Inter-Procedural Heap Reference Analysis for Concurrent Programs

B.Tech. Project 1st Stage Report

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology (Honors)

Student:
Anshul Purohit
Roll No: 110050002

Guide:
Prof. Uday Khedkar

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India

Abstract

This report mainly deals with designing inter-procedural heap reference analysis for concurrent programs in Java. The analysis is both flow-sensitive and context-sensitive at the inter-procedural level. Heap Reference analysis determines a collection of objects pointed to by program variables or their fields. Java implements pointers by references. This report also discusses about analysis techniques for concurrent programs at intra-procedural level and about extending analysis to inter-procedural level using the VASCO tool.

Contents

Chapter 1

Introduction

1.1 Background on Program Analysis

Program Analysis is the set of techniques to get appropriate information about the behavior of programs. Static analysis techniques involve computing approximate information about the program without executing it. In this report, only static analysis techniques would be discussed.

Data flow analysis is a technique for gathering information about the possible set of values calculated at various points in a program. The value at each program point is then propagated in the Control Flow graph (CFG) of the program. The information gathered is often used by compilers when optimizing a program.

1.2 Types of Data Flow Analysis

A data flow analysis can be either flow-insensitive or flow-sensitive.

Flow-insensitive analysis:

- Ignores the control-flow graph, and assumes that statements can execute in any order.
- Rather than producing a solution for each program point, produces a single solution that is valid for the entire program

Flow-sensitive analysis:

- Takes the control flow structure of a program into account

- Has the computed abstract state that represents different reachable memory states at different program points.

A flow-sensitive version of an analysis is more precise and expensive than the flow-insensitive version.

Categorizing according to the direction of traversal of program statements in the CFG, data flow analysis can be broadly of 2 types : forward and backward. Examples of data-flow analysis are Available Expressions analysis which is a forward data flow analysis and Live Variables analysis which is a backward data flow analysis.

1.3 Inter-Procedural Analysis

Inter-procedural analysis operates across an entire program makes information flow from caller to callee and vice-versa. It extends the scope of data flow analysis across procedure boundaries and it incorporates the effects of procedure calls in the caller procedures, and calling contexts in the callee procedure. A context-sensitive analysis is an **interprocedural analysis** which reanalyzes callee procedure for each context whereas context insensitive analysis performs analysis independent of calling context.

1.4 Concurrency

For concurrency, we assume the thread model in Java. The high level abstractions of concurrency (such as the DOALL, FORALL, PARBEGIN , PAREND constructs of FORTRAN) can be modeled in terms of threads and so we use the model of threads to uniformly refer to concurrent programs. However, we would expect a data-race free program

1.5 Pointers in Java

Java does not allow the address of a variable to be taken and thus does not allow assignments of the form y = &x. Java only provided heap-based pointers. Dereferencing is only possible through object fields in Java as the allocated data items do not possess compile-time name in the program. Java does not provide the * operator for dereferencing.

1.6 Organization of the Report

In this chapter, an introduction to the basics of Data Flow Analysis and it types was provided. I have also mentioned about concurrency model used and pointer analysis for Java and how it is different from that of C/C++.

In the subsequent chapters , I will be covering Heap Reference Analysis, Concurrent Data Flow Analysis and Inter-Procedural Analysis respectively. I would then present the tools required for the implementation in the next chapter followed by summary and future work.

Chapter 2

Heap Reference Analysis

Analysing properties of heap data is not very trivial. This is because the spatial and temporal structure of stack and static data is simple to understand. The stack variables have a compile-time name(alias) associated with it. However, this is not the case with heap data. We need to devise a flow and context sensitive points-to analysis to get information from heap data.

2.1 Difficulties in Analysis of Heap Data

A program accesses data through expressions having l-values and hence are called access expressions. The l-values can either be a scalar (x), or may involve array access such as a[2*i] or can be a reference expression like x.l.data. In the case of reference or array, the mapping of the access expression and the l-value may change. The reference expression as primarily used to access the heap.

