Inter-Procedural Heap Reference Analysis for Concurrent Programs

BTP Presentation

Anshul Purohit
Roll No: 110050002

Guide: Prof. Uday Khedkar

Outline

- Introduction to Problem Statement
- Heap Analysis
- Inter-Procedural Analysis
- Analysis of Concurrent Programs
- Design of analysis and tools

Introduction - Flow Sensitivity

Data flow analysis is a technique for gathering program information at various points in a program. The value is then propagated in the Control Flow graph (CFG) of the program. It can be classified in two types:

- Flow-insensitive analysis ignores the control-flow graph, and assumes that statements can execute in any order.
- Flow-sensitive analysis takes control flow structure of program into account.

Introduction - Interprocedural Analysis

- Inter-procedural analysis extends the scope of data flow analysis across procedure boundaries.
- It incorporates the effects of procedure calls in the caller procedures and calling contexts in the callee procedure.
- A context-sensitive analysis is an **inter-procedural analysis** which reanalyzes callee procedure for each context.
- Context insensitive analysis performs analysis independent of calling context.

Introduction - Problem Statement

- Designing a framework for carrying out inter-procedural analysis of concurrent programs and then implement heap reference analysis using the framework.
- Determining the information like liveness, accessibility and points-to information of reference expressions.
- Reference expression like *x.lptr.rptr.data* are primarily used to access the objects in the heap.
- Java model: The root variables, which are stored on stack, represent references to memory in heap.

Introduction - Concurrency

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

- Need to define a run method. Invoking the run method starts concurrent thread execution.
- For accessing shared data, critical sections need to be guarded by the lock and unlock statements.
- High level abstractions of concurrency can be modeled in terms of threads. Thread library handles spawning, joining and synchronization of threads.

Heap Analysis

Analyzing properties of heap data is not very trivial.

- The structure of stack and static data is simple to understand.
- 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?

Points-to analysis for Java

In Java pointers are not created explicitly. All objects in Java are accessed using references. Points-to analysis for Java programs identifies the objects pointed to by references at run time.

```
class A(){}
class B(){
public A f;
public void set(A p)
{
this.f = p;
}
}
class C(){
public B g;
public void set(B q){
this.g = q;
}
}
s1 : A x = new A()
s2 : B y = new B()
s3 : C z = new C()
s4 : y.set(x);
s5 : z.set(y);
s5 : A a = z.g.f;
```

Figure: Example for heap access and points-to

Points-to Graph

Points-to graph in Java contain two types of edges. The first type of edge is to represent the information that reference variable v is pointing to object o. The second type of edge represents the field f of o_1 pointing to o_2.

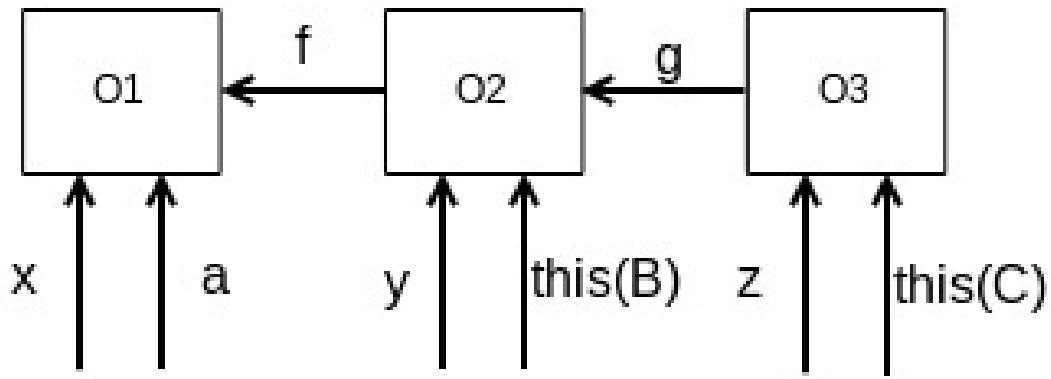


Figure: Example for points-to graph

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.

