Stacking-based Context-Sensitive Points-to Analysis for Java

Xin Li and Mizuhito Ogawa

School of Information Science,
Japan Advanced Institute of Science and Technology, Nomi, Japan

Abstract. Points-to analysis for Java infers heap objects that a reference variable can point to. Existing practiced context-sensitive points-to analyses are cloning-based, with an inherent limit to handle recursive procedure calls and being hard to scale under deep cloning. This paper presents a *stacking-based* context-sensitive points-to analysis for Java, by deriving the analysis as weighted pushdown model checking problems. To generate a tractable model for model checking, instead of passing global variables as parameters along procedure calls and returns, we model the heap memory with a global data structure that stores and loads global references with synchronized points-to information on-demand. To accelerate the analysis, we propose a two-staged iterative procedure that combines *local exploration* for lightening most of iterations and *global update* for guaranteeing soundness. In particular, *summary transition rules* that carry cached data flows are carefully introduced to trigger each local exploration, which boosts the convergence with retaining the precision. Empirical studies show that, our analysis scales well to Java benchmarks of significant size, and achieved in average $2.5X$ speedup in the two-staged analysis framework.

1 Introduction

The notion of context-sensitivity bears a similarity to inline expansion, as if method calls are replaced with bodies of the callees. As such, the typical *cloning-based* program analysis [17] creates a separate copy of a method call within a bounded call depth or with collapsing recursive procedure calls. The cloning-based approach has an inherit limit to handle (recursive) procedure calls. An alternative to obtaining context-sensitivity in terms of *valid call paths* is to model the program's call stack with the pushdown stack. Since the stack can grow unboundedly, no restriction is placed on the call depth and recursions. By valid, it means a procedure call always returns to the most recent call site.

Points-to analysis (PTA) infers the set of heap objects that a reference variable may point to. PTA for Java is featured for being interdependent of call graph construction, due to dynamic language features like late binding. The long-standing challenge is to design a scalable yet precise PTA. Context-sensitivity is shown to be crucial to the precision of PTA for Java. To the best of our knowledge, existing practiced PTA for Java are all cloning-based [17, 13]. However,

empirical study recently shows that, more than one thousand of methods are typically contained within recursive procedure calls in practice [18], and approximating recursions potentially threatens the analysis precision [6].

This paper presents a *stacking-based* context-sensitive PTA for Java, by encoding the analysis as model checking problems on WPDSs [11]. Our analysis is context-sensitive, field-sensitive, and flow-insensitive, with the call graph constructed on-the-fly. In contrast to the cloning-based approach, there is a single copy for each procedure in the analysis, while calling contexts are entirely characterized as (regular) configurations over the pushdown stack. Our first step to scalability is that, instead of passing global variables explicitly as parameters along procedure calls and returns (that is hopeless to scale from our empirical study) [10], we model the heap memory with a global data structure during the analysis, which loads intermediate points-to information of global references, and stores cached values to global references on-demand when they are referred to inside procedures. This encoding dramatically reduces the number of pushdown transitions and generates a tractable model for model checking.

To further accelerate the analysis, we propose a two-staged iterative procedure, denoted by $(\texttt{LE} \circ \texttt{GU})^*$ as opposed to the traditional iterative procedure denoted by $\texttt{GU}^*$, which combines *local exploration* (LE) for lightening most of iterative cycles and *global update* (GU) for guaranteeing the completeness. Our insight is, to localize most of iterative cycles on the partial program models in LEs, and perform GUs on the entire program model as few times as possible. In particular, *summary transition rules* that carry previously computed data flows are introduced to effectively trigger each LE and boost the convergence. In effect, the computation of data flows to some program point in the partial program model is divided into independent phases via *frontiers*: the computation of data flows from the program entry to frontiers and the computation of data flows from frontiers to the concerned program point. By carefully adding summary transition rules to frontiers, the analysis by $(\texttt{LE} \circ \texttt{GU})^*$ retains the same precision as the analysis by $\texttt{GU}^*$. Empirical studies show that, a substantial speedup in practice can be achieved by the two-staged analysis.

This paper primarily makes the following contributions.

- We present a scalable stacking-based context-sensitive PTA for Java by model checking WPDSs, with no restriction on (recursive) procedure calls.
- We propose a two-staged iteration procedure, supported by carefully introducing summary transition rules, to effectively accelerate the analysis.
- We implemented the analysis algorithms as a tool named *Japot*. Empirical study shows that *Japot* scales well to Java benchmarks of significant size.

The rest of the paper is organized as follows. Section 2 briefly reviews weighted pushdown model checking. Section 3 presents Java semantics and abstractions. Detection of points-to information by model checking is in Section 4. Section 5 presents a two-staged iteration procedure. Section 6 gives experiments and Section 7 discusses related work. Section 8 concludes the paper.

2 Weighted Pushdown Model Checking

Definition 1. *Define a* ***pushdown system*** $P = (Q, \Gamma, \Delta, q_0, \omega_0)$*, where* Q *is a finite set of states called control locations,* Γ *is a finite stack alphabet, and* $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$ *is a finite set of transition rules.* $q_0 \in Q$ *and* $\omega_0 \in \Gamma^*$ *are the initial control location and stack contents respectively. A transition rule* $(p, \gamma, q, \omega) \in \Delta$ *is denoted by* $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$*. A* ***configuration*** *of* P *is a pair* $\langle q, \omega \rangle$ *for* $q \in Q$ *and* $\omega \in \Gamma^*$*.* Δ *defines a transition relation* $\Rightarrow$ *on configurations such that* $\langle p, \gamma\omega' \rangle \Rightarrow \langle q, \omega\omega' \rangle$*for each* $\omega' \in \Gamma^*$ *if* $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$*.*

Definition 2. $S = (D, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ *with* $\boldsymbol{0}, \boldsymbol{1} \in D$ *is a* ***bounded idempotent semiring*** *if*

1. $(D, \oplus)$ *is a commutative monoid with* $\boldsymbol{0}$ *as its unit element, and* $\oplus$ *is idempotent, i.e.,* $a \oplus a = a$ *for* $a \in D$*;*
2. $(D, \otimes)$ *is a monoid with* $\boldsymbol{1}$ *as the unit element;*
3. $\otimes$ *distributes over* $\oplus$*;*
4. $\forall a \in D, a \otimes \boldsymbol{0} = \boldsymbol{0} \otimes a = \boldsymbol{0}$*;*
5. *The partial ordering* $\sqsubseteq$ *is defined on* D *such that* $\forall a, b \in D, a \sqsubseteq b$ *iff* $a \oplus b = a$*.*

By Def. 2, it is guaranteed there are no infinite descending chains in D wrt $\sqsubseteq$, and $\mathbf{0}$ is the greatest element.

Definition 3. *Define a* ***weighted pushdown system*** *(WPDS)* $W = (P, S, f)$*, where* $P = (Q, \Gamma, \Delta, q_0, \omega_0)$ *is a pushdown system,* $S = (D, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ *is a bounded idempotent semiring, and* $f \colon \Delta \to D$ *is a weight assignment function.*

When encoding the program as a WPDS, the bounded idempotent semiring models program data flows. A weight element encodes traditional program transformers; $f \oplus g$ combines data flows at the meet of control flows; $f \otimes g$ composes sequential control flows; $\mathbf{1}$ is identity function, and $\mathbf{0}$ implies program errors.

