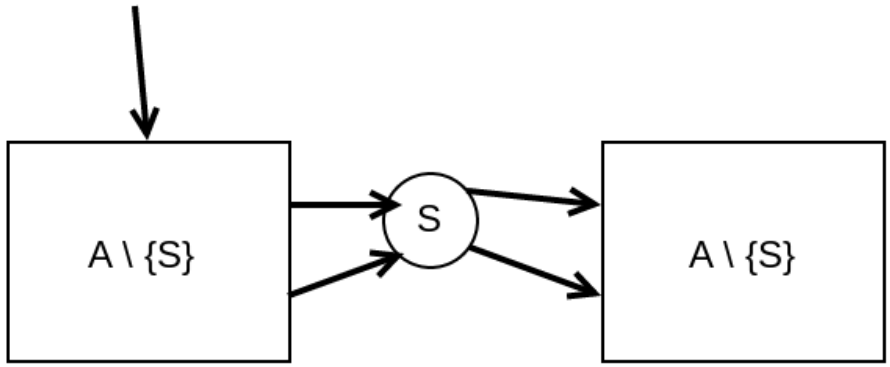



Figure: Ensuring S gets visited only once for a valid path

We need to come up with a way such to ensure that states representing critical sections can only be reached once. All the unreachable states in the two copies are deleted. Also the conditions to the duplicated states in the right copy gets conjuncted by the count of that states $<= 1$.

# Conversion of transition system to automata

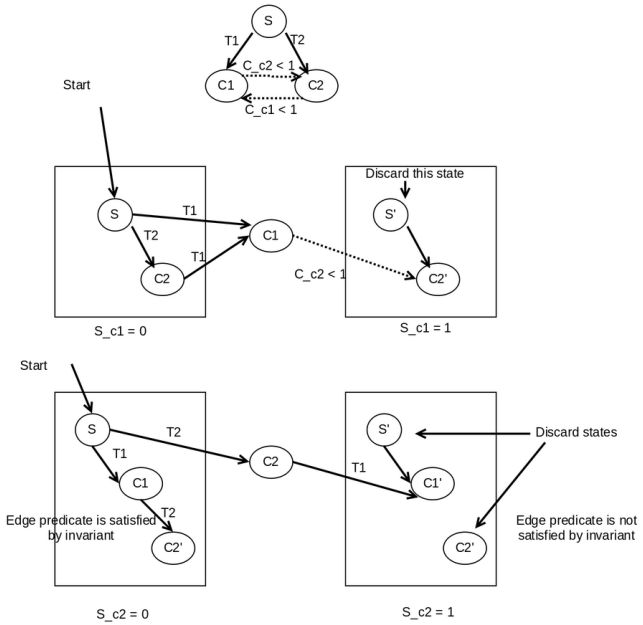



Figure: Converting the condtion based transition system to automata like structure.

The executions possible are $C_2, C_1$ and $C_1 C_2$.

# Conclusions

- Problems with the analysis on the sync-cfg/program graph and possible reasons.
- Some improvements to the basic sync-cfg concurrent analysis suggested. Distinguish between inter and intra-thread edges. Handle execution semantics of critical sections and thread switchings.
- The notion of thread context(execution orders) is introduced in a vague way. Analysis should only be performed over the thread switches which can actually be feasible.
- The automata representation of execution orders can be used to form an exploded-thread-cfg graph and we can directly perform analysis over the statements in the execution order.

# Future work

- Finding if this method can be applied to other concurrent analysis.
- Implementation of the transition diagram to automata converter routine. Generation and implementation of exploded-thread-cfg from the automata structure of critical section execution.
- Figuring out if the analysis based on thread context/execution semantics extended to handle function calls in threads. (Extension to Inter-procedural)

# Thank You