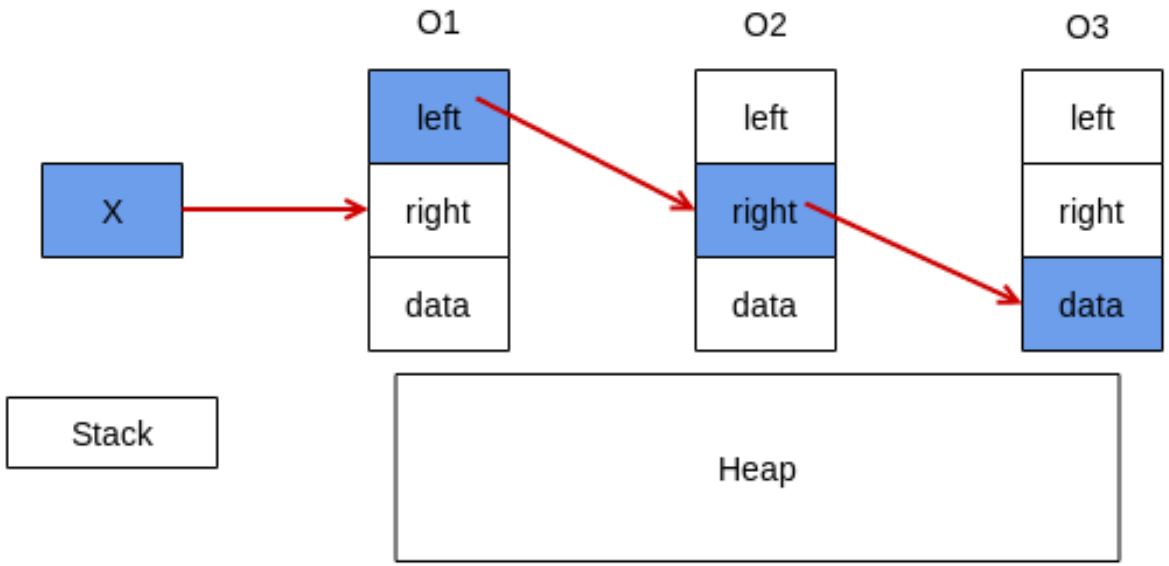


Figure: access path for the expression *x.left.right.data*

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.

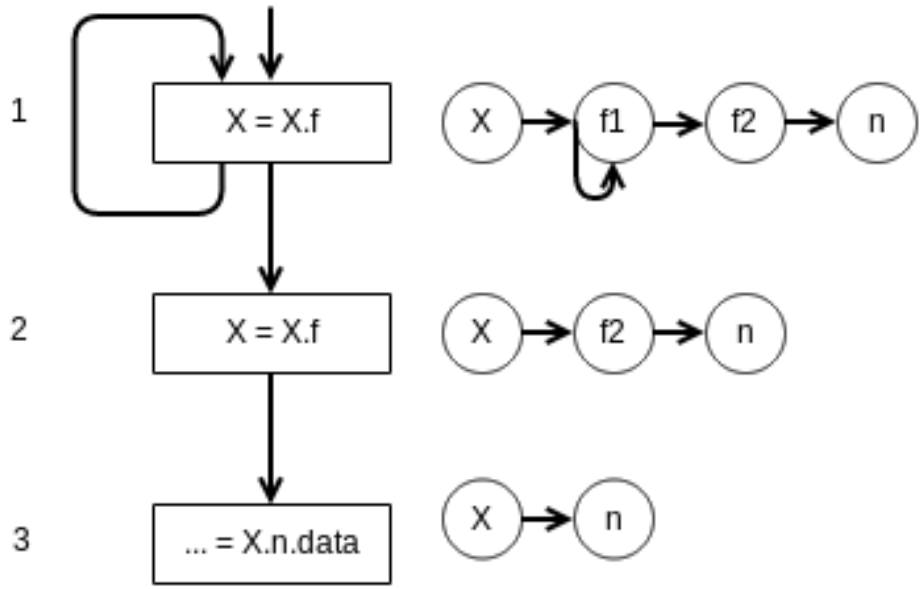


Figure: Use of access graph and liveness data flow values

Availability and Anticipability

- An access path ρ is said to available at a program point p if the target of each prefix of ρ is guaranteed to be created along each path reaching p.
- An access path ρ is said to be anticipable at p if the target of each prefix of ρ will be dereferenced along every path starting from p.
- Access graphs are not needed to carry out available and anticipable analysis over heap data because the sets are bounded. This is due to the nature of the problem (over all paths).