Definition 4. *Given a weighted pushdown system* $W = (P, S, f)$*, where* $P = (Q, \Gamma, \Delta, q_0, w_0)$*. Assume* $\sigma = [r_0, ..., r_k]$ *for* $r_i \in \Delta (0 \leq i \leq k)$ *to be a sequence of pushdown transition rules, and* $v(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$*. Let* $\mathtt{path}(c,c')$ *be the set of all transition sequences that transform configurations from* c *into* c'*. Given sets of regular configurations* $C, C' \subseteq Q \times \Gamma^*$*, for each configuration* $c \in Q \times \Gamma^*$*,*

- *the Generalized Pushdown Successor (****GPS****) problem is to find* $\mathtt{gps}(c) = \bigoplus\{v(\sigma) \mid \sigma \in \mathtt{path}(c', c), c' \in C\}$*.*
- *the Generalized Pushdown Predecessor (****GPP****) problem is to find* $\mathtt{gpp}(c) = \bigoplus\{v(\sigma) \mid \sigma \in \mathtt{path}(c, c'), c' \in C'\}$*.*
- *The Meet-Over-All-Valid-Path (****MOVP****) problem is to find* $\mathtt{MOVP}(C, C', W) = \bigoplus\{v(\sigma) \mid \sigma \in \mathtt{path}(c, c'), c \in C, c' \in C'\}$*.*

Given $p \in Q, \gamma \in \Gamma$ *and* $c \in Q \times \Gamma^*$*, and let* $\mathtt{conf}(p, \gamma) = \{\langle p, \gamma\omega \rangle \mid \omega \in \Gamma^*\}$*, further define* $\widehat{\mathtt{MOVP}}(c, \langle p, \gamma \rangle, W) = \mathtt{MOVP}(\{c\}, \mathtt{conf}(p, \gamma), W)$*.*

Efficient algorithms for solving the GPS and GPP problems are proposed based on the fact that a regular set of configurations is closed under forward and backward reachability [11]. Then, MOVP is solved based on the results of either GPS or GPP. There are two off-the-shelf implementations of weighted pushdown model checking, Weighted PDS Library [1] and WPDS++ [3, 4]. We exploit the former as the back-end analysis engine.

Example 1. As shown in Table 1, a context-sensitive PTA is able to distinguish that, x_1 and x_2 points to objects created at line 2 and 3, respectively. In contrast, an imprecise analysis may mix them.

Table 1. A Java Code Snippet

0. public class Main {	
1. public static void main(String[] args){	8. public static Object f_1(Object a){
2. Object y_1 = new String();	9. return f_2(a);
3. Object y_2 = new Object();	10. }
4. Object x_1 = $f_1(y_1)$;	11. public static Object f_2(Object b){
5. Object x_2 = $f_1(y_2)$;	12. return b;
6. System.out.println(x_1.equals(x_2));	13. }
7. }	14. }

3 Semantics and Abstraction

3.1 Java Semantics on the Heap and Call Stack

Definition 5. *A method signature consists of method name, parameter types, and return type. We denote by $\mathcal{C}$ the set of classes, and denote by Ψ the set of method signatures. A method is identified by a pair of its enclosing class $C \in \mathcal{C}$ and its method signature $\psi \in \Psi$, denoted by $C.\psi$. The set of method identifiers is denoted by $\mathcal{C}.\Psi \subseteq \mathcal{C} \times \Psi$. We denote by Θ the class environment, including all classes, type representations of classes, and the type hierarchy.*

In Java, a heap object is a dynamically created instance of either a class or an array. Reference variables are typically local variables, method parameters, array references, and static or instance fields that hold reference types. Fields and array references can be regarded as global variables. A local variable v from its enclosing method $C.\psi$ is denoted by indexing with the scope as $v^{C.\psi}$. If $C.\psi$ is clear from the context, we often simply write v.

Definition 6. *The set of references is denote by $\mathcal{V}$, and the set of heap objects is denoted by $\mathcal{O}$. An abstract heap environment $\boldsymbol{henv}$ is a mapping, denoted by $\mapsto$, from $\mathcal{V}$ to $\mathcal{O}$. The set of abstract heap environments is denoted by Λ, on which the update operation $\odot$ is defined such that for $r, r' \in \mathcal{V}$, $o \in \mathcal{O}$, $(\boldsymbol{henv} \odot [r \mapsto o])r' = o$ if $r = r'$ and $(\boldsymbol{henv} \odot [r \mapsto o])r' = \boldsymbol{henv}(r')$ otherwise.*

[1] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/

Definition 7. *We denote by $\mathcal{L}$ the set of program line numbers and denote by $\mathcal{S}$ the set of program statements. Let $\texttt{Stmt} : \mathcal{L} \to \mathcal{S}$ be the function that returns the statement at a given line number. $\mathcal{S}_\epsilon \subseteq \mathcal{S}$ denotes the set of statements that do not contain explicit method invocations and operate on the heap memory, and by $\mathcal{S}_I \subseteq \mathcal{S}$ denotes the set of statements that contains explicit method invocations.*

Definition 8. *Let $\texttt{Elem} = \mathcal{L} \times (\mathcal{O} \cup \{*\}) \times \mathcal{C}.\Psi$. Let $\Pi = \texttt{Elem}^*.\{\bot\}$ be the set of calling histories over the call stack. Define $\texttt{push}(\texttt{stack}, \texttt{e}) = \texttt{e.stack}$ for $\texttt{stack} \in \Pi$; and $\texttt{pop}(\texttt{e.stack}) = \texttt{stack}$, $\texttt{top}(\texttt{e.stack}) = \texttt{e}$ for $\texttt{e} \in \texttt{Elem}$ and $\texttt{stack} \in \Pi$; and $\texttt{pop}(\bot) = \texttt{top}(\bot) = \bot$.*

A call stack symbol $\langle l, o, C.\psi \rangle \in \texttt{Elem}$ denotes the program execution point at line l of the instance method $C.\psi$ that is invoked on the object o, and $\langle l, *, C.\psi \rangle \in \texttt{Elem}$ represents an execution point inside a static method $C.\psi$.