Heap analysis tries to find out the answer to the question: Can an access expression at program point a1 at program point p1 have the same l-value as a2 at program point p2.

2.2 Pointer Analysis

Pointer analysis is a static analysis technique that establishes which pointers or heap references can point to which variables. Pointer analysis collects information about indirect accesses in programs. It can enable precise data analysis and precise inter-procedural control flow analysis. The latter will be used in the VASCO tool that will be discussed later in the report.

We have two types of pointer analysis information: points-to analysis and alias analysis. Alias analysis tell if two references point to the same location on the heap. It is transitive in nature. Whereas, points-to analysis tells which memory locations are pointed by references at run-time.

In Java pointers are not created explicitly. All objects in Java are accessed using references and these references are termed as pointers here. Every time we create an object in Java it creates pointer to the object. This pointer could then be set to a different object or to null, but the original object will still exist. Thus points-to analysis for Java programs identifies the objects pointed to by references at run time. Thus we wish to determine the objects pointed to by a reference variable or a field.

Alias information plays an important role in the liveness analysis. Must-alias information is needed to improve precision. May aliases can be found out using information from the points-to analysis.

Liveness and Points-to Analysis : We can remove some information from the points-to analysis result by considering only information for live pointers. For field references and indirections, liveness is defined using points-to information.

Points-to graph in Java contain two types of edges. The first type of edge is to represent the information that reference variable v pointing to object o. The second type of edge represents the field f of o1 pointing to o2. Example figure.

2.3 Heap Reference Analysis

A reference can be represented by access path. In order to perform liveness analysis of heap and identify the set of live links, naming of links is necessary. This is achieved by access path. An Access Path is defined as root variable name following any number of field names and is represented as x $\rightarrow$ n1 $\rightarrow$ n2 ....nk where x is root variable, n1 , n2 .. are field names. If access path x $\rightarrow$ f $\rightarrow$ d is live then, the the objects pointed to by x, x $\rightarrow$ f and x $\rightarrow$ f $\rightarrow$ d are live.

An access path can be unbounded in the case of loops. Thus, we require to set a bound on the representation for liveness information. This is achieved using access graphs which summarizes information based on allocation sites.

Access Graph is a directed graph representing access paths starting from root variable. Root node is connected to any number of nodes each having unique labels of form ni where n is name of the field and i is the program point. Inclusion of program points in access graphs helps in summarization. Example of access graph and liveness analysis.

Chapter 3

Inter-Procedural Analysis

Interprocedural Analysis is required to 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. Handling concurrency also needs to be supported , which would be discussed in the next chapter.

3.1 Context Sensitivity

A context sensitive analysis is an inter-procedural analysis which analyses callee procedure for each context whereas context insensitive analysis performs analysis irrespective of calling context. Context insensitive analysis over-approximates inter procedural control flow which results in imprecision because it takes into account invalid control paths. Each function is analyzed once with single abstract context. Whereas context sensitive analysis is more precise as it considers the valid inter procedural control flow.

Java is an object oriented language supporting features like encapsulation and inheritance. Thus data access is indirect through method calls for each class. So sontext sensitivity plays a very important role for Object Oriented languages. To add examples of object sensitivity.

3.2 Call Graph

Call Graph is graph with nodes and edges in which nodes represent procedures and there is edge from a to b if some call-site at a calls procedure b. Hence this is a static data structure that represents the run-time calling relationships among procedures in program. Soot provides a Spark engine which generates the call graph. VASCO on the other hand returns a much precise call graph, which is generated using liveness based inter-procedural pointer analysis. A thing to note here is that construction of call graph requires inter-procedural analysis and inter-procedural analysis on the other hand requires call graph. An approach to breaking this dependency is to initially approximate the call-graph and in every iteration perform inter-procedural analysis and improve the precision of the call graph.