Inter-procedural analysis

Inter-procedural Analysis is required to obtain more precise results as it is very common that programs can have multiple function calls.

- It is essential to consider the effect of function call on the data flow value entering the node.
- Inter-procedural analysis takes into account call return , parameter passing , local variables of the function, return values and recursion into account
- Major issue to be dealt while handling inter-procedural analysis is to deal with calling contexts.

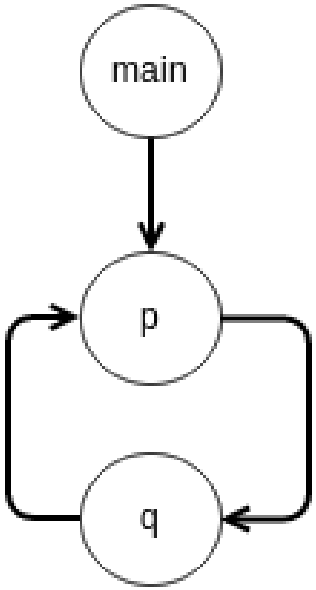
Context Sensitivity

- Context insensitive analysis over-approximates inter-procedural control flow. Invalid return control paths are accessed.
- Context sensitive analysis takes into account the calling context, thus only valid return paths are considered.
- Context sensitivity plays an important role in Java as it is an OO language. Data access is indirect through method calls.

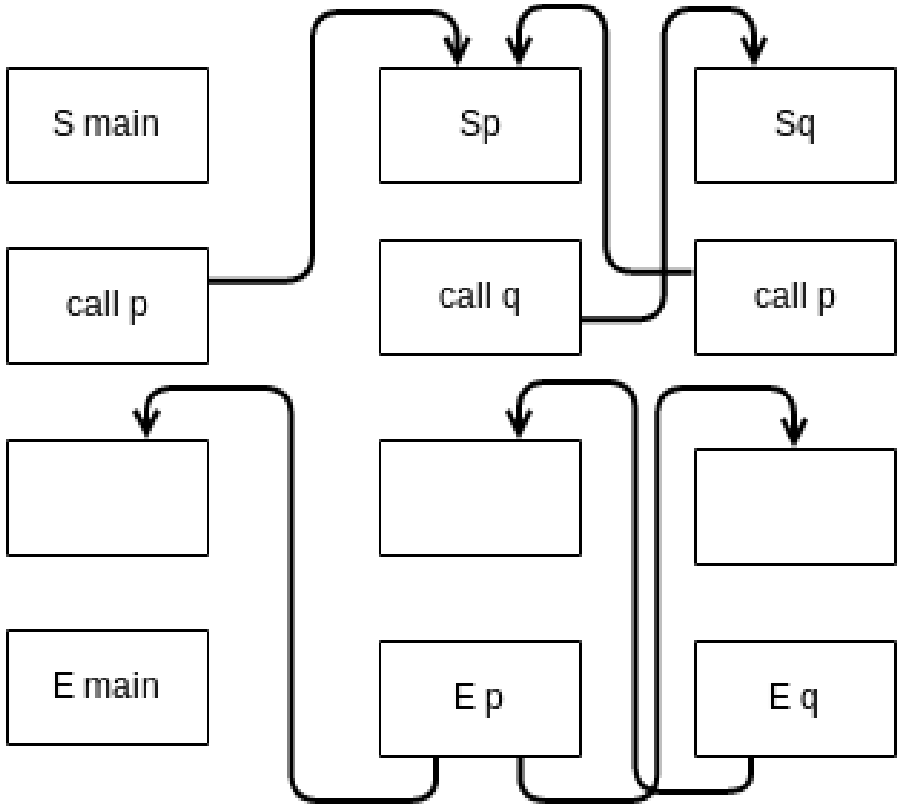
Call Graph

- Static data structure representing run-time calling relationships among procedures.
- Call multi-graph is a directed graph which represents calling relationships.
- In Super graph callsites are connected to the callee procedure entry node and the exit node is connected to return node in the caller.