Table 2. Transition Rules on the Heap and Call Stack

$\texttt{stmt}(l)$ from $C.\psi$	henv'	stack'
$x =$ new T	$\texttt{henv} \odot [x \mapsto \nu(\texttt{henv})]$	
$x = y$	$\texttt{henv} \odot [x \mapsto \texttt{henv}(y)]$	$\texttt{push}(\texttt{s}, \texttt{e})$ where
$x :=$ (T)y	$\texttt{henv} \odot [x \mapsto \texttt{henv}(y)]$	$\texttt{s} = \texttt{pop}(\texttt{stack})$
$x :=$ @this : T	$\texttt{henv} \odot [x \mapsto \texttt{henv}(\texttt{this})]$	$\texttt{e} = \langle \texttt{next}(l), o, C.\psi \rangle$
$x :=$ @parameter$_k$: T	$\texttt{henv} \odot [x \mapsto \texttt{henv}(\texttt{arg}_k)]$	$o =' *'$ if $C.\psi$ is static
$x = y[i]$	$\texttt{henv} \odot [x \mapsto \texttt{henv}(y)[i]]$	$o \in \texttt{henv}(\texttt{this}^{C.\psi})$ o.w.
$y[i] = x$	$\texttt{henv} \odot [\texttt{henv}(y)[i] \mapsto \texttt{henv}(x)]$	
$x = y.f$	$\texttt{henv} \odot [x \mapsto \texttt{henv}(\texttt{henv}(y).f)]$	
$y.f = x$	$\texttt{henv} \odot [\texttt{henv}(y).f \mapsto \texttt{henv}(x)]$	
return y	$\texttt{henv} \odot [\texttt{ret} \mapsto \texttt{henv}(y)]$	$\texttt{pop}(\texttt{stack})$
$z = r_0.m(r_1, ..., r_n)$	$\texttt{henv} \odot [\texttt{this}^{C'.\psi'} \mapsto \texttt{henv}(r_0)]$	$\texttt{push}(\texttt{s}', \texttt{e}')$ where
	$\odot [\texttt{arg}_1^{C'.\psi'} \mapsto \texttt{henv}(r_1)] \odot \cdots$	$\texttt{s}' = \texttt{push}(\texttt{s}, \texttt{e})$
	$\odot [\texttt{arg}_n^{C'.\psi'} \mapsto \texttt{henv}(r_n)]$	$\texttt{s} = \texttt{pop}(\texttt{stack})$
	$\odot [z \mapsto \texttt{ret}^{C'.\psi'}]$	$\texttt{e} = \langle \texttt{next}(l), o, C.\psi \rangle$
	where $\psi' \in \Psi$ is the method signature of m,	$\texttt{e}' = \langle l_0^{C'.\psi'}, o', C'.\psi' \rangle$
	and $C'.\psi' = \texttt{resolve}(\texttt{TypeOf}(o'), \psi', \Theta)$	
	for $o' = \texttt{henv}(r_0)$.	
$z = C'.m(r_1, ..., r_n)$	$\texttt{henv} \odot [\texttt{arg}_1^{C'.\psi'} \mapsto \texttt{henv}(r_1)] \odot \cdots$	$\texttt{push}(\texttt{s}, \texttt{e}')$ where
	$\odot [\texttt{arg}_1^{C'.\psi'} \mapsto \texttt{henv}(r_1)]$	$\texttt{s} = \texttt{push}(\texttt{pop}(\texttt{stack}), \texttt{e})$
	$\odot [z \mapsto \texttt{ret}^{C'.\psi'}]$	$\texttt{e} = \langle \texttt{next}(l), o, C.\psi \rangle$
	where $\psi' \in \Psi$ is the method signature of m,	$\texttt{e}' = \langle l_0^{C'.\psi'}, *, C'.\psi' \rangle$
	and $C'.\psi' = \texttt{resolve}(C', \psi', \Theta)$.	

We use a transition system $(\texttt{states}, \texttt{s}_{init}, \rightarrow)$ to represent the operational Java semantics on the heap and call stack, where $\texttt{states} \subseteq \mathcal{L} \times \Theta \times \Lambda \times \Pi$ is the set of program states, each of which is a tuple of program locations, class environment, heap environments and calling histories, and $\texttt{s}_{init} \in \texttt{states}$ is the initial state; $\rightarrow \subseteq \texttt{states} \times \texttt{states}$ is the set of transition rules.

As far as single-threaded Java programs are concerned, the next program location after each execution step at l ($\in \mathcal{L}$) is uniquely determined, and is denoted by $\texttt{next}(l)$. As given in Table 2, for the program execution of the statement $\texttt{stmt}(l)$ at $l \in \mathcal{L}$ from the method $C.\psi$, the transition rule is $\langle l, \Theta, \texttt{henv}, \texttt{stack}\rangle \rightarrow \langle \texttt{next}(l), \Theta, \texttt{henv}', \texttt{stack}'\rangle$. Here, ν is a function that generates a fresh heap object. $\texttt{this}$, $\texttt{arg}_k$ and $\texttt{ret}$ are fresh variables to denote the *this* reference of a class instance, the k^{th} method argument, and the return variable, respectively. $\texttt{TypeOf} : \mathcal{O} \rightarrow \mathcal{C}$ is the function that returns the runtime type of a heap object. $\texttt{resolve} : \mathcal{C} \times \Psi \times \Theta \rightarrow \mathcal{C}.\Psi$ is the function implements how JVM resolves and the method to be invoked at runtime according to its method signature and possible enclosing class. $l_0^{C'.\psi'}$ refers to the entry point of the method $C'.\psi'$.

3.2 Abstraction

We apply the following abstractions to abstract away various sources of infinities.

- A unique abstract heap object models concrete heap objects created at the same allocation site (a.k.a., the context-insensitive heap abstraction). Thus, the number of abstract heap objects are syntactically bounded to be finite. An abstract heap object is a pair of its *allocation site* and *runtime type*.
- The indices of arrays are ignored, such that members of an array are not distinguished. We denote by $[\![o]\!]$ the unique representative for all members of the array instance o ($\in \mathcal{O}$). After abstracting the set of heap objects to be finite, the nesting of array and field reference become finite correspondingly.

Definition 9. *The set of abstract heap objects is* $\mathtt{Obj} = (\mathcal{L} \cup \{\_\}) \times \mathcal{C}$*, where* $\_$ *is a fresh symbol for indicating nowhere. Let* $\mathtt{TypeOf} : \mathtt{Obj} \rightarrow \mathcal{C}$ *be the function returns the second projection of an abstract heap object.*

Definition 10. *Let* $\mathtt{RetPoint} \subseteq \mathcal{L} \times \mathcal{C}.\Psi$ *be the set of return points for method invocations. Define abstractions* $\alpha_t : \mathtt{Elem} \rightarrow \{\_\} \times \mathcal{C}.\Psi$*,* $\alpha_r : \mathtt{Elem} \rightarrow \mathtt{RetPoint}$*, and* $\alpha_o : \mathtt{Elem} \rightarrow \mathtt{Obj} \times \mathcal{C}.\Psi$ *such that, for* $\langle l, o, C.\psi\rangle \in \mathtt{Elem}$*,*

- $\alpha_t(\langle l, o, C.\psi\rangle) = (\_, C.\psi)$*,*
- $\alpha_r(\langle l, o, C.\psi\rangle) = (l, C.\psi)$*, and*
- $\alpha_o(\langle l, o, C.\psi\rangle) = (o, C.\psi)$*.*

α_r *and* α_o *are extended to the call stack in an element-wise manner.*

Definition 11. *Let* $\mathbb{C} = (\{\_\} \times (\mathcal{C}.\Psi)).\mathtt{RetPoint}^*$ *be the set of abstract calling contexts. Define a calling context abstraction* $\alpha : \Pi \rightarrow \mathbb{C}$ *such that* $\alpha(\bot) = \epsilon$ *and* $\alpha(\mathtt{e.stack}) = \alpha_t(\mathtt{e}).\alpha_r(\mathtt{stack})$ *for* $\mathtt{e} \in \mathtt{Elem}$*, and* $\mathtt{stack} \in \Pi$*.*

By Def. 10, α_t abstracts the topmost stack symbol for flow-insensitivity. α_r abstracts the return points of method invocations, which results in calling contexts in terms of call site strings. Our choice of α_r indicates, method invocations to the same method from different places of the same caller is still distinguished. An alternative of α_r is α_o, which abstracts calling contexts as sequences of heap objects on which methods are invoked, also known as object-sensitivity [9].

