Context-sensitive Bottom-Up Pointer Analysis for Parallel Java Programs

Rajat Singhal, CS17B042 Rushabh Lalwani, CS18B046

May 2021

1 Abstract

This report contains an analysis of the paper - **Bottom-up Context-Sensitive Pointer Analysis for Java**. The paper discusses a modified context-sensitive field-sensitive pointer analysis. This analysis does a constraint based handling of virtual method calls. It creates a polymorphic method summary once for a given method and uses it multiple times for each call site and hence obtains context-sensitivity without analysing the method multiple times. Further, we extend this analysis to handle parallel Java programs by utilizing May-Happen-in-Parallel (MHP) information.

2 Motivation

Pointer analysis or points-to analysis statically determines the possible values for a pointer, variable or object reference. It is a fundamental analysis which predicts what could the variables possibly point to during run-time. In this analysis, typically heap memory is abstracted as finite set of memory objects pointing to the subset of these memory objects at run-time. A very basic fully conservative points-to analysis would suggest all the variables and heap objects points-to all other heap objects. There are variants of this analysis depending on the flow and context sensitivities which improve its precision. As we add more sensitivity in the analysis the points-to information becomes more precise. Highly sensitive analyses have been proven too slow in practise.

In context-sensitive analysis, context for a method call is the current heap at the invocation site. Each method can be called at multiple program points which are distinguished with these contexts. The number of contexts in a program can become exponential and hence context-sensitive analysis can have scalability and termination issues. The main challenge in context-sensitive analysis is to analyze the method in different contexts (method calls) while context-insensitive analysis need not worry about the contexts and maintain a single-summary. In a usual context-sensitive analyses, methods are re-analyzed multiple times under multiple contexts which can potentially lead to exponential computation. This paper proposes a bottom-up inter-procedural analysis. It maintains a polymorphic summary of the method such that it can be used in any calling context and produce the context-sensitive summary easily without reanalyzing the method multiple times at different contexts.

The key novelty in the analysis is constrained treatment for virtual method calls. The dynamic types gets resolved and method summary gets created in a context-sensitive manner. Secondly this approach while creating polymorphic method summaries, it does not perform expensive case splits on possible alias patterns at method call sites. The analysis does have strong updates for locals while performing intra-procedural analysis, so it is partially flow-sensitive.

3 Algorithm

3.1 Heap Representation

The analysis uses a may-points-to graph referred to as an *abstract heap* to represent variables and the heap objects which they might point to. For an abstract heap H, a *normalization operation N(H)* yields a normalized heap H^* and a mapping ζ from nodes in H to nodes in H^*.

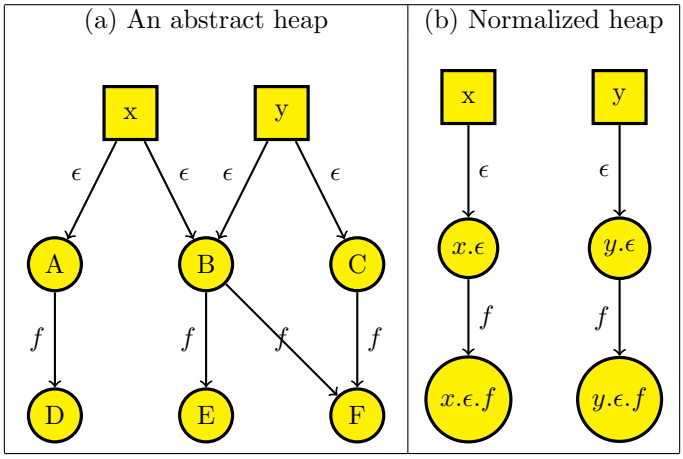

Figure 1: Abstract Heap and Normal form

Normalized heap is used to represent the heap at the entry of a method. The mapping ζ from abstract heap H to normal form H^* differs for each call site, but the normalized heap for a method is the same irrespective of the calling context. ζ^-1 is defined to be a mapping of nodes from H^1 to H. The mapping ζ^-1 is used to instantiate a method summary to a particular abstract heap at a call site.

3.2 Summary-based Pointer Analysis

SumAnalyze(H, S, A):

input: abstract heap H, code S, intraprocedural analysis A
output: abstract heap H'

(1) let $H^* = \text{NormalForm}(H)$
(2) let $H'^* = \text{Analyze}(H^*, S, A)$
(3) let $\Delta = H'^* \backslash \text{default}(H'^*)$
(4) let $\zeta_0 = \text{Map}(H, H^*)$
(5) let $H' = H$; let $\zeta = \zeta_0$
(6) do {
(7) $\zeta_0 = \zeta$
(8) $H' = \overline{\zeta^{-1}}(\Delta) \cup H'$
(9) $\zeta = \text{Map}(H', H^*)$
(10) }
(11) while$(\zeta \neq \zeta_0)$
(12) return H'

Figure 2: Basic Structure of summary-based analysis

The *SumAnalyze* algorithm takes in as input a code-snippet S, abstract heap H and pointer analysis A. Line 1 constructs a normalized heap H^*, representing the unknown state of the heap before executing S, and line 2 analyses S without any info about the point-to information before S. Line 3 generates a polymorphic points-to summary Δ which includes the side-effects of S. Hence lines 1-3 correspond to *summary generation.*

Lines 4-11 instantiate the summary by computing the context-specific mapping ζ_0 and adding all the points-to edges representing S's side effects. The algorithm maps each node in the summary to a set of nodes at the call site using the mapping $\overline{\zeta^{-1}}$ and adds edges to the initial abstract heap H. As new edges are added to H, mapping ζ has to be recomputed since locations used in the summary can point to new additional

locations after the summary has been applied.