Example



Call MultiGraph



Super Graph

Figure: Call Multi-graph and super graph examples

Approaches to Inter-procedural Analysis.
Functional

- Summary flow functions are computed for each function.
- Directly apply the summarized flow function for handling call. Summarized flow function is dependent on the callee.
- Iterative computation of summary flow function would be need till fixed point is achieved. Guaranteed to terminate only for finite lattices.

Call string at a program point is the sequence of unfinished calls reaching that point starting from the main procedure call.

The data flow equations are changed to incorporate the merging of the data flow values only if the contexts(call strings) are the same.

At a call node c_i , c_i is appended to the call-string value at that point. Similarly at a return node the last call site ci is removed. For recursive programs, number of call strings can be infinite.

Value termination of Call Strings : Grouping into equivalence classes if they have same data flow value at the start node.

Combination of the two views of contexts: data flow values at call site are stored as value contexts and call strings as calling contexts.

A value context is defined by a particular data flow value reaching a procedure. It is used to enumerate and tabulate the summary flow function of the procedure in terms of input and output data flow values.

When a new call to a procedure is encountered, the value context table is consulted to decide if the procedure needs to be analyzed again.

A calling context transition table is maintained allowing flow of information along inter-procedurally valid paths.

Transitions are recorded in the form ((X,c) , Y) where X represents calling context, c represents call site and Y represents callee context.