Definition 12. *We denote by* $\mathtt{Ref}$ *the set of abstract reference variables, and by* $\mathtt{cc}(v)$ *all possible abstract calling contexts for a reference variable* $v \in \mathtt{Ref}$.

Note that, $\mathtt{cc}(v)$ are exactly all possible calling contexts for the method which v belongs to, and $\mathtt{cc}(v)$ is automatically computed as the set of reachable regular configurations during model checking (Section 4).

Definition 13. *Let* $\boldsymbol{R} : \mathtt{Ref} \times \mathbb{C} \to \mathcal{P}(\mathtt{Obj})$ *be the function that stores the points-to relation, where* $\mathcal{P}$ *denotes the powerset operator.* $\boldsymbol{R}\downarrow_V : V \times \mathbb{C} \to \mathcal{P}(\mathtt{Obj})$ *is the restriction of* $\boldsymbol{R}$ *to* $V \subseteq \mathtt{Ref}$. *Define* $\ominus : \boldsymbol{R} \times \boldsymbol{R} \to \boldsymbol{R}$, *such that for any* $r \in \mathtt{Ref}$ *and* $\mathtt{cc} \in \mathbb{C}$, $(\boldsymbol{R}_1 \ominus \boldsymbol{R}_2)(r, \mathtt{cc}) = \boldsymbol{R}_1(r, cc) \setminus \boldsymbol{R}_2(r, \mathtt{cc})$.

4 Detecting Points-to Information by Model Checking

This section presents, given points-to information and a call graph, how to detect new points-to information and enlarge a call graph in a context/field-sensitive and flow-insensitive way.

Definition 14. $\mathcal{G} = (M, \boldsymbol{E})$ *is a call graph of a program if* $M \subseteq \mathcal{C}.\Psi$ *and* $\boldsymbol{E} \subseteq M \times \mathcal{L} \times M$. *An element in* $\boldsymbol{E}$ *is called a call edge.*

We call $\mathbf{P} \subseteq \mathcal{C}.\Psi$ a program coverage, and denote the program coverage consisting of enclosing methods of program entries by $\mathbf{P}_0$.

Definition 15. *Let* $\mathtt{Henv}$, $\mathtt{sp}$ *be fresh symbols. Define a weighted pointer assignment graph (WPAG)* $G = (N, L, \rightsquigarrow, n_0)$, *where* $N \subseteq (\mathtt{Ref} \cup \{\mathtt{Henv}\}) \times (\mathcal{C}.\Psi \cup \mathtt{RetPoint})$ *is the set of nodes,* $L \subseteq \{\lambda x.\{o\} \mid o \in \mathtt{Obj}\} \cup \{\lambda x.x\}$ *is the set of labels,* $\rightsquigarrow \subseteq N \times L \times N$ *is the set of edges, and* $n_0 = (\mathtt{Henv}, \mathtt{sp}) \in N$ *is the root.*

A WPAG G is a directed labeled graph to represent data flow of heap objects. `Henv` and `sp` indicate the program environment that provides new abstract heap objects and program inputs, and the dummy program entry, respectively. The first projection of N represents abstract references, and the second projection represents their program scopes. Edges of G are classified into inter-edges ($\rightsquigarrow_i$, defined in Table 3) and intra-edges ($\rightsquigarrow_c, \rightsquigarrow_r, \rightsquigarrow_t$, defined in Table 4). An edge $(v, m) \rightsquigarrow (v', m')$ is denoted by

$$\begin{cases} \rightsquigarrow_i & \text{if } m = m' \in \mathcal{C}.\Psi \\ \rightsquigarrow_c & \text{if } m \neq m' \text{ and } m, m' \in \mathcal{C}.\Psi \end{cases} \qquad \begin{cases} \rightsquigarrow_r & \text{if } m' \in \mathtt{RetPoint} \\ \rightsquigarrow_t & \text{if } m \in \mathtt{RetPoint} \end{cases}$$

The procedure of finding new points-to information consists of the following steps. Step 1 builds intra-edges of a WPAG G given $\mathbf{R}$. Step 2 builds inter-edges of G given $\mathbf{E}$. During Step 2, the set of return points associated with each method invocation is recorded as a mapping $\mathbb{M} : \rightsquigarrow_c \to \mathcal{P}(\mathtt{RetPoint})$. Initially, $\mathbb{M}$ is the constant function to the empty set. Step 3 encodes G as a WPDS W and detects new points-to information, denoted by $\widehat{\mathbf{R}}$, by model checking.

Table 3. $\mathcal{A}[\![\_]\!] : \mathcal{S}_\epsilon \to \mathcal{P}(\rightsquigarrow_i)$