3.3 Formalization of Algorithm

$$
\begin{array}{ll}
\text{(Field selector)} & \eta : f \mid \eta.f \mid \eta^{\star} \\
\text{(Heap obj)} & o : a_i.\eta \mid \text{alloc}(T)@\boldsymbol{\rho} \\
\text{(Abstract loc)} & \pi : o \mid a_i \mid v_i@\boldsymbol{\rho} \\
\text{(Pts set)} & \theta : o \rightarrow \phi \\
\text{(Abstract heap)} & \Gamma : (\pi \times f) \rightarrow \theta \\
\text{(Summaries)} & \Upsilon : (T \times M) \rightarrow \Gamma
\end{array}
$$

Figure 3: Abstract domains used in the Analysis

Figure 3 shows the various notations used while describing the algorithm. Since the analysis is bottom-up, two kinds of heap objects o are used: Access paths of the form $a_i.\eta$ represent caller-allocated unknown heap objects reachable through the i'th argument, and objects named $alloc(T)\rho$ represent heap objects of type T allocated either in the currently analyzed method or transitive callee. Calling context is represented using a sequence of program points $\rho_1...\rho_n$, e.g. $alloc(T)\rho_1\rho_2$ corresponds to a heap object allocated at program point ρ_2 of a method m invoked at call site ρ_1. An *abstract heap* Γ maps each field f of a location π to a points-to set. A guarded points-to set θ is a set of pairs (o, ϕ) where o is a heap object and ϕ is a constraint given by $\phi = True|False|type(t)==T|\phi_1 \wedge \phi_2|\phi_1 \vee \phi_2$. An environment Υ maps each method M in class T to its corresponding summary, which is an abstract heap Γ summarizing M 's side effects.

$$
(1)\ \frac{\Gamma' = \Gamma[(v_1, \epsilon) \leftarrow \Gamma[v_2, \epsilon]]}{\Upsilon, \Gamma \vdash v_1 = v_2 : \Gamma'} \qquad (2)\ \frac{\Gamma' = \Gamma[v \leftarrow \{(\text{alloc}(T)@\rho, \top)\}]}{\Upsilon, \Gamma \vdash v = \text{new}^{\rho}\ T : \Gamma'}
$$

$$
(3)\ \frac{\begin{array}{c}\theta = \Gamma[v_2, \epsilon] \\ \Gamma' = \Gamma[(v_1, \epsilon) \leftarrow \Gamma[\theta, f]]\end{array}}{\Upsilon, \Gamma \vdash v_1 = v_2.f : \Gamma'}
$$

$$
(4)\ \frac{\begin{array}{c}\theta_1 = \Gamma[v_1, \epsilon] \quad \theta_2 = \Gamma[v_2, \epsilon] \\ \Gamma' = \Gamma[(o_i, f) \leftarrow (\Gamma(o_i, f) \sqcup (\theta_2 \downarrow \phi_i)) \mid (o_i, \phi_i) \in \theta_1]\end{array}}{\Upsilon, \Gamma \vdash v_1.f = v_2 : \Gamma'}
$$

$$
(5)\ \frac{\begin{array}{c}\Upsilon, \Gamma \vdash I_1 : \Gamma_1 \\ \Upsilon, \Gamma \vdash I_2 : \Gamma_2\end{array}}{\Upsilon, \Gamma \vdash \text{if}(*)\ I_1\ \text{else}\ I_2 : \Gamma_1 \sqcup \Gamma_2} \qquad (6)\ \frac{\begin{array}{c}\Upsilon, \Gamma \vdash I_1 : \Gamma_1 \\ \Upsilon, \Gamma_1 \vdash I_2 : \Gamma_2\end{array}}{\Upsilon, \Gamma \vdash I_1; I_2 : \Gamma_2}
$$

Figure 4: Rules for intra-procedural analysis

Figure 4 describes intra-procedural analysis using the form $\upsilon, \Gamma| - I : \Gamma'$ indicating that if statement I is executed in summary environment Υ and abstract heap Γ, new heap Γ' is obtained, distinguished from Γ since the analysis is (partially) flow-sensitive. Strong updates to variables are applied in rules 1,2,3 while rule 4 does a weak update.

The following rules deal with instantiating summaries at call sites. Rules in Figure 5 describe the instantiation of memory locations, constructing ζ^{-1} for Section 3.2. They produce judgements of the form

$M, \Gamma, \rho| - \mathit{inst\_loc}(\pi) : \theta$ where M maps formals-to-actuals and under M, Γ, ρ, location π used in the summary maps to (guarded) location set θ.