Example of value context method

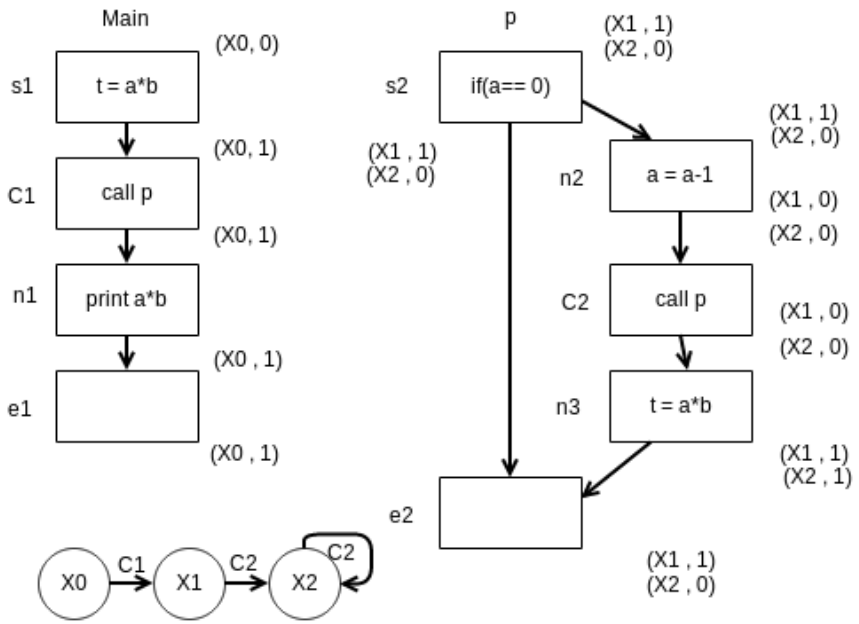


Figure: Example for inter-procedural available analysis

Example of value context method

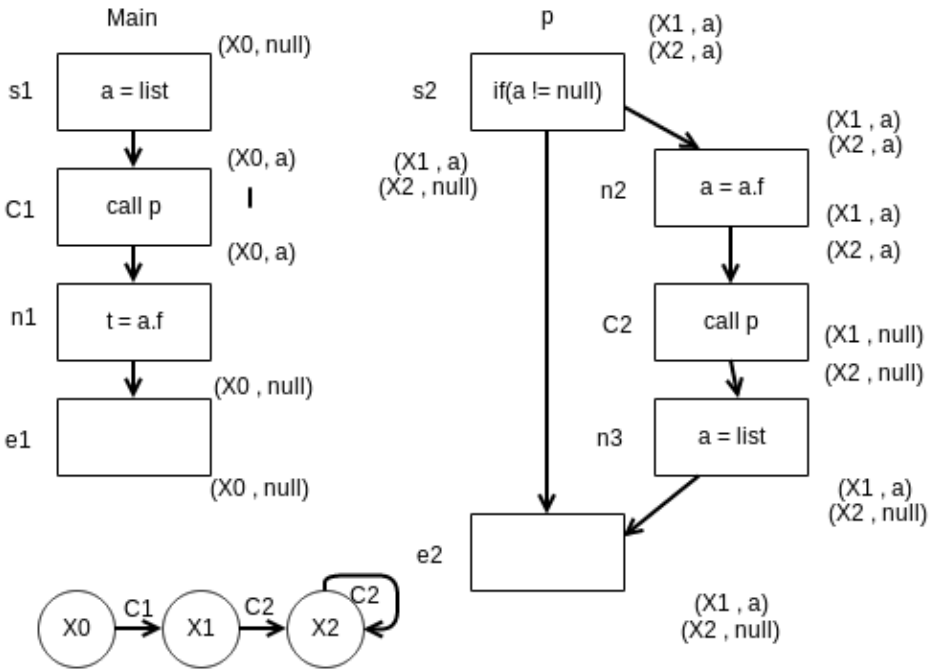


Figure: Example for inter-procedural heap liveness

In the example the 2 contexts X_1 and X_2 are stored for the recursive procedure p.Value based contexts are used as a cache table for distinct call sites apart from terminating analysis of recursive procedures.

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 : Data flow facts should be dependent on the contents of the memory access path. This can be applied to constant propagation, not-null, liveness analysis. The program should be free of data races.

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 order? .