$\mathcal{A}[\![x = \text{new } \mathtt{T}]\!] = \{(\mathtt{Henv}, C.\psi) \overset{\lambda x.\{(l,\mathtt{T})\}}{\rightsquigarrow_i} (x, C.\psi)\}$
$\mathcal{A}[\![x = y]\!] = \{(y, C.\psi) \rightsquigarrow_i (x, C.\psi)\}$
$\mathcal{A}[\![x := (\mathtt{T})y]\!] = \{(y, C.\psi) \rightsquigarrow_i (x, C.\psi)\}$
$\mathcal{A}[\![x := @\text{this} : \mathtt{T}]\!] = \{(\mathtt{this}, C.\psi) \rightsquigarrow_i (x, C.\psi)\} \cup A_e$
where $A_e = \{(\mathtt{Henv}, C.\psi) \overset{\lambda x.\{(\_,\mathtt{T})\}}{\rightsquigarrow_i} (\mathtt{this}, C.\psi)\}$ if $C.\psi \in \mathbf{P}_0$ and $A_e = \emptyset$ otherwise
$\mathcal{A}[\![x := @\text{parameter}_k : \mathtt{T}]\!] = \{(\mathtt{arg}_k, C.\psi) \rightsquigarrow_i (x, C.\psi)\} \cup A_p$
where $A_p = \{(\mathtt{Henv}, C.\psi) \overset{\lambda x.\{(\_,\mathtt{T})\}}{\rightsquigarrow_i} (\mathtt{arg}_k, C.\psi)\}$ if $C.\psi \in \mathbf{P}_0$ and $A_p = \emptyset$ otherwise
$\mathcal{A}[\![\text{return } x]\!] = \{(x, C.\psi) \rightsquigarrow_i (\mathtt{ret}, C.\psi)\}$
$\mathcal{A}[\![x = y[i]]\!] = \{([\![o]\!], C.\psi) \rightsquigarrow_i (x, C.\psi) \mid o \in \mathbf{R}(y, \mathtt{cc}(y))\} \cup A_g$
$\mathcal{A}[\![y[i] = x]\!] = \{(x, C.\psi) \rightsquigarrow_i ([\![o]\!], C.\psi) \mid o \in \mathbf{R}(y, \mathtt{cc}(y))\} \cup A_g$
where $A_g = \{(\mathtt{Henv}, C.\psi) \overset{\lambda x.s}{\rightsquigarrow_i} ([\![o]\!], C.\psi) \mid o \in \mathbf{R}(y, \mathtt{cc}(y)), s = \mathbf{R}([\![o]\!], \mathtt{cc}([\![o]\!]))\}$
$\mathcal{A}[\![x = y.f]\!] = \{(o.f, C.\psi) \rightsquigarrow_i (x, C.\psi) \mid o \in \mathbf{R}(y, \mathtt{cc}(y))\} \cup A_f$
$\mathcal{A}[\![y.f = x]\!] = \{(x, C.\psi) \rightsquigarrow_i (o.f, C.\psi) \mid o \in \mathbf{R}(y, \mathtt{cc}(y))\} \cup A_f$
where $A_f = \{(\mathtt{Henv}, C.\psi) \overset{\lambda x.s}{\rightsquigarrow_i} (o.f, C.\psi) \mid o \in \mathbf{R}(y, \mathtt{cc}(y)), s = \mathbf{R}(o.f, \mathtt{cc}(o.f))\}$

Step 1: Building Intra-procedural Data Flows

Table 3 gives rules that translate statements from $\mathcal{S}_\epsilon$ at line $l(\in \mathcal{L})$ of the method $C.\psi$ to intra-edges of G, denoted by $\mathcal{A}[\![\_]\!] : \mathcal{S}_\epsilon \to \mathcal{P}(\rightsquigarrow_i)$. For simplicity, we omit a weight associated to $\rightsquigarrow$ if it is $\lambda x.x$. Our modeling is featured as follows.

- In contrast to cloning-based approach, there is the unique abstract reference of each local reference variable. Global references are cloned only for methods inside which they are referred.
- Instead of passing global variables explicitly as parameters along procedure calls and returns, the heap memory is modelled with the global data structure $\mathbf{R}$ and provides global references with *cached data flows* (i.e., A_g, A_f) when they are locally referred (only necessary for field read).

Step 2: Building Inter-procedural Data Flows

Table 4 gives rules that translate statements from $\mathcal{S}_I$ that contains explicit method invocations to inter-edges of G, denoted by $\mathcal{A}[\![\_]\!] : \mathcal{S}_I \to \rightsquigarrow_c \cup \rightsquigarrow_r \cup \rightsquigarrow_t$, where A_c denotes call edges, A_r denotes return edges, and A_t denotes data flows from return points to the calling procedure. Note that, `Henv` as the program environment is explicitly passed as a parameter among calls and returns. During generation of inter edges, the mapping $\mathbb{M}$ is updated with newly produced return points. The translation rules for static method invocations can be defined similarly. Finally, new edges $\{n_0 \rightsquigarrow (\mathtt{Henv}, C.\psi) \mid n_0 = (\mathtt{Henv}, \mathtt{sp}), C.\psi \in \mathbf{P}_0\}$ are added to G that lead from the dummy root node n_0 to the program entries.

Step 3: Building the WPDS W from G and Model Checking

Definition 16. *Let* $D_1 = \{\lambda x.s \mid s \in \mathcal{P}(\mathit{Obj})\}$ *and* $D_2 = \{\lambda x.x \cup s \mid s \in \mathcal{P}(\mathit{Obj})\}$. *Define a bounded idempotent semiring* $S = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, *such that*

Table 4. $\mathcal{A}[\![\_]\!] : \mathcal{S}_I \rightarrow \rightsquigarrow_c \cup \rightsquigarrow_r \cup \rightsquigarrow_t$

$\mathcal{A}[\![z = r_0.f(r_1, ..., r_n)]\!] \;=\; A_c \cup A_r \cup A_t$
where $A_c = \{(r_0, C.\psi) \rightsquigarrow_c (\texttt{this}^{C'.\psi'}, C'.\psi')\} \cup \{(\texttt{Henv}, C.\psi) \rightsquigarrow_c (\texttt{Henv}, C'.\psi')\}$
$\cup \bigcup_{r_i \in \texttt{Ref}} \{(r_i, C.\psi) \rightsquigarrow_c (\texttt{arg}_i{}^{C'.\psi'}, C'.\psi')\}$
$A_r = \{(\texttt{ret}^{C'.\psi'}, C'.\psi') \rightsquigarrow_r (\texttt{ret}^{C'.\psi'}, \texttt{rp})\} \cup \{(\texttt{Henv}, C'.\psi') \rightsquigarrow_r (\texttt{Henv}, \texttt{rp})\}$
$A_t = \{(\texttt{ret}^{C'.\psi'}, \texttt{rp}) \rightsquigarrow_t (z, C.\psi)\} \cup \{(\texttt{Henv}, \texttt{rp}) \rightsquigarrow_t (\texttt{Henv}, C.\psi)\}$
ψ' is the method signature of the method f, and
$(C.\psi, l, C'.\psi') \in \mathbf{E}$, and $\texttt{rp} = (l, C.\psi)$, and
for all $r \in A_c,\ \mathbb{M}(r) = \mathbb{M}(r) \cup \{\texttt{rp}\}$

- *The weighted domain* $D = D_1 \cup D_2 \cup \{\boldsymbol{0}\}$*, and* $\boldsymbol{1} = \lambda x.x$*;*
- $d_1 \otimes d_2 = d_1 \oplus d_2 = \lambda x.\ d_1(x) \cup d_2(x)$ *for* $d_1, d_2 \in D \setminus \{\boldsymbol{0}\}$
- $d \otimes \boldsymbol{0} = \boldsymbol{0} \otimes d = \boldsymbol{0}$ *for* $d \in D$*;*

It is easy to see that both the distributivity of $\otimes$ over $\oplus$ and the associativity of $\otimes$ hold. D_1 consists of constant functions, and $\lambda x.s \in D_1$ is that a reference points to the set of abstract heap objects s; and $\lambda x.x \cup s \in D_2$ is that a reference may keep unchanged along a path and be updated to point to s along another.

