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.

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.

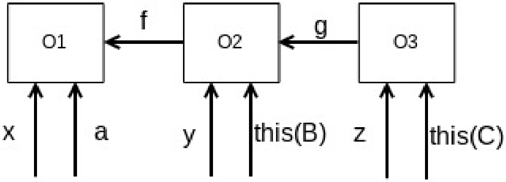

Figure: Example for points-to graph

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.

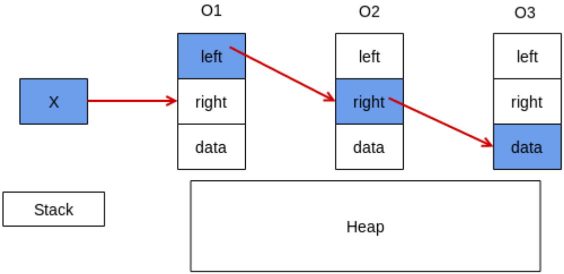

Figure: access path for the expression *x.left.right.data*

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.

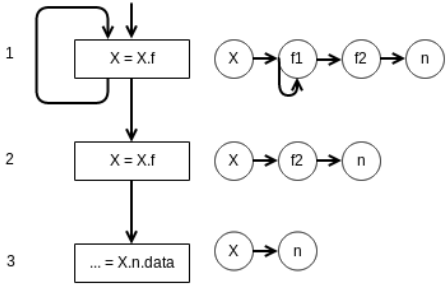

Figure: Use of access graph and liveness data flow values

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-Sensitive analysis

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

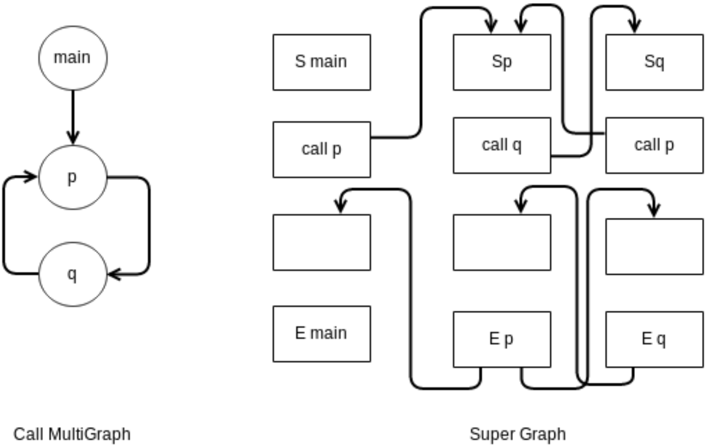

Figure: Call Multi-graph and super graph examples

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

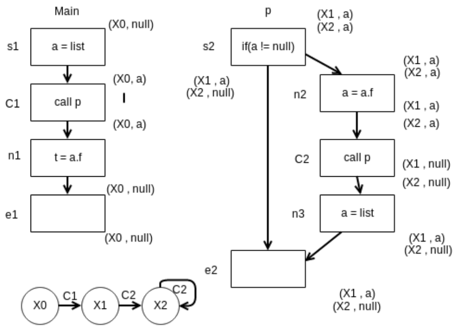

Figure: Example for inter-procedural heap liveness

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 : 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