$$\frac{}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(a_i) : \{\mathcal{M}(a_i), \top\}} \quad \frac{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi) : \theta}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi.f) : \mathit{\Gamma}[\theta, f]}$$

$$\frac{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi) : \theta_0 \quad \theta_i = \bigsqcup_{1 \le j \le n} \mathit{\Gamma}[\theta_{i-1}, f_j]}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi.(f_1 ... f_n)^\star : \bigsqcup_{i \ge 0} \theta_i)} \quad \frac{\boldsymbol{\rho}_{\text{new}} = \text{new\_ctx}(\rho, \boldsymbol{\rho})}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(v@\boldsymbol{\rho}) : \{(v@\boldsymbol{\rho}_{\text{new}}, \top)\}}$$

$$\frac{\boldsymbol{\rho}_{\text{new}} = \text{new\_ctx}(\rho, \boldsymbol{\rho})}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\text{alloc}(T)@\boldsymbol{\rho}) : \{(\text{alloc}(T)@\boldsymbol{\rho}_{\text{new}}, \top)\}}$$

Figure 5: Rules for instantiating memory locations

$$\frac{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\mathit{lift}^{-1}(t)) : \theta \quad \phi = \mathit{has\_type}(\theta, T)}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\text{type}(t) = T) : \phi} \quad \frac{\star \in \{\wedge, \vee\} \quad \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_1) : \phi_1' \quad \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_2) : \phi_2'}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_1 \star \phi_2) : \phi_1' \star \phi_2'}$$

Figure 6: Rules for instantiating constraints

$$\frac{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi_1) : \theta_1 \ldots \mathit{inst\_loc}(\pi_n) : \theta_n \quad \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst}_\phi(\phi_1) : \phi_1' \ldots \mathit{inst}_\phi(\pi_n) : \phi_n'}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_pts}(\{(\pi_1, \phi_1), \ldots, (\pi_n, \phi_n)\}) : \sqcup_i (\theta_i \downarrow \phi_i)}$$

$$\frac{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_loc}(\pi) : \theta' \quad \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_pts}(\theta) : \theta'' \quad \mathit{\Delta} = [(\pi_i, f) \leftarrow (\theta'' \downarrow \phi_i) \mid (\pi_i, \phi_i) \in \theta']}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_partial\_heap}(\pi, f, \theta) : \mathit{\Delta}} \quad \frac{\begin{array}{c}\mathit{\Delta} = \{(\pi_1, f_{11}) \mapsto \theta_{11}, \ldots, (\pi_n, f_{nk}) \mapsto \theta_{nk}\} \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_partial\_heap}(\pi_1, f_{11}, \theta_{11}) : \mathit{\Delta}_{11} \\ \ldots \\ \mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_partial\_heap}(\pi_n, f_{nk}, \theta_{nk}) : \mathit{\Delta}_{nk}\end{array}}{\mathcal{M}, \mathit{\Gamma}, \rho \vdash \mathit{inst\_heap}(\mathit{\Delta}) : \sqcup_{ij} \mathit{\Delta}_{ij}}$$

Figure 7: Rules for instantiating summary

Figure 7 shows how to instantiate an abstract heap Δ. Given location π_i and field f_j from callee heap, *inst_partial_heap* instantiates all points-to edges from (π_i, f_j) and yields partial heap Δ_{ij}. Union over all Δ_{ij} gives Δ.

Figure 8 describes the analysis of method calls. For a static call to method m with summary Δ, a formal-to-actual mapping M is constructed and perform a least fixed-point computation that instantiates Δ till an over-approximation of m's side effects Δ' is obtained. For virtual calls, the call's targets are over approximated and using the previous static method call rule, heap Γ_i is obtained for some method $T_i :: m$. The final abstract heap after the call is obtained by union after applying type constraint over each heap.

$$(1)\ \frac{\begin{array}{c}\Upsilon(T,m) = \Delta \\ \mathcal{M} = [a_1 \mapsto v_1, \ldots, a_n \mapsto v_n] \\ \mathcal{M}, \Gamma \sqcup \Delta', \rho \vdash \mathit{inst\_heap}(\Delta) : \Delta'\end{array}}{\Upsilon, \Gamma \vdash m^{\rho}@T(v_1, \ldots, v_n) : \Gamma \sqcup \Delta'} \quad (2)\ \frac{\begin{array}{c}\text{static\_type}(v_0) = T \quad T_1 <: T, \ldots, T_n <: T \\ \phi_i = \text{has\_type}(\Gamma(v_0), T_i) \\ \Upsilon, \Gamma \vdash m^{\rho}@T_1(v_0, \ldots, v_k) : \Gamma_1 \\ \ldots \\ \Upsilon, \Gamma \vdash m^{\rho}@T_n(v_0, \ldots, v_k) : \Gamma_n\end{array}}{\Upsilon, \Gamma \vdash v_0.m^{\rho}(v_1, \ldots, v_k) : \sqcup_i(\Gamma_i \downarrow \phi_i)}$$

Figure 8: Analysis of Method calls

4 Example

This section demonstrates the approach on an example with virtual method calls and the use of context-sensitivity.

```
class X {
   Z f; Z g;
   void bar(Z z) {
      this.f = z;
   }
}

class Y extends X {
    void bar (Z z) {
        this.g = z;
    }
}

class A {
   X x; X y;

   void a1() {
      y = new Y();
      x = y;
      Z z = new Z();
      foo(z);
   }

   void a2() {
      x = new X();
      y = x;
      Z z = new Z();
      foo(z);
   }

   void foo(Z a) {
      x.bar(a);
      y.bar(a);
   }
}
```

Figure 9: Code example for demonstrating analysis

Consider the code in Figure 9, which defines classes X, Y and A. Suppose we want to check whether $x.f$ and $y.g$ are aliases at the end of $A :: a1$ and $A :: a2$.