Call multi-graph is a directed graph which represents calling relationships between procedures in a program where nodes represent procedures and edges procedure call. A cycle in the call multi-graph denotes recursion. Super graph is another representation in which callsites are connected to the callee procedure entry node and and the exit node of the callee is connected to return node in the caller procedure.

Data flow analysis uses static representation of programs to compute summary information along paths. For ensuring safety, all the valid paths must be covered. A valid path is the path which represents legal control flow. Ensuring precision is subject to merging data flow values at shared program points without including invalid paths. For ensuring efficiency, only those valid paths that yield information that affects the summary information should be covered.

3.3 Approaches to Inter-procedural Analysis

In this section approaches to perform inter-procedural analysis is discusses. A very simple approach is to perform procedure in-lining where every procedure call is replaced by the procedure body. this would however only be applicable when target of the call is known and call is not made by pointers or is virtual. However this is not good way to handle recursion as the code size can increase in an unbounded manner.

3.3.1 Functional approach

In the functional approach, summary flow functions are computed for each function. The summary flow functions are used as the flow functions for the procedure call. The summary flow function of a given procedure is influenced by the summary flow functions of the callees of r and not by the callers of r. Also in the presence of loops or recursion, iterative computation will be needed till fixed point is achieved. Termination is only guaranteed if, lattice is finite.

3.3.2 Call Strings Approach

This is a general flow and context sensitive method. In this approach the call history stored for information to be propagated back to the correct point. 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 ci , ci is appended to the call-string value at that point. Similarly at a return node the last call site ci is removed. And other data flow values are blocked. For non-recursive programs number of call strings are finite. For recursive programs, number of call strings can be infinite. However, the problem is decidable for finite lattices.

There is an approach of value based termination of call-strings. This method deals with creating equivalence classes. If two call-strings have same data flow values at the start node of the procedure, then they will produce the same data flow values at the return node of the procedure call. Such call strings are grouped into equivalence classes.

3.3.3 Value Context Method

In this method, a combination of tabulation (functional approach) and value based termination (call strings) approach is adopted. The call-stings are partitioned based on the data flow value at the call site. And then analysis of the procedure can then be performed once for each partition. It also combines the two views of contexts: data flow values at call site are stored as value contexts and call strings as calling contexts. Distinct data flow values are maintained for each context of a procedure.

A value context is defined by a particular data flow value reaching a procedure. It is used to enumerate and store the summary flow function

of the procedure in terms on input and output pairs. In order to compute these pairs, data flow analysis is performed within the procedure for each context(input data flow value). Each context is initialized with the top element initially. This approach also maintains a context transition table which allows flow of information along inter-procedurally valid paths. Transitions are recorded in the form (X , c) - Y where X represents calling context, c repents call site and Y represents callee context. Hence when analysis of callee procedure completes where to return is identified by context transition table, therefore propagation to valid path is guaranteed.

When a new call to a procedure is encounterd, the pairs are consulted to decide if the procedure needs to be analysed again. If it was already analysed once for the input value, output can be directly processed else a new context is created and the procedure is analysed for this new context.

