Bottom-up Context-Sensitive Pointer Analysis for Java

This note is the summary of the paper: Bottom-up Context-Sensitive Pointer Analysis for Java.

Introduction

We can use `top-down` or `bottom-up` method for context-sensitivity.

A `top-down` method starts at entry methods of a program and analyzes callers before callees. It knows the context when analyzing each method, but it needs to re-analyze the same method multiple times under different contexts.

A `bottom-up` method starts at leaf methods of a program and analyzes callees before callers. It can generate a summary that can be used in any calling context to get context-sensitive results.

This paper presents a bottom-up context- and field-sensitive pointer analysis for Java. A key novel feature of our approach is the constraint-based treatment of virtual method calls.

Concepts

Abstract heap

An abstract heap H is a graph (N, E) where N is a set of nodes corresponding to `abstract memory locations` and E is a set of directed edges between nodes, which is labeled with field names or ϵ.

An edge (o_1, o_2, f) indicates that the f field of o_1 may point to o_2. An edge (o_1, o_2, ϵ) indicates that o_1(from stack) may point to o_2.

An `abstract memory location` represents either the stack location of a variable or a set of heap objects.

The root nodes of an abstract heap H(that is $root(H)$) denote locations of variables.

Given abstract heap H_1 and H_2, $H_1 \cup H_2$ represents the abstract heap containing nodes and edges from both H_1 and H_2.

Normalization

Given an abstract heap H, we can define a normalization operation $N(H)$ to get a normalized heap $H^* = (N^*, E^*)$ and a map $\zeta : N \to N^*$ such that

- If $x \in root(H)$, then $x \in X^*$ and $\zeta(x) = \{x\}$.
- If $(o, o', f) \in E$ and $o^* \in \zeta(o)$, then $o^*.f \in N^*$, $o^*.f \in \zeta(o')$ and $(o^*, o^*.f, f) \in E^*$.

```
o      zeta   o*
|             |
| f    ===>   | f
|             |
o'     zeta   o*.f
```

H^* corresponds to a generic heap representing the unknown points-to target of object o's f field as $o.f$.

For a method m, the normalization heap is the same regardless of different calling context.

Given a map ζ, we can define ζ^{-1} such that $n \in \zeta^{-1}(n^*)$ iff $n^* \in \zeta(n)$.

ζ^{-1} can instantiate a method summary to a particular abstract heap at a call site.

Default edge

An edge (n, n', f) is a default edge of an abstract heap H iff $n' = n.f$.

Given a heap H, we use $default(H)$ to represent the set of default edges in H.

Summary-based Pointer Analysis

Given code snippet S, an abstract heap H and pointer analysis A, we write $H' = Analyze(H, S, A)$ to show that: If statement S is executed in an environment that satisfies abstract heap H, it will get a concrete heap H' after using pointer analysis A to analyze S.

Formalization of Algorithm

Grammar

The grammar of the oop language is shown as figure 1

$$
\begin{array}{ll}
\text{Program } P & := C^+ \\
\text{ClassDecl } C & := \text{class } T_1 \ [\text{extends } T_2]? \ \{F^*; M^*\} \\
\text{FieldDecl } F & := T \ \text{fld\_name}; \\
\text{MethodDecl } M & := m(T_0 \ v_0, \ \ldots, \ T_k \ v_k) = \{V^*; I; \} \\
\text{VarDecl } V & := T \ \text{var\_name}; \\
\text{Instruction } I & := v_1 = v_2 \mid v_1 = v_2.f \mid v_1.f = v_2 \mid v = \text{new}^\rho \ T \\
 & \quad \mid \text{if}(*) \ I_1 \text{ else } I_2 \mid I_1; I_2 \mid m^\rho @T(v_1, \ldots, v_n) \mid v_0.m^\rho(v_1, \ldots, v_n)
\end{array}
$$

Figure 1: Grammar of oop language

Abstract Domains

Heap object o has two kinds:

- $a_i . \eta$ represents unknown heap objects reachable through the i'th argument.
- $alloc(T)@\rho$ represents heap objects of type T that are allocated either in the currently analyzed method or in a transitive callee.

Abstract memory location π is either heap object o or stack location(a_i denotes the stack location of the i'th argument and $v_i@\rho$ denotes the location of a local variable v_i under context ρ).