First, we analyze the leaf procedures $X :: bar$ and $Y :: bar$. The summaries are represented in the form of a normalized heap. arg_0 is used to represent the *this* pointer, and arg_1 to denote the first formal parameter and so on. $arg_i.\epsilon$ denotes the heap objects pointed by arg_i on method entry.

In method $X :: bar$, we use the rule 4 in Figure 4 for the field-assignment statement and add a points-to edge via field f to the objects pointed to by z which is denoted by $arg_1.\epsilon$ in the summary in Figure 10. Similarly, in method $Y :: bar$, field g adds an edge to z denoted by $arg_1.\epsilon$. Note that these summaries encode the heap without any caller-info and need to be instantiated with the heap information in the caller. Also, at the call site of $X :: bar$, parameter z may have multiple points-to targets, hence a single edge in $X :: bar$'s summary can introduce multiple point-to edges at the call site.

$arg_0.\epsilon$	
f	($arg_1.\epsilon$, T)
g	

Summary for X::bar

$arg_0.\epsilon$	
f	
g	($arg_1.\epsilon$, T)

Summary for Y::bar

Figure 10: Method summaries for X::bar, Y::bar

Now consider the method $A :: foo$. Since x has type X, the virtual method call $x.bar(a)$ can invoke either $X :: bar$ or $Y :: bar$. For this, we refer to rule 2 in Figure 8. We instantiate the summaries of both $X :: bar$ and $Y :: bar$, with the heap instantiated from $T :: bar$ enforced by a constraint $type(x) == T$.

$arg_0.\epsilon.x$	
f	($arg_1.\epsilon$, type($arg_0.\epsilon.x$)==X)
g	($arg_1.\epsilon$, type($arg_0.\epsilon.x$)==Y)

$arg_0.\epsilon.y$	
f	($arg_1.\epsilon$, type($arg_0.\epsilon.y$)==X)
g	($arg_1.\epsilon$, type($arg_0.\epsilon.y$)==Y)

Figure 11: Method summary for A::foo

The virtual method call rule uses the static method rule (rule 1 in Figure 8) for each type. Instantiating the method $X :: bar$, first, the summary $\Delta = \Upsilon(X, bar)$ is fetched, then the mapping $M = [arg_0.\epsilon \rightarrow arg_0.\epsilon.x, arg_1.\epsilon \rightarrow arg_1.\epsilon]$ is created. In M, the left-side indicates the formal argument in the method summary, and the right are the actuals at the call site. Instantiating the summary for $X :: bar$ for this call site, by following the rules in Figure 7, an edge is added from field f in $arg_0.\epsilon.x$ to $arg_1.\epsilon$. As mentioned before, this edge is qualified by the constraint $type(arg_0.\epsilon.x) == X$ shown in Figure 11. Similarly, edge from summary of $Y :: bar$ is added from field g of $arg_0.\epsilon.x$ to $arg_1.\epsilon$ with the constraint $type(arg_0.\epsilon.x) == Y$.

Continuing with the method call $y.bar(a)$ in the second line of $A :: foo$, the summaries of both $X :: bar$ and $Y :: bar$ are instantiated again since the type of y is X. However, the formal-to-actual mapping M is changed, $arg_0.\epsilon$ in *X::bar* and *Y::bar* now refers to *this.y* denoted by $arg_0.\epsilon.y$ in method *foo*. Hence, as shown in Figure 11, the summary for *foo* includes points-to edges from fields f and g for $arg_0.\epsilon.y$ to $arg_1.\epsilon$ with the appropriate type constraints.

Note that this approach is context-sensitive since two different invocation of *bar* results in different points-to edges.