3.4 Example of value context method

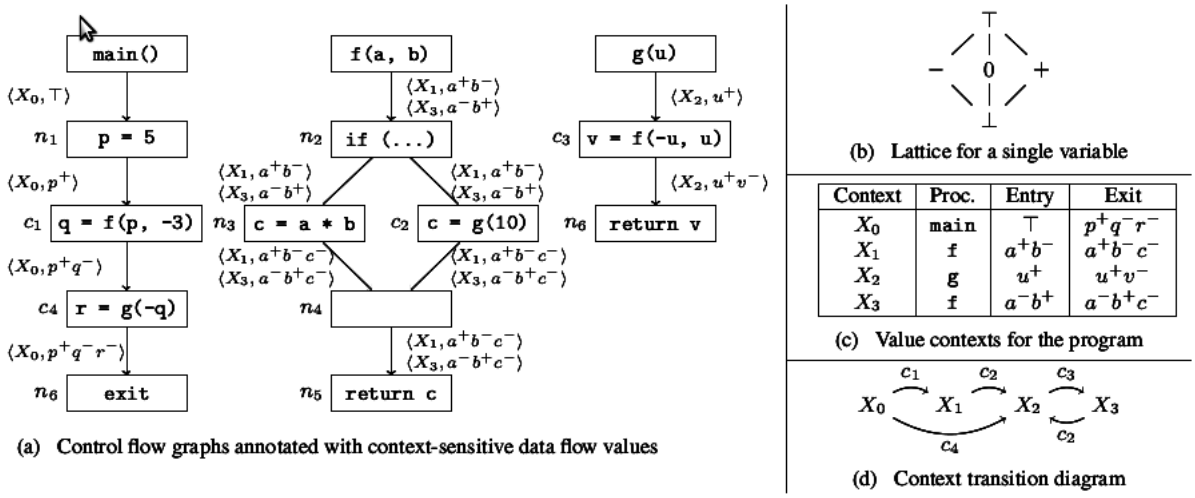

Figure 3.1: Example of value-context based IP analysis

This is the example from the paper on IP data flow analysis in soot using value contexts. In this example we wish to perform sign analysis to determine whether a scalar local variable is positive, negative or zero. There is mutual recursion in the f and g procedure calls.

The analysis starts with the intial context X0 for main with the value top. The flow function for n1 is processed , which makes p positive. On the next node c1, we have a function call to f whose first argument is positive and second argument is negative. So a new value context X1 is created with input a+b- and value top and the transition for X0 to X1 is noted. Now in X1 context, node n2 is processed followed by c2, which creates another value context X2 for procedure g with input u+ and value top. The transition X1 to X2 is also recorded in the context transition table.

In the context X2, node c3 is evaluated. The arguments to f are negative and positive , creating a new context X3 with input u-v+ and value top. Also the transition from X2 to X3 is noted. The worklist now picks up nodes of context X3 and when node c2 is picked up, the entry value at g is u+ for which context X2 exists. The transition from X3 to X2 needs to be recorded. The exit value for context X2 at the moment is top as its exit node has not been processed yet. Thus the call-exit flow function sets the values of OUT c2 in X3 to a-b+. The next node processed is n3 in X3, whose flow function computes the sign of c to be negative. Thus, the IN value at n4 in X3 is merge of a-b+ and a-b+c- which is a-b+c-. Thus the sign of the returned variable c is found to be negative. the exit value is modified to a-b+c-. Also as n4 is the exit node of the procedure f, the callers of X3 are looked up in the transition table and added to the work-list.

The only caller of X3 is X2 which is reprocessed now. The exit value of X3 being a-b+c- the returned variable v becomes negative. So the value at the out of C3 in X2 becomes u+v-. This is added to the exit value of X2. And now the callers of X2 which are X3 and X1 are added to the work-list.

X3,c2 is picked up for processing and this time the correct exit value of X2 i.e u+v- is used and so the out value of X3,c2 becomes a-b+c-. However the values do not change for its successor. Thus no more nodes of X3 need to be expanded. The process continues with X1,c2. It's out value comes out to be a+b-c- as the exit value of X2 is u+v-. X1,n3 is then picked up. The sign of c is found out to be negative and this propagates to the end of the procedure. The exit values of X1 becomes a+b-c-. And now X0,c1 is added to the work list for processing and thus q is found out to be negative.

Value based contexts are used as a cache table for distinct call sites apart from terminating analysis of recursive procedures.

Chapter 4

Analysis for Concurrent Programs

We will be using the technique mentioned in the paper Dataflow Analysis for Datarace-Free Programs. This technique when given a sequential data-flow analysis produces an efficient and fairly precise analysis for concurrent programs. The criteria to be met for applying this analysis is that data-flow fact should be dependent on the contents of the associated lvalues (expression referring to memory location at runtime). Sequential analysis like null-pointer analysis, interval analysis and constant propagation. In terms of precision of the data-flow facts, useful information will be derived at program points where the lvalue is read.