Given a WPAG G from Definition 15, a WPDS $W = (P, S, f)$ with $P = (Q, \Gamma, \Delta, q_0, w_0)$ is encoded G as follows,

- The set of control locations Q is the first projection of N, i.e., $\texttt{Ref} \cup \{\texttt{Henv}\}$;
- The stack alphabet Γ is the second projection of N, i.e., $\mathcal{C}.\Psi \cup \texttt{RetPoint}$;
- S is from Definition 16;
- $q_0 = \texttt{Henv}$ and $w_0 = \texttt{sp}$;
- For each edge r represented as $(v_1, m_1) \overset{l}{\rightsquigarrow} (v_2, m_2)$ such that $f(r) = l$ and
  - $\langle v_1, m_1 \rangle \hookrightarrow \langle v_2, m_2 \rangle$ if $r \in \rightsquigarrow_i$ or $\rightsquigarrow_t$;
  - $\langle v_1, m_1 \rangle \hookrightarrow \langle v_2, m_2 m_r \rangle$ for each $m_r \in \mathbb{M}(\rightsquigarrow_c)$ if $r \in \rightsquigarrow_c$;
  - $\langle v_1, m_1 \rangle \hookrightarrow \langle v_2, \epsilon \rangle$ if $r \in \rightsquigarrow_r$.

Definition 17. *Let* $W = (P, S, f)$ *be a weighted pushdown system with* $P = (Q, \Gamma, \Delta, q_0, \gamma_0)$*. For any reference* $v \in \texttt{Ref}$ *from the method* $C.\psi$*,* $\widehat{\boldsymbol{R}}(v, \texttt{cc}(v)) = \widehat{\mathit{MOVP}}(c, \langle v, C.\psi \rangle, W)\ (H_{\mathit{init}}(v))$*, where* $c = \langle q_0, \gamma_0 \rangle$ *and* H_{init} *denotes the initial abstract heap environment such that* $H_{\mathit{init}}(v) = \emptyset$ *for any* $v \in \texttt{Ref}$*.*

To solve $\texttt{MOVP}(C_s, C_t, W)$, we (i) first compute $\texttt{gps}(c)$ for each $c \in C_t$ given C_s, and then (ii) read out and combine the value of all paths between C_s and C_t. We denote by $H = 2^{|\texttt{Obj}|}$ the length of the longest descending chain of the weighted domain, and by T the time to perform either $\otimes$ or $\oplus$. In our case, the time required to perform step (ii) can be ignored, and the worst case time complexity of performing step (i) is $\mathcal{O}(|Q|^2\ |\Delta|\ |\Gamma|\ H\ T)$.

5 Acceleration by Lightening Iterative Cycles

5.1 A Traditional Iterative Procedure Scheme

Algo. `OnTheFlyPTA` in Fig. 1 sketches a procedure scheme for on-the-fly Java PTA. It starts with analyzing the program entry points $\mathbf{P}_0$, and computes the call graph $\mathbf{E}$ and points-to relation $\mathbf{R}$ until convergence. For each iterative cycle,

- `FindPointsTo` : $\mathbf{P} \times \mathbf{E} \times \mathbf{R} \rightarrow \mathbf{R}$ (line 3) detects points-to information $\widehat{\mathbf{R}}$ on the partial program $\mathbf{P}$, according to updated information in the previous iteration. The updated points-to information $\Delta\mathbf{R}$ is derived at line 4.
- `FindCallEdges` : $\mathbf{P} \times \mathbf{R} \times \Theta \rightarrow \mathbf{E}$ (line 5) resolves call relation $\widehat{\mathbf{E}}$ according to $\widehat{\mathbf{R}}$, obeying to the standard JVM semantics. The updated call relation $\Delta\mathbf{E}$ is derived at line 6.
- `TakeReachables` : $\mathbf{E} \rightarrow \mathbf{P}$ (line 8) returns the set of methods (reachable from program entries) to be analyzed in the next iteration. It can be defined as the union of the first and third projection of $\mathbf{E}$.

Algorithm `OnTheFlyPTA`
Input: the program entry points $\mathbf{P}_0$ and the class environment Θ
Output: $\mathcal{G} = (M, \mathbf{E})$ and $\mathbf{R}$
0. $\mathbf{E} := \emptyset$; $\mathbf{R} := \emptyset$; $\mathbf{P} := \mathbf{P}_0$
1. **do**
2. $\quad \widehat{\mathbf{R}} = \texttt{FindPointsTo}(\mathbf{P}, \mathbf{E}, \mathbf{R})$
3. $\quad \Delta\mathbf{R} := \widehat{\mathbf{R}} \ominus \mathbf{R}$
4. $\quad \mathbf{R} := \mathbf{R} \sqcup \Delta\mathbf{R}$
5. $\quad \widehat{\mathbf{E}} := \texttt{FindCallEdges}(\mathbf{P}, \mathbf{R}, \Theta)$
6. $\quad \Delta\mathbf{E} := \widehat{\mathbf{E}} \setminus \mathbf{E}$
7. $\quad \mathbf{E} := \mathbf{E} \cup \Delta\mathbf{E}$
8. $\quad \mathbf{P} := \texttt{TakeReachables}(\mathbf{E})$
9. **while** $\Delta\mathbf{E} \neq \emptyset$ or $\Delta\mathbf{R} \neq \emptyset$

Fig. 1. A Procedure for On-the-fly Java Points-to Analysis

Theorem 1. *The algorithm* `OnTheFlyPTA` *terminates if (i) the domain of* $\mathbf{P}$, $\mathbf{E}$ *and* $\mathbf{R}$ *are finite, and (ii) each of these functions* `FindPointsTo`, `FindCallEdges` *and* `TakeReachables` *is monotonic wrt the set inclusion on* $\mathbf{P}$, $\mathbf{E}$ *and the element-wise extension on* $\mathbf{R}$ *of the set inclusion.*

`FindPointsTo` is the core procedure of PTA for Java. For most cloning-based algorithms, `FindPointsTo` corresponds to the propagation of points-to sets, which is typically reduced to constraint solving problems. In contrast, we derive the analysis algorithm as model checking problems on WPDSs (Section 4). Since the abstraction given in Section 3.2 is an over approximation, soundness of our analysis is straightforward.

5.2 A Two-Staged Iterative Procedure Scheme

For an on-the-fly points-to analysis, the program coverage is enlarged when points-to analysis proceeds. However, we found that only part of the whole program would effectively contribute to the enlargement of the program coverage. To boost on-the-fly PTA, we propose a two-staged iterative procedure, denoted by $(\texttt{LE} \circ \texttt{GU})^*$, which combines two phases of `LE` (*local exploration*) and `GU` (*global update*). Generally, an `LE` iteration localizes the analysis on small parts of the program, which is more likely to enlarge the program coverage, and GU is performed on-demand for guaranteeing completeness. Line 9 switches `LE` and `GU` when conditions defined by `SwitchCond` are satisfied. Otherwise, a `GU` iteration will be triggered to check sound convergence.