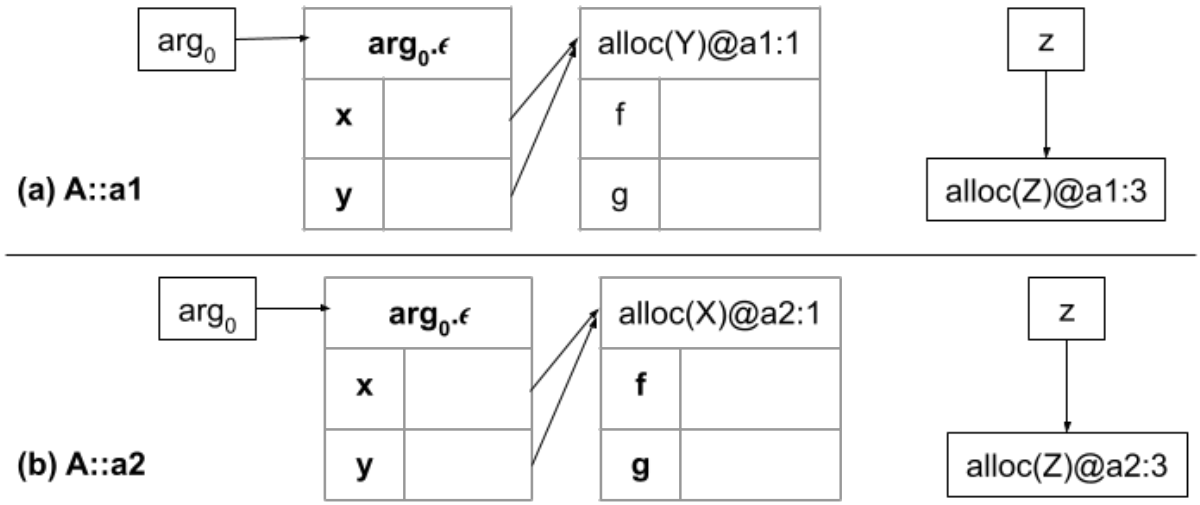

Figure 12: Points-to facts before *foo(z)* in A::a1, A::a2

Now lets consider method $A :: a1$ in Figure 9. First we perform the intra-procedural analysis using rules in Figure 4 for the statements before the call *foo(z)*. y points to location $alloc(Y)@a1 : 1$, indicating an object of type Y with context *a1:1* for first line in method *a1*. Instruction $x = y$ applies strong-update and both the x and y fields of $arg_0.\epsilon$ points to $alloc(Y)@a1 : 1$, local z points to $alloc(Z)@a1 : 3$ as ca be seen in Figure 12.

Now coming to method call *foo(z)*, in the formal-to-actual mapping M here, $arg_0.\epsilon.x$ and $arg_0.\epsilon.y$ in *foo*'s summary refer to *this.x* and *this.y*, and $arg_1.\epsilon$ refers to local z which points to $alloc(Z)@a1 : 3$. Again following the rules in Figure 8, instantiating *A::foo* and following the points-to edges of this.x and this.y in Figure 12, $arg_0.\epsilon.x$ and $arg_0.\epsilon.y$ both point to $alloc(Y)@a1 : 1$. Since the type of allocation is Y, the constraints $type(arg_0.\epsilon.x) == X$ and $type(arg_0.\epsilon.y) == X$ are false, while $type(arg_0.\epsilon.x) == Y$ and $type(arg_0.\epsilon.y) == Y$ are true. Hence, only field g in $alloc(Y)@a1 : 1$ has a points-to edge to $arg_1.\epsilon$ which is $alloc(Z)@a1 : 3$.

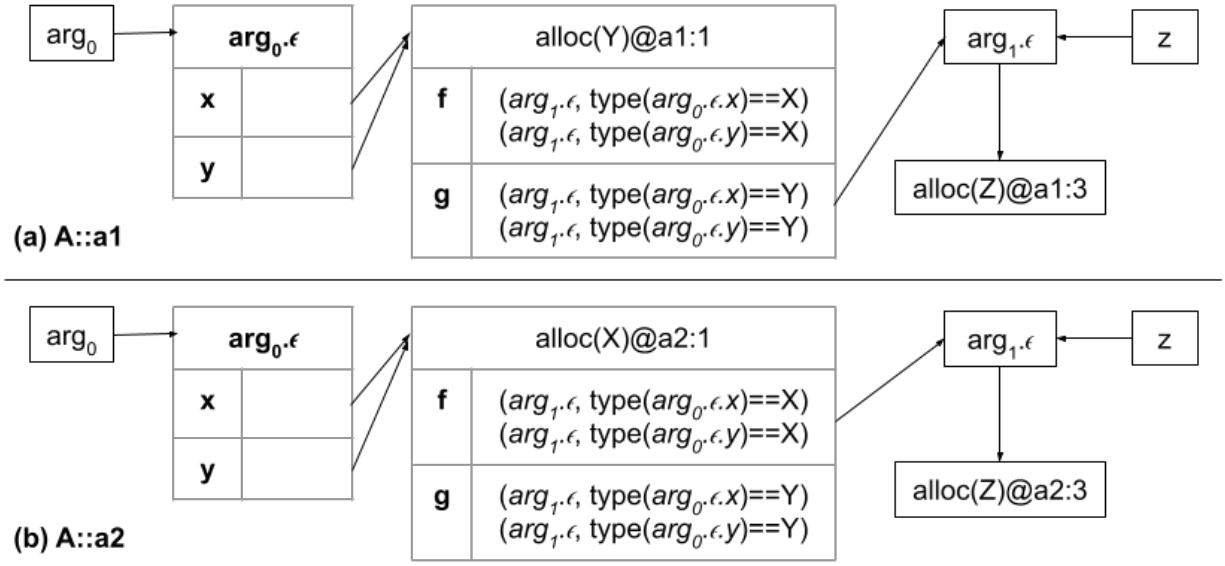

Figure 13: Points-to facts at the end in A::a1, A::a2

Similarly, for method $A :: a2$ in Figure 9, before the call $foo(z)$, both fields x and y point to $alloc(X)@a2 :$

1. Since the allocation type is X, the constraints for $type(arg_0.\epsilon.x) == X$ and $type(arg_0.\epsilon.y) == X$ succeed and the points-to edges at the end only has edge for field f to $alloc(Z)@a2 : 3$.

As can be seen in Figure 13, $x.f$ and $y.g$ are not aliases at the end of both $A :: a1$ and $A :: a2$. Note that an analysis that is either context-insensitive or uses an imprecise callgraph would result in may-alias. And even though there are 2 different calls to *foo* and 4 different calls to *bar*, each method is analyzed only once.

5 Future Work

To extend this analysis for **MHP analysis** (May-Happen-in-Parallel), we need to add the related constructs including *async*, $finish$, *synchronized*, etc. This will update the core language semantics considered during the analysis and also have an updated abstract domain which will abstract notify and wait calls. We will work on extending this context-sensitive analysis to utilize the MHP information to find method calls which could potentially happen in parallel along with other method calls during the inter-procedural analysis, and include all the possible side-effects.