The main challenge in converting the analysis from sequential to concurrent programs is that propagating data-flow values such that all the possible concurrent executions of the thread are taken care of. In this technique, the synchronization structure of the program is made use of to propagate data-flow values. The main insight used is that the data-flow values are only propagated between threads at the lock and unlock points in threads for access to critical sections. Also this approach will not work for programs containing data race.

4.1 Memory Model and Data Races

A memory model specifies the interactions of threads with memory and its shared use. Thus , memory model specify the constraints on data access and conditions on how data written by one thread is accessible to other threads.

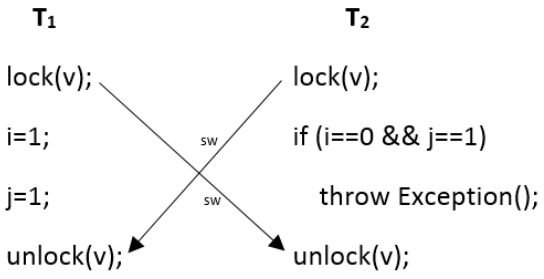

Figure 4.1: Happens Before memory model with threads

Happens before is the memory model described in the paper and thesis on Concurrent Program analysis.

The happens before memory model is based on the happens-before relation which relates the happenings of two events such that if one happens before the other, it should be reflected in the results. That is if statement A occurs before statement B then the memory written by statement A is visible to statement B.
There are three components of the happens before memory model.

1. Program Order: It is generally used to refer to the order of statement in a thread as they appear in the program. However the formal definition of program order is the intra-thread order during the execution of the program.

2. Synchronizes-with Relation: This relation is defined over synchronization relations which are lock, unlock (for acquiring/releasing locks), spawn, join (to synchronize creation/endind of multiple threads), start, end(first/last statement of a thread). The synchronizes-with relation is defined from

    - Lock statement to all previous unlock statements

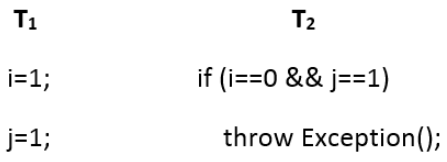

Figure 4.2: Happens Before memory model without threads

- A begin statement to the spawn statement of the thread
- Join statement to the end statements of all the threads synchronized

3. 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.

Consider the example presented in the thesis on concurrent data flow analysis techniques in Figure 4.1. The variables i and j have initial values to be 0. We need to check if the exception in T2 can be thrown. Applying the happens-before relation, we can find out this will never be the case as the synchronization of locks and unlocks always prevent the possibility of i getting value 1 and j getting value 2 at the program point , irrespective of any possible thread scheduling.

Now consider the case given in Figure 4.2. The program does not contain synchronization edges, so all the actions in T1 need not happen before the evaluation of the condition. The exception is now thrown when the two statements in T1 are flipped by the compiler. Note that this can happen as the statements are independent.

Data Race : A data race occurs when two or more threads can access shared data at the same time and try to change it at the same time. Also the order of access among the threads will not be known, so both the threads are racing to access or modify the shared data. Formally, a data race is said when

1. Two or more threads access the same memory location concurrently
2. At least one of the accesses is a write
3. he threads are not using any exclusive locks to control their accesses to that memory.

Data race can be defined in terms of happens before relation. Two statements are said to be conflicting/racy if neither a happens before b or b happens before a. We will only deal with programs which are data race free.