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

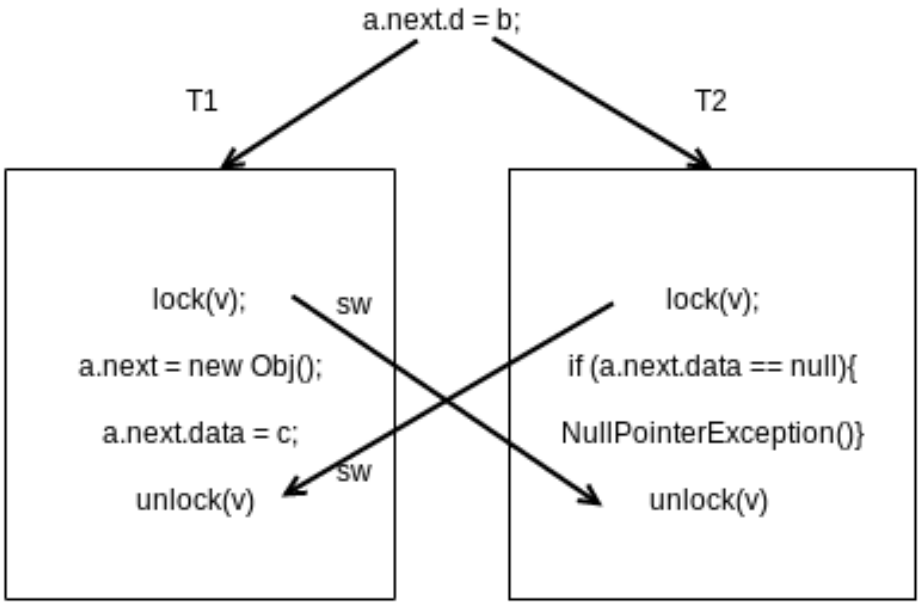


Figure: Happens Before memory model with thread synchronization

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

Memory Model

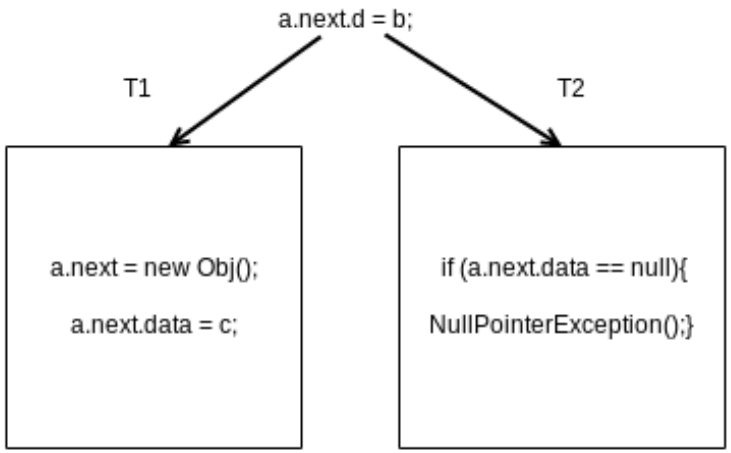


Figure: Happens Before memory model without thread synchronization

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

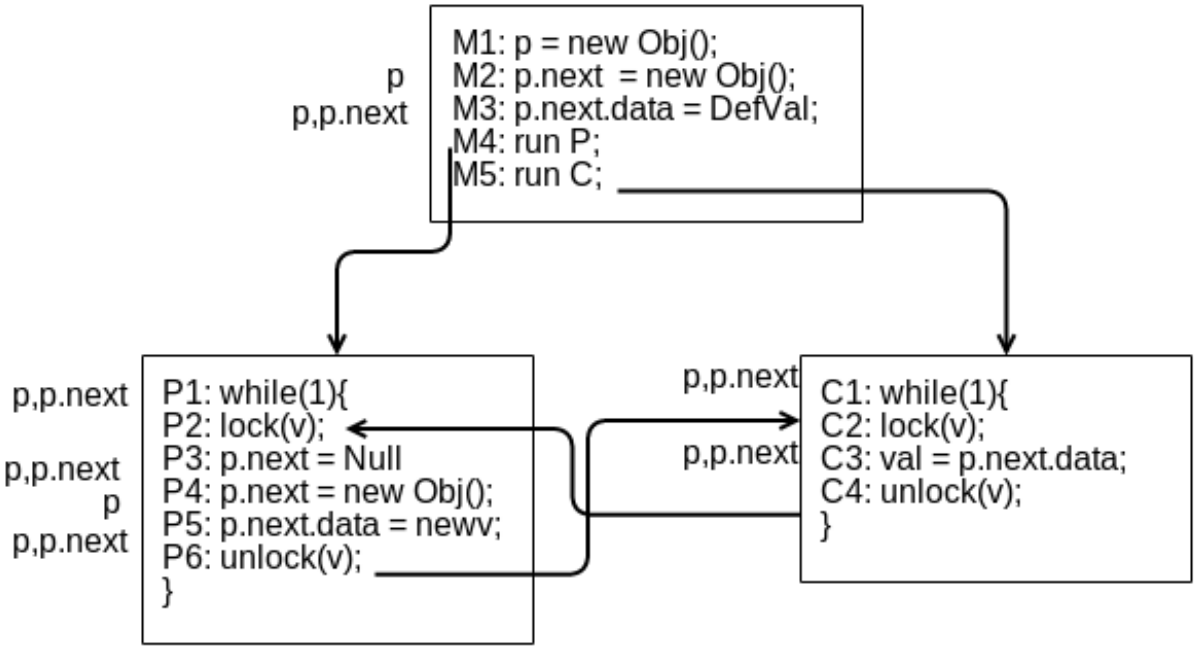


Figure: Heap Access path based null pointer analysis

- 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