6 Extending to Parallel Programs

The paper[1] which is elaborated previously presents an algorithm for sequential programs to perform context-sensitive pointer analysis. However, utilizing the above algorithm to analyse parallel programs is a challenging task, due to the inherent randomness associated with them. The main difficulty lies in ensuring that the pointer analysis remains conservative and considers all the possibilities due to interleaving of instructions across threads while still trying to keep the results as precise as possible.

We now try to extend the concepts used in the paper to analyse parallel programs in Java. First we state some assumptions used, and then describe the extensions to the algorithm. Finally, an example is used to demonstrate the working, along with some small examples later on to describe why we chose a particular procedure. Finally, some thoughts on possible future work.

6.1 Assumptions

We assume that the May-Happen-in-Parallel (MHP) information is already given to us. The statements are represented in the form of nodes, where each node can be a single statement, function call, or multiple statements as well. These nodes are similar to the nodes in the PEG graph mentioned by Feng[2] except the notify edges. We consider each node to be a single statement. Apart from the normal Java statements, parallelism constructs such as thread *start(), join()*, *wait-notify, synchronized* statements are also considered as described by Feng[2] as nodes of the PEG. The MHP information for each node is the set of nodes which may run in parallel with the current node. Each node is a set of statements, all of which may run in parallel with any statements present in the MHP nodes of the current node. The MHP analysis is conservative and over-approximates the possibilities.

This information could have been generated from an MHP generation algorithm like [2], which itself is based on an underlying pointer analysis. The MHP analysis handles the parallelism constructs, and hence using that information we need not worry about the individual parallel construct.

Method calls which include parallelism constructs are assumed to be in-lined, while normal method calls are not in-lined, and there can be multiple calls in a single node.

7 Algorithm for Parallel Programs

First, we define an additional update rule for fields which performs strong updates for the fields. This is in contrast to Rule 4 in Figure 4 which performs weak updates on fields. The additional rule is mentioned in the figure 14.

In the original algorithm, a single method summary was used which did weak updates for fields. Here, two different summaries are defined per method, one for strong updates and another for weak updates. These are used to generate the possible inter-leavings from different nodes running in parallel. The resulting heaps from strong and weak updates are denoted as H_s and H_w respectively.

$$\frac{\theta_1 = \Gamma[v_1.\epsilon], \theta_2 = \Gamma[v_2.\epsilon] \quad \Gamma' = \Gamma[(o_i, f) \leftarrow (\theta_2 \downarrow \phi_i) | (o_i, \phi_i) \in \theta_1]}{\Upsilon, \Gamma \vdash v_1.f = v_2 : \Gamma'}$$

Figure 14: Strong field update

The paper[1] didn't perform strong updates for the fields since the MHP information was not available, and the benchmarks utilized [3, 4] also consisted of parallel programs. Since we assume that the MHP information is present, we can apply strong updates for fields as well and utilize it for precision, for eg. when it's known that a statement doesn't run in parallel with anything else, a strong update is valid.

Rest of the rules for the inter-procedural analysis (Figure 4) and instantiating summaries (Figure 7) remain as is.

We use a slightly modified union operation for combining abstract heaps from different nodes. The intention here is to improve precision by only adding field nodes and edges and not modifying the existing stack variables i.e *root(H)* since parallel threads don't share stack and can't modify the existing variables. We denote this union of abstract heaps ignoring the stack variables using the symbol $\cup_H$.

$$G = A \cup_H B \implies \{V(G) = V(A) \cup V(B) \setminus (root(A) \cup root(B)), E(G) = E(A) \cup E(B)\}$$

Algorithm 1 Fixed Basic structure for summary-based analysis for Parallel Programs

```
1:  procedure SumAnalyze(H, N, A)
2:      input: abstract heap H, node N, intra-procedural analysis A
3:      output: abstract heap
4:      H*_init = NormalForm(H)
5:      for each Node N' ∈ MHP(N) do
6:          H*_init = H*_init ∪_H WeakOutput(N')
7:      end for
8:      H'*_outS = StrongAnalyze(H*_init, N, A)
9:      H'_outS = StrongFixedPointCompute(H'*_outS, H*_init, H_init)
10:     return H'_outS
```

Algorithm 1 shows the basic structure of the summary-based pointer analysis for parallel programs. The algorithm *SumAnalyze* takes as input abstract heap H, a node N and the pointer analysis A.

Lines 4-6 generates the initial heap before analyzing the node N. We take the input heap and perform a union of the heap with the weak output heaps of all the nodes which may run in parallel with N, given by MHP(N). The output heaps with weak updates are used from the nodes to consider the inter-leavings of fields writes with the current node. Here, the $WeakOutput(N')$ returns the normal heap after applying weak updates from the statements in N'.

The $WeakOutput$ of node N is derived without considering the parallelism constructs in the given program. It is calculated using the pointer analysis mentioned in section 3. As it does not contain any parallelism information, it should not be confused with the weak heap which could potentially be a superset of all the points-to information of the given node considering all the parallelism. It just represents the local thread information and does not include any side effects from the concurrency. Our algorithm for calculating summary of each node in figure 1 includes the side effects due to parallel constructs by considering a union of $WeakOutput$ of all the MHP nodes of the given node.