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

Figure 2: Abstract domains

Here, the calling contexts is represented by a sequence of program points $\rho_1, \rho_2, \ldots, \rho_n$, where ρ_i corresponds to some call or allocation site.

An environment Υ maps each method M in class T to its corresponding summary, which is a abstract heap Γ that summarizes M's side effects.

Argument-derived location

A location π is derived from an argument, written $arg(\pi)$, iff π is

- a_i represents the location of the i'th argument or
- a heap object represented with an access path $a_i . \eta$

An abstract heap Γ maps each field f of location π to a points-to set θ. θ is a set of pairs (o, ϕ) where o is a heap object and ϕ is a constraint.

Constraint ϕ is defined in Figure 3.

$$
\begin{array}{ll}
\text{Function } f & := \text{pts} \mid \text{alloc} \mid \varsigma_i \\
\text{Term } t & := c \mid v \mid f(\boldsymbol{t}) \\
\text{Formula } \phi & := \top \mid \bot \mid \text{type}(t) = T \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2
\end{array}
$$

Figure 3: Constraints

Here, terms include constant c, variables v and function applications $f(t)$.

Function f is point-to-set pts, or allocation $alloc$ or an n-ary function ζ_i.

Formulas are composed of true, false, type checking and their conjunctions and disjunctions.

The alloc and type functions obey the additional axiom $\forall x, type(alloc(T,\rho)) = T$.

We also define an operation called $lift(\pi)$, abbreviated $\overline{\pi}$, as follows:

- $\overline{a_i} = a_i$
- $\overline{alloc(T)@\rho} = alloc(T,\rho)$
- $\overline{\pi.\, f} = pts(\overline{\pi}, f)$
- $\overline{\pi.\,(f)^*} = \zeta_i(\overline{\pi}, f)$

Given a term t, $lift^{-1}(t)$ yields an abstract memory location representation of that term.

has_type(θ, T)

Given a points-to-set θ, the function $has\_type(\theta, T)$ yields the following constraint:

$$\cup_{(\pi_i,\phi_i)\in\theta}((type(\overline{\pi_i}) = T) \wedge \pi_i)$$

Operations on Abstract Domains

Default target

Given an argument-derived location π and a field f, the default target of $\pi.\, f$, written $def(\pi, f)$ is given as follows:

$$\mathrm{def}(\pi, f) = \begin{cases} \pi & \text{if } \pi = \pi'.(\boldsymbol{f})^\star \text{ and } f \in \boldsymbol{f} & (1) \\ \pi'.(f.\boldsymbol{g})^\star & \text{if } \pi = \pi'.f.\boldsymbol{g} & (2) \\ \pi.f & \text{otherwise} & (3) \end{cases}$$

It distinguish the case of recursive data structures(1 and 2) and non-recursive data structures(3).

This is used to ensure termination of the fixed-point computation performed by our algorithm.

Field look-up

Given heap Γ, field f, and location π, the field lookup operation $\Gamma[\pi, f]$ retrieves the points-to target for π's field f:

$$\mathit{\Gamma}[\pi, f] = \begin{cases} \mathit{\Gamma}(\pi, f) \cup \{(\mathrm{def}(\pi, f), \top)\} & \text{if } \arg(\pi) \\ \mathit{\Gamma}(\pi, f) & \text{otherwise} \end{cases}$$

If π is an argument derived location, we can add $def(\pi, f)$ as a default edge.

Join

Since the analysis is a weak update, we should merge two points-to sets using the following join operator:

$$(\theta_1 \sqcup \theta_2)(o) = \begin{cases} \theta_1(o) \vee \theta_2(o) & \text{if } o \in dom(\theta_1) \cap dom(\theta_2) \\ \theta_1(o) & \text{if } o \in dom(\theta_1) \text{ and } o \notin dom(\theta_2) \\ \theta_2(o) & \text{if } o \in dom(\theta_2) \text{ and } o \notin dom(\theta_1) \end{cases}$$

Note that $o \in dom(\theta_1)$ means that o is in the domain of θ_1.