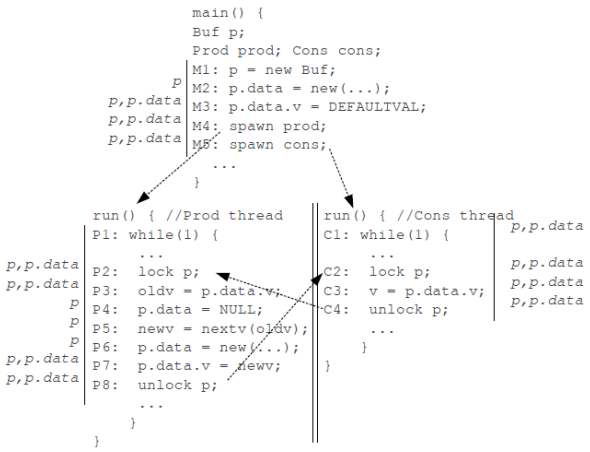

Figure 4.3: Null pointer analysis example program from Arnab's paper

4.2 Concurrent analysis technique

Arnab De has explained his approach to concurrent data flow analysis by giving example of access path based null-pointer analysis. He argues that the data-flow value is true only before the statement where the corresponding access path/lvalue is relevant.

The first step in the analysis is to add edges between nodes of control flow graph representing different threads. These edges map to the synchronize-with edges in the happens before memory model. Hence, the name sync-CFG is given to this control flow graph. The edges are added from the spawn statement to the first statement of each thread, from the unlock statement to lock statement if they access the same lock variable and belong to different threads. In the next step sequential data flow analysis is performed on the syn-CFG. The synchronization edges have identity flow functions. The example of the null pointer analysis from Arnab De's paper is shown in Figure 3.1

In Figure 3.1 the synchronization edges are added to generate the sync-CFG.It also shows access paths to memory locations which are discovered to be not null by the analysis. The values are marked in italics for each corresponding statement.

As p.data is non-null at point M5 in the main thread before spawning the cons thread, this value gets propagated to the first instruction C1 of

the cons thread though one of the added edges, and from there to the lock instruction at C2. Similarly, although p.data is set back to non-null at P 6 before the unlock, despite being set to null in the prod thread at P4 in the prod thread. This fact also gets propagated to the lock statement of the cons thread through the edge P 8 to C2. As p.data is not null in both the paths joining at C2 of the cons thread, we can determine p.data to be non-null before the lock statement in all executions by the merge operation.

An important point to note about this technique is that, it may compute incorrect values at program point not containing relevant access path. For all data-race free programs, relevant statements will occur in the lock-unlock regions. In the given example C1 is not a relevant statement and the data flow value p→data is incorrect considering the program execution order [P4 C1].

Chapter 5

Design of the Analysis

For programs with procedure call, context-sensitive analysis is required to improve the precision of data flow facts. It is needed to extend the concurrent analysis technique to inter-procedural level. In the paper, an extension based on the call strings approach was proposed. However we wish to perform inter-procedural analysis using value contexts, for better way to handle recursive programs.

A thread now consists of a number of procedure calls, each with their own CFGs. Each thread has an entry procedure with the same name as the thread. Execution of a thread starts with the execution of the start node of the entry procedure. Due to the inter-procedural nature we would need to handle call and return statements by adding edges from the call statement to the root node of the called procedure CFG and edges from the return statement of the called procedure CFG to the statement immediately following the call statement.

As an input for the concurrent inter-procedural analysis, we will supply programs with multiple threads with procedure calls in the main thread functions which can be recursive.

5.1 Call Graph Generation

Call graph construction can be done using Soot. The call graph is available only in whole program mode of soot (-w option). It can be accessed through the getCallGraph method. Alternatively, call graph can also be constructed using the VASCO using the class vasco.callgraph.CallGraphTest in the vasco.callgraph package. VASCO generates a much precise call graph

using flow and context sensitive points to analysis over the program. The arguments that need to be given along to run this class is the classpath containing the soot jar file, the output directory, maximum depth of call chains and the main class indicating the entry point of the program.

The command is:
java [-cp CLASSPATH] vasco.callgraph.CallGraphTest [-out DIR] [-k DEPTH] MAIN_CLASS.

For example
java -cp bin:jars/ vasco.callgraph.Test -out "vasco-output/" -k 9 tests.test* is executed from the project root directory. The class file to be given as input is int the package tests.