Lines 8-9 perform the same work as Lines 2-11 from Figure 2, with some modifications for the strong updates. The analysis outputs H'^*_{outS}, a normalized heap with strong updates. We have separated Lines 3-11 from Figure 2 into *StrongFixedPointCompute* which is described in Algorithm 2, to generate the final abstract heap from the summary heap. The final abstract heap is returned as a precise intra-procedural points-to analysis, to be used as input for its successors.

The nodes have successors and predecessors, wherein the predecessors are nodes from where the control flows into the current node. Similarly, control goes from the current node to its successors, and as expected a node cannot run in parallel with either its successors or predecessors. The union of strong output heaps from the predecessors is taken as input for the current node.

Predecessor for the first node of a thread is the corresponding *.start()* call, and one of the predecessor for *thread.join()* node is the corresponding thread's last or ending node.

Algorithm 2 Fixed point computation for strong updates

1: **procedure** StrongFixedPointCompute(H'^*, H^*, H)
2: **input**: analysed normal heap H'^*, non-analysed normal heap H^*, initial abstract heap H
3: **output**: analysed abstract heap
4: $\Delta = H'^*$
5: $\zeta_0 = \text{Map}(H, H^*)$
6: let $H' = \emptyset$; $\zeta = \zeta_0$
7: **do**
8: $\zeta_0 = \zeta$
9: $H' = \overline{\zeta^{-1}}\ (\Delta) \cup H'$
10: $\zeta = \text{Map}(H', H^*)$
11: **while** $\zeta \neq \zeta_0$
12: **return** H'

For the strong heap updates, we start with an empty abstract heap H' in contrast to H (Line 6 in Algorithm 2, since we don't want to carry forward all the sets already present in the initial heap. We also set $\Delta = H'^*$ as the set of edges to be added, as compared to Line 4 in Algorithm 2 where only the new edges are taken as the side-effects. The rest of the fixed-point computation is the same as Lines 6-11 in Figure 2, where it maps each edge in the summary to edges in the initial abstract heap using the mapping $\overline{\zeta^{-1}}$ and adds the new edges to H'.

8 Example

We now demonstrate the working of our approach using some examples. The complete code isn't shown below for brevity.

```
1
2  Thread T1:
3
4  class Thread1 {
5      void run (A a) {
6          /* Node N11 */ a.f.foo();
7          /* Node N12 */ a.f = new W();
8      }
9  }
10
11
12 Thread T2:
13
14 class Thread2 {
15     void run (A a) {
16         /* Node N21 */ a.f = new Y();
17         /* Node N22 */ a.f = new Z();
18     }
19 }
20
21
22 Main Thread:
23
24 Main() {
25     /* Node N0 */ a.f = new X();
26
27     t1.start(a);
```

```
28      t2.start(a);
29
30      b = a.f
31
32      /* Node N3 */ t1.join();
33      /* Node N4 */ t2.join();
34  }
35
36
37  // Classes:
38
39  class X,Y,Z,W extends Base {
40      Base g;
41
42      void foo() {
43          this.g = new Base();
44      }
45  }
46
47  class A {
48      Base f;
49  }
```

There are 2 threads T1, T2 which run in parallel. Both take as argument the same object a which the threads modify. We try to obtain the possible points-to information during the execution of each thread, and after the threads have joined.

MHP Info	
N11	N21, N22
N12	N21, N22
N21	N11, N12, N3
N22	N11, N12, N3
N3	N21, N22
N4	$\emptyset$

Table 1: MHP Information for all the nodes

Output of N0 is the input to both threads i.e. the first nodes N11 and N21. Therefore, *a.f* which will be $arg_1.\epsilon.f$ for both N11, N21 is *alloc(X)@n0*.

In Thread 2, we set *a.f* to new objects *alloc(Y)@n21* and then *alloc(Z)@n22*. Considering strong updates, the output of N22 will be *alloc(Z)@n22* only since the last write is used, and for weak updates, all the values including the initial value is possible.

//

Table 2: N21 heap information

$arg_1.\epsilon$	
f	(alloc(X)@n0, T)

Table 3: Input Heap for N21

$arg_1.\epsilon$	
f	(alloc(Y)@n21, T)

Table 4: Strong output heap for N21

$arg_1.\epsilon$	
f	(alloc(X)@n0, T), (alloc(Y)@n21, T)

Table 5: Weak output heap for N21

In N1, we first call *foo()* on *a.f*, and then set *a.f* to *alloc(W)@n12*. Considering just single-threaded execution, only *X::foo()* will need to be analysed, and the next write will change the *a.f* field.

Now let's consider the possible inter-leavings of N11 with N21 & N22. In N21, *a.f = new Y();* can happen first and the context-switch to N11 occurs, hence *a.f.foo()* can be called on an object of type Y, and *Y::foo()* needs to be analysed. Similarly, *a.f* can be *alloc(Z)@n22*, therefore *Z::foo()* must be considered as well.