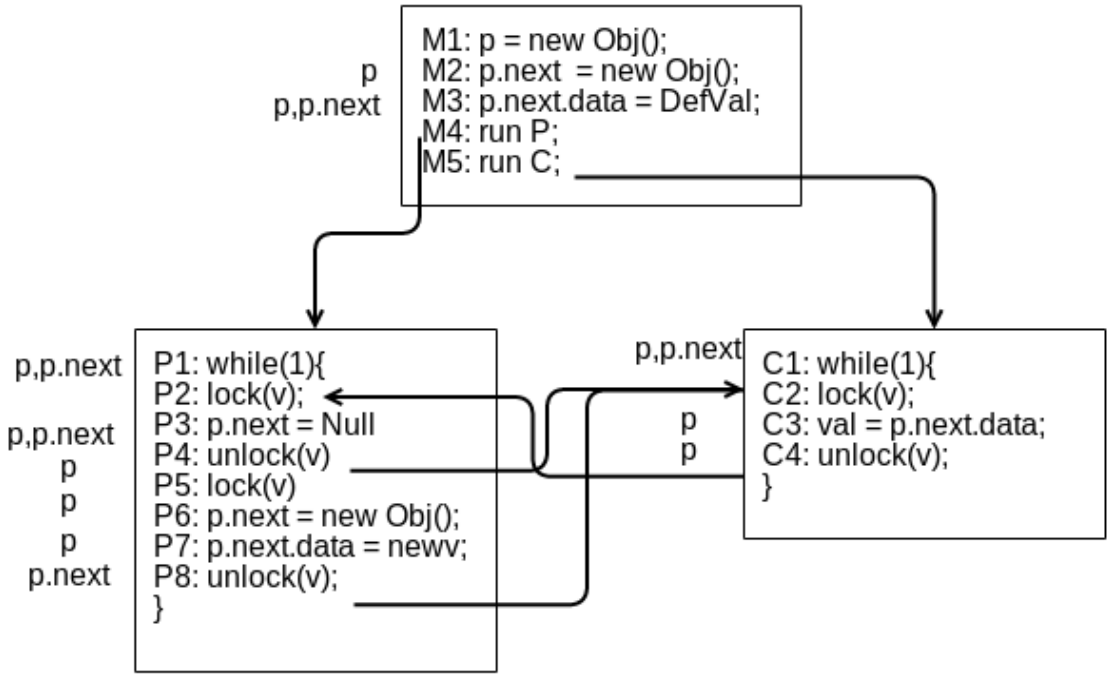


Figure: Heap Access path based null pointer analysis

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

Concurrent Heap Liveness Analysis

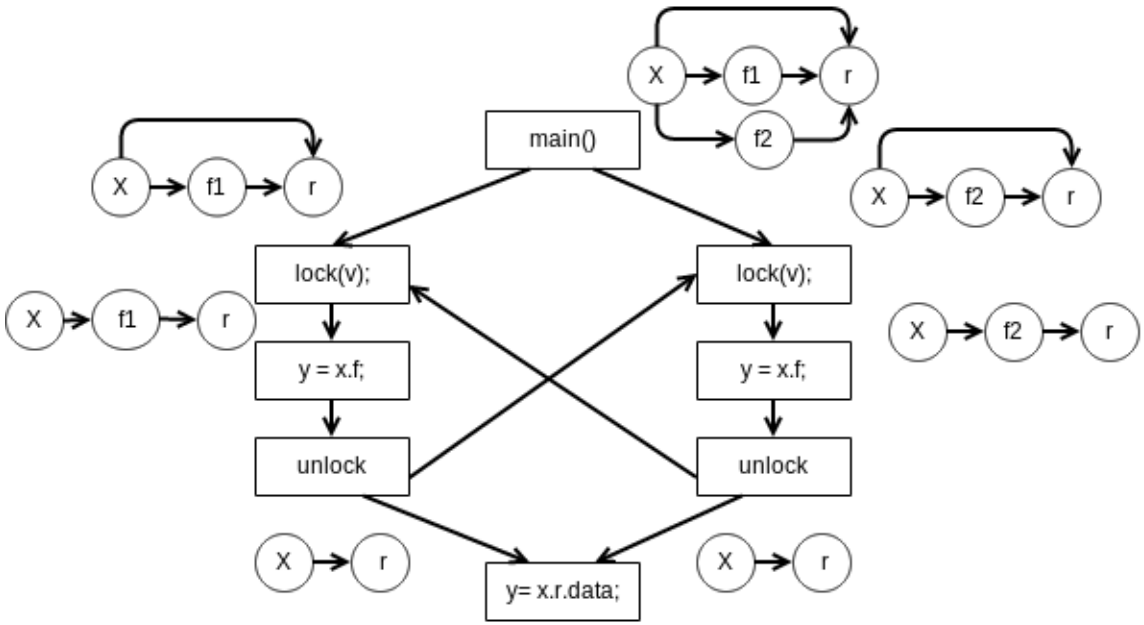


Figure: Concurrent Heap Liveness Analysis

Notice how the data flow value is precise only for statements within the lock-unlock section.

Analysis to be Implemented

- To implement concurrent heap liveness analysis.
- To extend the concurrent analysis technique to inter-procedural level using the method of value contexts. Support function calls in the critical section of the program.