Algorithm `TwoStaged_OnTheFlyPTA`
Input: the program entry points $\mathbf{P}_0$ and the class environment Θ
Output: $\mathcal{G} = (\mathcal{M}, \mathbf{E})$ and $\mathbf{R}$
0. $\mathbf{E} := \emptyset$; $\mathbf{R} := \emptyset$; $\mathbf{P} := \mathbf{P}_0$; `NotDone` = `NotDone'` := *true*
1. **do**
2. `NotDone` := `NotDone'`
3. $\widehat{\mathbf{R}} = \texttt{FindPointsTo}(\mathbf{P}, \mathbf{E}, \mathbf{R})$
4. $\Delta\mathbf{R} := \widehat{\mathbf{R}} \ominus \mathbf{R}$
5. $\mathbf{R} := \mathbf{R} \sqcup \Delta\mathbf{R}$
6. $\widehat{\mathbf{E}} := \texttt{FindCallEdges}(\mathbf{P}, \mathbf{R}, \Theta)$
7. $\Delta\mathbf{E} := \widehat{\mathbf{E}} \setminus \mathbf{E}$
8. $\mathbf{E} := \mathbf{E} \cup \Delta\mathbf{E}$
9. **if** $\texttt{SwitchCond}(\Delta\mathbf{E}, \Delta\mathbf{R}) = \mathit{true}$ **then**
10. $\mathbf{P} := \texttt{TakeCoverage}(\mathbf{E}, \Delta\mathbf{E}, \Delta\mathbf{R})$
11. `NotDone'` := *true*
12. **else**
13. $\mathbf{P} := \texttt{TakeReachables}(\mathbf{E})$
14. `NotDone'` := *false*
15. **while** $\Delta\mathbf{E} \neq \emptyset$ or $\Delta\mathbf{R} \neq \emptyset$ or `NotDone` = *true*

Fig. 2. A Two-Staged Procedure for On-the-fly Java Points-to Analysis

Definition 18. $\mathit{SwitchCond}(\Delta\boldsymbol{E}, \Delta\boldsymbol{R}) = (\Delta\boldsymbol{E} \neq \emptyset) \vee (\Delta\boldsymbol{R} \downarrow_{\mathit{Ref}_f} \neq \emptyset)$, *where* $\mathit{Ref}_f \subseteq \mathit{Ref}$ *is the set of base references of instance fields.*

Def. 18 means that, an `LE` is triggered when either new call edges are detected or new global references are found. Both indicates that the underlying model for model checking is extended with new pushdown transitions.

Definition 19. $\mathit{TakeCoverage}(\boldsymbol{E}, \Delta\boldsymbol{E}, \Delta\boldsymbol{R}) = M_1 \cup M_2 \cup M_3 \cup \mathit{TakeReachables}(\Delta\boldsymbol{E})$,

$$M_1 = \left\{ m'' \mid \begin{array}{l} \exists m, m' \in \mathcal{C}.\Psi\ \exists l, l' \in \mathcal{L}.\ (m, l, m') \in \Delta\boldsymbol{E}, (m, l', m'') \in \boldsymbol{E}, \\ \textit{and the return type of } m'' \textit{ is a reference type} \end{array} \right\}$$
$$M_2 = \{C.\psi \mid v^{C.\psi} \in \mathit{Ref}_f \textit{ and } \Delta\boldsymbol{R}(v^{C.\psi}, \mathtt{cc}(v^{C.\psi})) \neq \emptyset\}$$
$$M_3 = \{m, C.\psi \mid \Delta\boldsymbol{R}(\mathit{ret}^{C.\psi}, \mathtt{cc}(\mathit{ret}^{C.\psi})) \neq \emptyset \textit{ and } \exists l \in \mathcal{L}.(m, l, C.\psi) \notin \boldsymbol{E}\}$$

A partial model is taken in the ways defined in `TakeCoverage`, where M_1 says that, if a new call relation found from m to m', other callees of m that returns values of reference type are collected. M_2 says that, if the points-to information of base variables of instance fields are updated, their enclosing methods are collected. M_3 says that, if the points-to information of return variables of $C.\psi$ is updated, $C.\psi$ and methods that call $C.\psi$ and are not included in the previous LE are collected. Note that, our choice of `TakeCoverage` is inspired and decided by empirical studies on practiced Java benchmarks regarding efficiency.

Theorem 2. *The algorithm* $\mathit{TwoStaged\_OnTheFlyPTA}$ *terminates if (i) the domain of* $\boldsymbol{P}$, $\mathbf{E}$ *and* $\mathbf{R}$ *are finite, and (ii) each of these functions* $\mathit{FindPointsTo}$, $\mathit{FindCallEdges}$, $\mathit{TakeCoverage}$ *and* $\mathit{TakeReachables}$ *is monotonic on all arguments from their domains, and (iii)* $\mathit{SwitchCond}(\emptyset, \emptyset) = \text{false}$.

5.3 Adding Summary Transition Rules in LE

As given in Table 5 that extends translation rules Table 5, to make the two-staged iterative procedure work effectively, *summary transition rules* (i.e., A_c) that carry cached data flows to $\mathbf{P}_E$ (Def. 20) are introduced, when building a WPAG in an LE iteration. Translation rules leading from the dummy root node n_0 to the program entries are lifted to $\{n_0 \rightsquigarrow (\mathtt{Henv}, C.\psi) \mid n_0 = (\mathtt{Henv}, \mathtt{sp}), C.\psi \in \mathbf{P}_E\}$.

Definition 20. *Given a program coverage* $\boldsymbol{P}$ *and the call relation* $\boldsymbol{E}$, $\boldsymbol{P}_E = \{m \in \boldsymbol{P} \mid m' \notin \boldsymbol{P} \textit{ if } (m', m) \in \boldsymbol{E}\}$.

Table 5. $\mathcal{B}[\![\_]\!] : \mathcal{S}_\epsilon \rightarrow \mathcal{P}(\rightsquigarrow_i)$

$\mathcal{B}[\![x := @\text{this} :\ \mathtt{T}]\!] = \mathcal{A}[\![x := @\text{this} :\ \mathtt{T}]\!] \cup A_c$
where $A_c = \{(\mathtt{Henv}, C.\psi) \overset{\lambda x.s}{\rightsquigarrow}_i (\mathtt{this}, C.\psi) \mid s = \mathbf{R}(\mathtt{this}, \mathtt{cc}(\mathtt{this}))\}$
if $C.\psi \in \mathbf{P}_E$ and $A_c = \emptyset$ otherwise
$\mathcal{B}[\![x := @\text{parameter}_k :\ \mathtt{T}]\!] = \mathcal{A}[\![x := @\text{parameter}_k :\ \mathtt{T}]\!] \cup A_c$
where $A_c = \{(\mathtt{Henv}, C.\psi) \overset{\lambda x.s}{\rightsquigarrow}_i (\mathtt{arg}_k, C.\psi) \mid s = \mathbf{R}(\mathtt{arg}_k, \mathtt{cc}(\mathtt{this}))\}$
if $C.\psi \in \mathbf{P}_E$ and $A_c = \emptyset$ otherwise

In sequel, we show that adding summary transitions to arguments (i.e., $\mathtt{arg}_i, \mathtt{this}$) of methods from $\mathbf{P}_E$ will not cause any loss of precision.