Table 6: N22 heap information

$arg_1.\epsilon$	
f	(alloc(Z)@n22, T)

Table 7: Strong output heap for N22

$arg_1.\epsilon$	
f	(alloc(X)@n0, T), (alloc(Y)@n21, T), (alloc(Z)@n22, T)

Table 8: Weak output heap for N22

Running the algorithm for N11, we take the union of weak output heap of N21 & N22 as shown in Tables 5 & 8, which results in the same heap as N22's weak heap. Using that as input, we run the analysis on N11's statements which is just instantiating the strong summary and effects of *foo()* call are added with the appropriate type constraints on *a.f.*

$arg_1.\epsilon$	
f	(alloc(W)@n12, T)

Table 9: Strong output heap for N12

Figure 15: N11 heap information

$arg_1.\epsilon$	
f	(alloc(X)@n0, T)

alloc(X)@n0	
g	(alloc(Base)@X::foo:1, $type(arg_0.\epsilon.f) == X$)

Figure 16: Initial output heap for N11

alloc(X)@n0	
g	(alloc(Base)@X::foo:1, $type(arg_0.\epsilon.f) == X$)

alloc(Y)@n21	
g	(alloc(Base)@Y::foo:1, $type(arg_0.\epsilon.f) == Y$)

alloc(Z)@n22	
g	(alloc(Base)@Z::foo:1, $type(arg_0.\epsilon.f) == Z$)

Figure 17: Final N11 output heap

The strong output heap of N22 remains as is, and hence is not shown again. Similarly, in N21 & N22, we would take the union of the output weak heap of N11 & N12 but this new initial heap doesn't cause any effects on the strong output heap on N21 & N22. Note that the heap objects given in table 17 will be present in all of these.

While analysing thread join node, the input heap is calculated in a special manner unlike just taking union of the strong heaps of the predecessors. We consider a special edge called "join" edge connecting the end node of thread CFG to the corresponding join node. Hence a join node will have always have a join predecessor other than local predecessors. To create the input heap, we take a union of all local predecessors similar to any node and then overwrite the heap of join predecessor. This heap goes as input heap H for the SUMANALYZE in algorithm 1.

Now coming to Nodes N3, N4 which are the thread join statements, N3 can run in parallel with N21, N22 only, and the predecessor of N3 is N1. Hence, the input heap for N3 will be the strong output heap of N12 as in Figure 9, and after taking the union with weak heap of N2, the final heap is as below. N4 will take the union of strong heaps of both threads.

a	
f	(alloc(X)@n0, T), (alloc(Y)@n21, T), (alloc(Z)@n22, T), (alloc(W)@n12, T)

Table 10: Output heap for N3

a	
f	(alloc(Z)@n22, T), (alloc(W)@n12, T)

Table 11: Output heap for N4

9 Future Work

One major drawback of the current approach is the assumption that MHP information is already available. However, the MHP analysis is generally dependant on an underlying pointer analysis, if that analysis is too imprecise, the MHP information will also be imprecise and cause an adverse effect on this analysis as well since the MHP nodes will be grossly over-approximated and reduce the effectiveness of all the stoes to improve precision. Since Pointer and MHP analysis are inter-dependent, the two analysis can be run one after the other, with each iteration utilizing the previous results and improving the precision.

Another is handling of multiple statements inside a single node. Currently, we consider each node to consist a single statement, this however increases the number of nodes, and hence requires much more computation & memory due to each node storing many MHP nodes and output heap as well. The current algorithm doesn't work due to the strong updates being applied. For e.g. consider that a new statement *a.f.foo()* is added after N12, and we consider all N11, N12 and the new N13 as node N1. Similarly, N21 & N22 is taken as node N2. The strong output of N12 will remove all the objects added by N2 as well, and now input of N13 is as Table 9. However, this is incorrect since N2 runs in parallel with N1 and *a.f* can be (alloc(Y)@n21, T) or (alloc(Z)@n22, T) as well. The strong update needs to be modified to somehow only replace objects which appear from predecessors and not the MHP nodes. Note that just keeping objects from MHP nodes doesn't work, since the parallel node can add an object which was created in a predecessor node such as the main thread. Properly handling the context of each edge in the graph and which node introduces it could be a possibility.

The modified algorithm is also much more time-consuming and memory-intensive, due to the single statement nodes as described above, and also the union of multiple heaps for creating the input heap. We considered an alternative of not taking the union of heaps but whenever a lookup in the heap is required, we do a union of the output from individual heaps. This however gave incorrect answers as can be seen from an example below -

Main Thread: a = R0; a.f = R1;
Thread 1: x = a.f;
Thread 2: a.f = R2; a = R4; a.f = R5;

a is passed as a parameter to both threads, hence its a stack variable and reassigning *a* in any thread doesn't affect the other. For Thread 1, lookup for *a.f* with union of sets - Main[a.f] $\cup$ Thread2[a.f] gives [R1, R5] which is wrong. Note that here all statements are considered to be a single node. One statement per node as we have used fixes this problem, but we haven't confirmed whether this is conservative and doesn't exclude any possibilities.

The underlying memory consistency model is also an assumption which has been taken here. We have assumed that writes from a node are immediately visible to the other threads and in the same order as the program order. The program order assumption allows for strong updates, however if writes can be visible in out-of-order fashion, then in order to be conservative, only weak updates must be applied since older entries are still applicable. But if non-synchronized data accesses are being done, then anything is possible.

References

[1] Y. Feng, X. Wang, I. Dillig, and T. Dillig, "Bottom-up context-sensitive pointer analysis for java," pp. 465–484, 11 2015.

[2] G. Naumovich, G. Avrunin, and L. Clarke, "An efficient algorithm for computing mhp information for concurrent java programs," *ACM SIGSOFT Software Engineering Notes*, vol. 24, 07 1999.

[3] "Ashes benchmark suite." http://www.sable.mcgill.ca/software/.

[4] "Dacapo benchmarks." http://www.dacapobench.org/.