Projection

Given a points-to set θ and constraint ϕ, we can conjoin ϕ with every constraint in θ:

$$\theta \downarrow \phi = \{(\pi_i, \phi_i \wedge \phi) \mid (\pi_i, \phi_i) \in \theta\}$$

Field lookup

Here is the field lookup operation for a points-to set:

$$\Gamma[\theta, f] = \cup_{(\pi_i, \phi_i) \in \theta} \Gamma[\pi_i, f] \downarrow \phi_i$$

So $\Gamma[\theta, f]$ includes the points-to target of every element in θ under the appropriate constraints.

Intraprocedural Analysis

The form $\Upsilon, \Gamma \vdash I : \Gamma'$ indicates that if statement I is executed in an environment that satisfies summary environment Υ and abstract heap Γ, then we can obtain a new heap Γ'.

Figure 4 shows the constraint rule of intraprocedural analysis:

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

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

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

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

Figure 4: Constraint rules for intraprocedural analysis

Rule (1) depicts **assignment** $v_1 = v_2$: It applies strong updates to variables and it updates the points-to set for (v_1, ϵ) to $\Gamma[v_2, \epsilon]$.

Rule (2) depicts **memory allocation** $v = new^\rho\ T$: It introduces a new abstract location named $alloc(T)@\rho$ and assigns v with it.

Rule (3) depicts **load** $v_1 = v_2.f$: It first looks up the point-to set θ of v_2 and then uses $\Gamma[\theta, f]$ to retrieve the targets of memory locations in θ. Finally, it override v_1's existing targets and change its points-to set to $\Gamma[\theta, f]$.

Rule (4) depicts **store** $v_1.f = v_2$: We apply only weak updates to heap objects, so we preserve the existing (o_i, f).

Interprocedural Analysis

Memory locations

Since a key part of summary instantiation is constructing the mapping from locations in the summary to those at the call site, we first start with the rules in Figure 5 which describe the instantiation of memory locations.

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

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

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

Figure 5: Instantiation of memory locations

The rules above produce judgment of the form $M, \Gamma, \rho \vdash \textit{inst\_loc}(\pi) : \theta$ where M maps formals to actuals, and Γ and ρ are the abstract heap and program point associated with a call site respectively. The form means that: Under M, Γ, ρ, location π used in the summary maps to location set θ.

Rule 1 maps formal parameter a_i to the actual $M(a_i)$.

Rule 2 instantiates argument-derived locations of the form $\pi.\ f$.

Rule 3 instantiates access paths f the form $\pi.\ (f_1, f_2, \ldots, f_n)^*$. It gets all locations that are reachable from θ_0 using any combination of field selectors $f_1, f_2, \ldots, f_n$.

Rule 4 and rule 5 describes the instantiation of allocations and local variables. $\textit{new\_ctx}$ gets a new context in a `k-limiting` way:

$$\text{new\_ctx}(\rho, \boldsymbol{\rho}) = \begin{cases} \rho, \boldsymbol{\rho} & \text{if } |\boldsymbol{\rho}| \le k \\ \boldsymbol{\rho} & \text{otherwise} \end{cases}$$

Here ρ is the current call site and ρ in bold is the current context of call string.

Constraints

The instantiation of constraints is summarized in Figure 6.

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

Figure 6: Instantiation of constraints

To solve a constraint $\textit{type}(t) = T$, we map t to its corresponding location set θ by using $\textit{inst\_loc}$ and leverages function $\textit{has\_type}$ to yield the condition.

Abstract Heap

Figure 7 shows how to instantiate an abstract heap Δ.

$$\frac{\begin{array}{c}\mathcal{M}, \Gamma, \rho \vdash \mathit{inst\_loc}(\pi_1) : \theta_1 \ldots \mathit{inst\_loc}(\pi_n) : \theta_n \\ \mathcal{M}, \Gamma, \rho \vdash \mathit{inst}_\phi(\phi_1) : \phi'_1 \ldots \mathit{inst}_\phi(\pi_n) : \phi'_n\end{array}}{\mathcal{M}, \Gamma, \rho \vdash \mathit{inst\_pts}(\{(\pi_1, \phi_1), \ldots, (\pi_n, \phi_n)\}) : \sqcup_i(\theta_i \downarrow \phi_i)}$$

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

Figure 7: Instantiation of abstract heap

Method Calls

Figure 8 shows the analysis of method calls.

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

Figure 8: Analysis of method calls