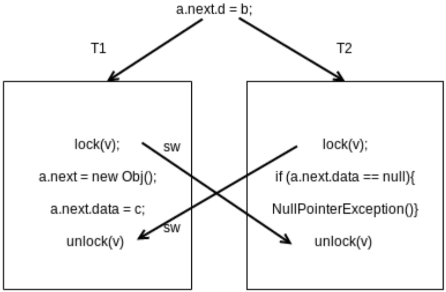

Figure: Happens Before memory model with thread synchronization

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

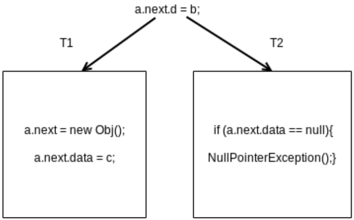

Figure: Happens Before memory model without thread synchronization

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

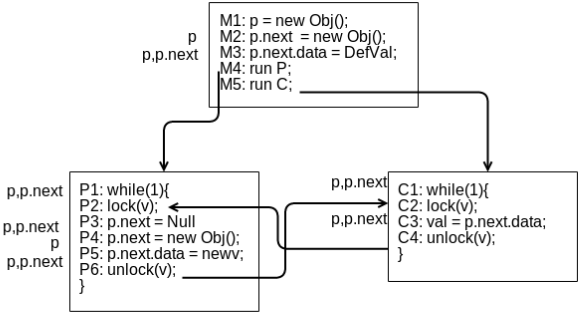

Figure: Heap Access path based null pointer analysis

- 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

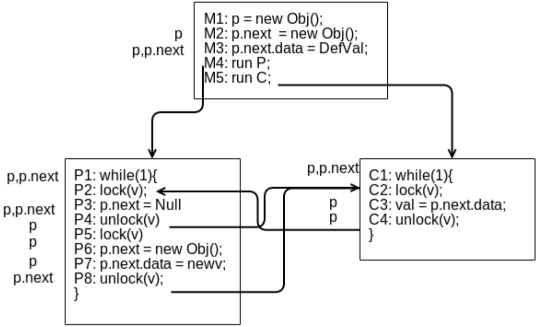

Figure: Heap Access path based null pointer analysis

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

Concurrent Heap Liveness Analysis

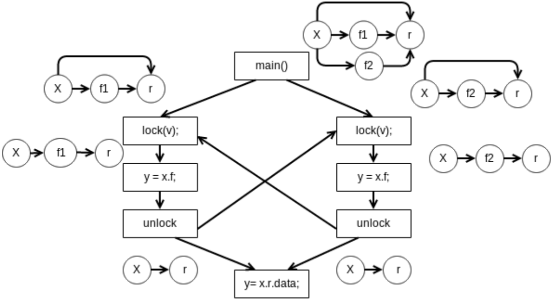

Figure: Concurrent Heap Liveness Analysis

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

Sync-cfg for Concurrent Heap Liveness

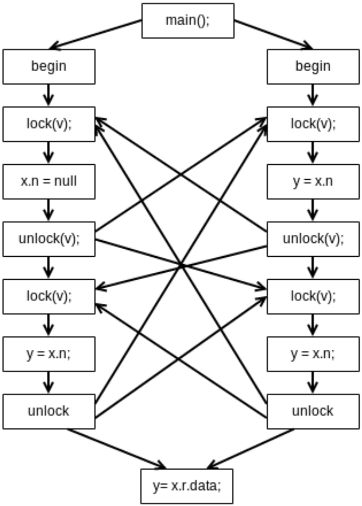

Figure: Concurrent Heap liveness analysis input

Example for concurrent heap liveness analysis

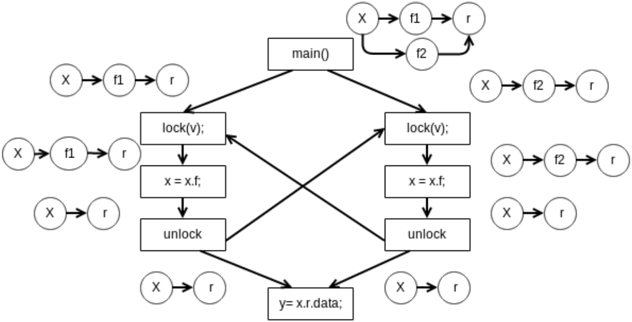

Figure: Concurrent Heap liveness analysis iteration 1

Example for concurrent heap liveness analysis

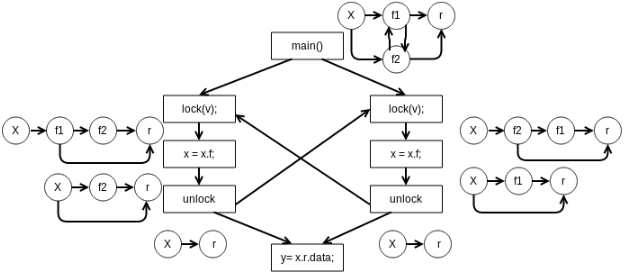

Figure: Concurrent Heap liveness analysis iteration 2

Example for concurrent heap liveness analysis

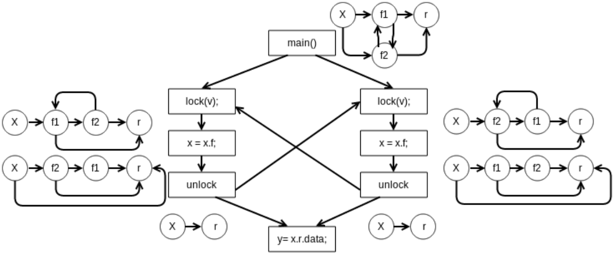

Figure: Concurrent Heap liveness analysis iteration 3

Example for concurrent heap liveness analysis

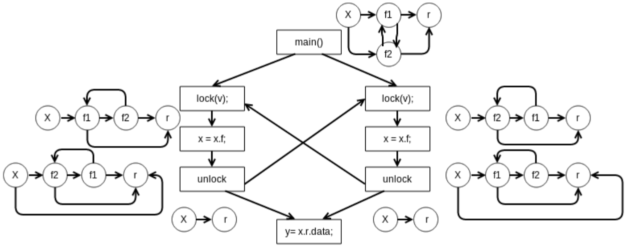

Figure: Concurrent Heap liveness analysis iteration 4

Example for concurrent heap liveness analysis

Example of the same program without synchronization edges.

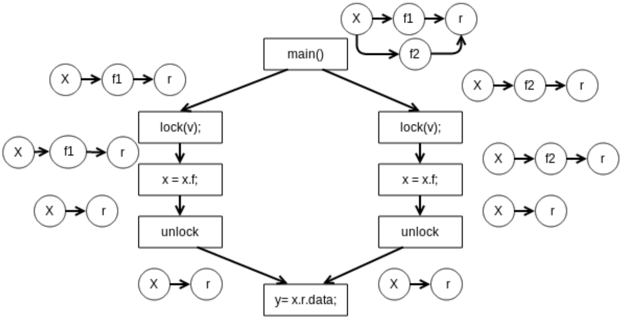

Figure: Concurrent Heap liveness analysis without synchronization edges

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
- With this , there is a need to 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.

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