For the given input program

```
package tests;
public class test {
   static class A {
      void foo() { bar(); }
      void bar() { }
   }

   public static void main(String[] args) {
      A a1 = new A();
      a1.foo();

      A a2 = new A();
      a2.foo();

      a2.bar();
   }
}
```

We get the calls in the main procedure as
PCG Method : main → foo
PCG Method : main → bar
PCG Method : main → foo
PCG Method : main → <init>
PCG Method : main → <init>

5.2 Combined Unit Graph

We will be using the CombinedUnitGraph API to create a sync-CFG required to perform the analysis. This is currently implemented to handle 2 threads and perform intra-procedural analysis. This needs to be improved to handle more than 2 threads and construct inter-procedural CFG as discussed.

Creating an intra-procedural CFG : We first create an object of the CombinedUnitGraph class by passing the body of the 1st CFG (for the first thread) to be combined to its constructor. Then, we call the addLockUnlockUnitsForThread1 method to add the lock and unlock nodes in the 1st CFG and store them in respective lists. Now, we can add the 2nd CFG to this object by calling the addGraph method with the 2nd graph as an argument. This is to be followed by a call to the addLockUnlockUnitsForThread2 method and finally the addUnlockToLockEdges method in order to complete the construction of the sync-CFG by joining unlock to lock edges for the same lock object. Exact API details are provided in the thesis on Concurrent Analysis of programs.

Consider this input program for intra-procedural live variable analysis:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LiveVariablesInput {
private Lock lock = new ReentrantLock();

private class Thread1 extends Thread {
   public void run() {
      int a=0,b=0;
      lock.lock();
      a = 0;
      lock.unlock();

      lock.lock();
      b = a;
      lock.unlock();
   }
}

private class Thread2 extends Thread {
```

```
   public void run() {
      int a=0,b=0;
      lock.lock();
      a = 0;
      lock.unlock();

      lock.lock();
      a = 0;
      lock.unlock();
   }
}

public static void main(String[] args) {
   LiveVariablesInput ip = new LiveVariablesInput();

   Thread t1 = new Thread(ip.new Thread1());
   t1.start();