Sync-cfg for Concurrent Heap Liveness

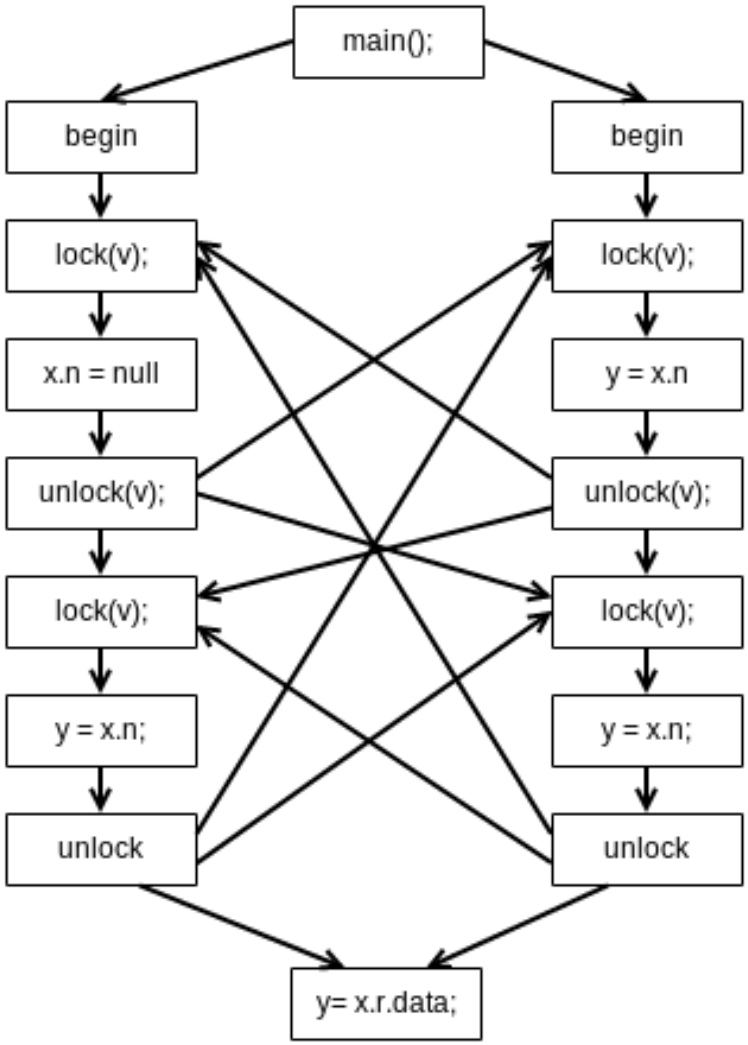


Figure: Concurrent Heap liveness analysis input

Tools to Use

- *SOOT*: Java Byte Code Optimization Framework. Can implement precise intra-procedural analysis. SPARK engine provided call graph.
- *VASCO*: Framework for carrying out precise inter-procedural analysis. Returns a better call graph as compared to SPARK.
- *CombinedUnitGraph*: Generation of sync-cfg for programs containing upto 2 threads. Need to extend it to n threads.

Thank You