   Thread t2 = new Thread(ip.new Thread2());
   t2.start();
   }
}
```

The combined unit graph creates a sync-CFG in Figure 5.1 for the statements in the jimple representation of the program. The class file is converted to Jimple using Soot.

5.3 VASCO framework

We have already seen the use of VASCO framework to generate precise call graph for inter-procedural programs. VASCO framework is also used to design inter-procedural analysis. To perform and analysis, it is needed to override the the methods *normalFlowFunction, callEntryFlowFunction, callExitFlowFunction, callLocalFlowFunction, boundaryValue, copy, meet, topValue, programRepresentation* from the ForwardInterProceduralAnalysis (or Backward depending on the nature of the analysis) of VASCO.

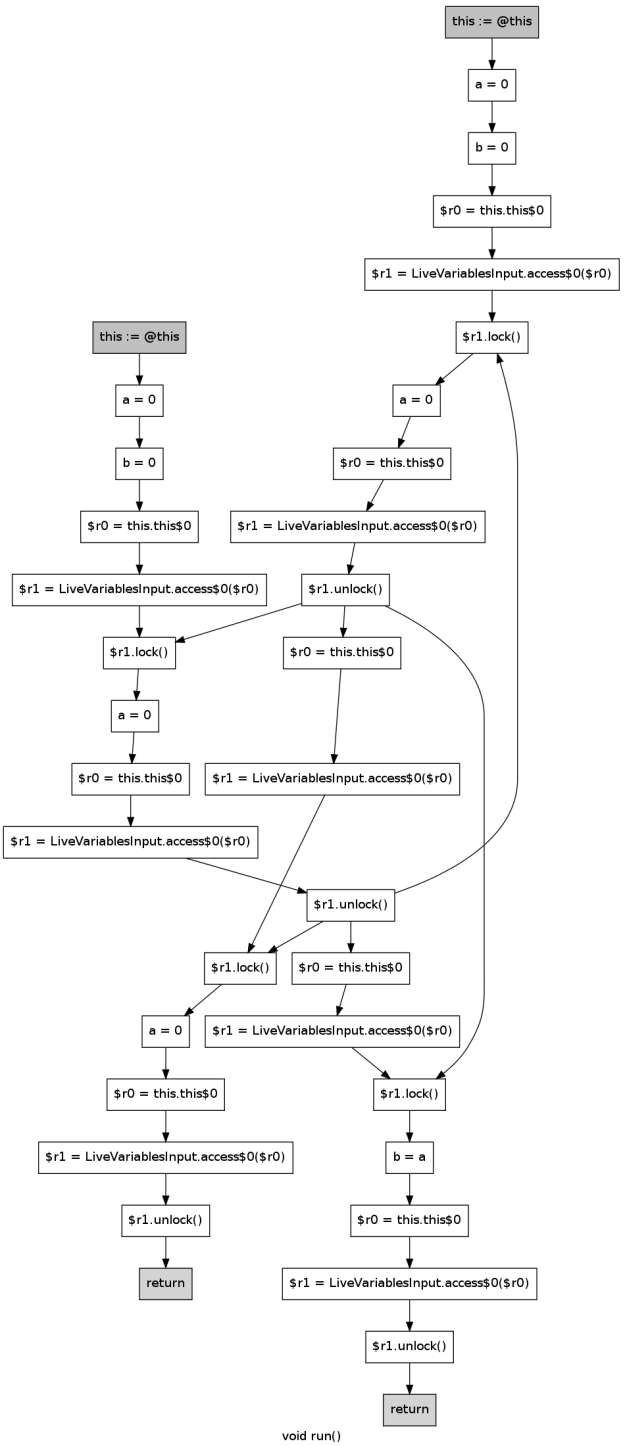

Figure 5.1: Sync-cfg for intra-procedural live variable analysis

Chapter 6

Conclusion and Future Work

In this report the topics covered are :

- The mechanism of Heap reference analysis using points-to and live variable analysis on access paths. It is necessary to store the access path values at a statement as an access graph to keep the data fact bounded.
- The method of value context used to carry out precise inter-procedural analysis with an example case on handling recursion.
- The technique of constructing sync-cfg for performing intra-procedural data flow analysis of concurrent programs.
- The extension of concurrent analysis to an inter-procedural level, by using VASCO for generating call-graph and designing inter-procedural analysis.

The future work to be done is:

- Implementation of CombinedUnitGraph class to handle programs with more than 2 threads.
- Using Value context method of performing inter-procedural analysis on data-race free concurrent programs containing function calls.
- Implementing Heap reference analysis on concurrent programs containing function calls.

Bibliography

[1] Alan Mycroft Uday P. Khedker and Prashant Singh Rawat. *Liveness-Based pointer analysis*. SAS'12 Proceedings of the 19th international conference on Static Analysis. 2012.

[2] Uday P. Khedker. *Slides on Interprocedural Data Flow Analysis*. Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. 2013.

[3] Amitabha Sanyal Uday P. Khedker and Amey Karkar. *Heap Reference Analysis Using Access Graphs*. ACM Transactions on Programming Languages and Systems. 2007.

[4] Aditya Bhandari. *Data Flow Analysis for Concurrent Programs*. BTP Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. 2013.

[5] Arni Einarsson and Janus Dam Nielsen. "A Survivor's Guide to Java Program Analysis with Soot". Notes for the Soot framework from McGill University. 2008.

[6] Deepak DSouza Arnab De and Rupesh Nasre. *Dataflow Analysis for Datarace-Free Programs*. 2010.

[7] Alefiya Lightwala. *Interprocedural Liveness Based Heap Points-to Analysis for Java*. MTP Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. 2013.