Definition 21. *For a pushdown system $P = (Q, \Gamma, \Delta, q_0, \omega_0)$, define $\trianglerighteq \subseteq Q \times \Gamma \times Q \times \Gamma$, such that $\langle p, \gamma\rangle \trianglerighteq \langle p', \gamma'\rangle$ if there exists $\langle p, \gamma\rangle \Rightarrow^* \langle p', \gamma'\omega'\rangle$ for some $\omega' \in \Gamma^*$. Define $\trianglerighteq_s \subseteq \trianglerighteq$ such that $\langle p, \gamma\rangle \trianglerighteq_s \langle p', \gamma'\rangle$ if (i) $\langle p, \gamma\rangle \trianglerighteq \langle p', \gamma'\rangle$ and (ii) for each $\omega \in \Gamma^*$ and all transition sequence of $\sigma : \langle p, \gamma\omega\rangle \Rightarrow^* \langle p', \gamma'\omega'\omega\rangle$ for some $\omega' \in \Gamma^*$, and any $\langle p'', \omega''\rangle$ appearing in σ satisfies $|w''| > |w|$. $\langle p, \gamma\rangle$ is a **dominator** of $\langle p', \gamma'\rangle$ if $\langle p, \gamma\rangle \trianglerighteq_s \langle p', \gamma'\rangle$.*

Definition 22. *Let $W = (P, S, f)$ be a WPDS with $P = (Q, \Gamma, \Delta, q_0, \gamma_0)$. For $p \in Q, \gamma \in \Gamma$, $\{\langle p_i, \gamma_i\rangle \mid 0 \leq i \leq k\}$ is a **dominator set** of $\langle p, \gamma\rangle$ if (1), for each i with $0 \leq i \leq k$, $\langle p_i, \gamma_i\rangle \trianglerighteq_s \langle p, \gamma\rangle$, and (2), for each transition sequence $\sigma : \langle q_0, \gamma_0\rangle \Rightarrow^* \langle p, \gamma w\rangle$ with $w \in \Gamma^*$, there uniquely exists $\langle p_j, \gamma_j\rangle$ such that $\langle p_j, \gamma_j w'\rangle$ for some $w' \in \Gamma^*$ appears in σ.*

Lemma 1. *Given a WPDS $W = (P, S, f)$ where $P = (Q, \Gamma, \Delta, q_0, \gamma_0)$. For $p \in Q, \gamma \in \Gamma$, let $\mathbb{H}$ be a dominator set of $\langle p, \gamma\rangle$ and let $c = \langle q_0, \gamma_0\rangle$, we have $\widehat{\mathit{MOVP}}(c, \langle p, \gamma\rangle, W) = \bigoplus_{\langle p_i, \gamma_i\rangle \in \mathbb{H}} \widehat{\mathit{MOVP}}(c, \langle p_i, \gamma_i\rangle, W) \otimes \widehat{\mathit{MOVP}}(\langle p_i, \gamma_i\rangle, \langle p, \gamma\rangle, W)$.*

Proof. Straightforward by definitions of frontiers and MOVP problems.

By Lemma 1, the computation of MOVP problems can be soundly divided into two independent phases via dominators.

Definition 23. *Given a WPDS $W = (P, S, f)$ with $P = (Q, \Gamma, \Delta, q_0, \gamma_0)$. For $p \in Q \setminus \{q_0\}, \gamma \in \Gamma \setminus \{\gamma_0\}$, $\langle p, \gamma\rangle$ is a **frontier** of W if either $\langle p, \gamma\rangle \trianglerighteq_s \langle p', \gamma'\rangle$ or $\langle p, \gamma\rangle \not\trianglerighteq \langle p', \gamma'\rangle$ for any $p' \in Q, \gamma' \in \Gamma$. A **frontier set** of W, denoted by $\mathbb{F}_W$, is a set of frontiers and $\langle p, \gamma\rangle \in \mathbb{F}_W$ implies $\langle p', \gamma'\rangle \notin \mathbb{F}_W$ if $\langle p, \gamma\rangle \trianglerighteq_s \langle p', \gamma'\rangle$.*

Theorem 3. *Given a WPDS $W = (P, S, f)$ with $P = (Q, \Gamma, \Delta, q_0, \gamma_0)$. Let $W' = (P', S, f')$ with $P' = (Q', \Gamma', \Delta' \cup \delta, q_0, \gamma_0)$, where $Q' \subseteq Q$, $\Gamma' \subseteq \Gamma$, $\Delta' \subseteq \Delta$, and $\delta = \{r = \langle q_0, \gamma_0\rangle \hookrightarrow \langle p_i, \gamma_i\rangle, f'(r) = \widehat{\mathit{MOVP}}(c, \langle p_i, \gamma_i\rangle, W) \oplus f(r) \mid \langle p_i, \gamma_i\rangle \in \mathbb{F}_{W'}\}$. For $p \in Q', \gamma \in \Gamma'$, let $c = \langle q_0, \gamma_0\rangle$, $\widehat{\mathit{MOVP}}(c, \langle p, \gamma\rangle, W') \sqsupseteq \widehat{\mathit{MOVP}}(c, \langle p, \gamma\rangle, W)$.*

Proof. By Definition 23, given $p \in Q, \gamma \in \Gamma$, any frontier set $\mathbb{F}_{W'}$ of W' can be decomposed into disjoint union $\mathbb{F}_{W'} = F_1 \uplus F_2$, where F_1 is some collection of dominators of $\langle p, \gamma\rangle$ and $F_2 \subseteq \{\langle p', \gamma'\rangle \mid \widehat{\mathtt{MOVP}}(\langle p', \gamma'\rangle, \langle p, \gamma\rangle, W') = \mathbf{0}\}$. The proof is done according to Lemma 1 and the fact that F_1 may not contain a dominator set of $\langle p, \gamma\rangle$.

By Theorem 3, the analysis of LEs with introducing summary transition rules will never cause any loss of precision, but can be not complete. The completeness is guaranteed by analysis of GUs.

Note that, the set of arguments $C_E = \{(\mathtt{arg}_k, C.\psi), (\mathtt{this}, C.\psi) \mid C.\psi \in \mathbf{P}_E\}$ from $\mathbf{P}_E$ is a witness of the frontier set $\mathbb{F}_{W'}$, where W' is the WPDS encoded from methods of $\mathbf{P}$ augmented with cached transition rules. Recall the example in Table 1, assume the partial model $\mathbf{P}$ taken in an LE consists of methods f_1 and f_2. We know $\mathbf{P}_E$ consists of f_1 only by definition. A loss of precision would be incurred, if summary transition rules A_c are introduced to arguments in f_2.

6 Empirical Studies

We developed our analysis algorithms as a tool named Japot [2], which exploits Soot2.3.0 [16] for preprocessing from Java programs to Jimple codes, and the Weighted PDS Library as the model checking engine. We perform experiments on Java applications from the Ashes benchmark suite [15] and the influential DaCapo benchmark suite [2] (Table 6). These applications are de facto benchmarks when investigating Java points-to analysis. We target on the newest version of DaCapo benchmark which requires JDK 1.5 or above, and stable Ashes benchmarks for which JDK 1.3 suffices. In sequel, the performance of Japot is measured by *call graph generation* in terms of the number of reachable methods, which is given in the "# Reachable Methods" column and these numbers take into account libraries used by each benchmark. Benchmarks on which the back-end model checker runs out of memory are not shown. All experiments were performed on a Mac OS X v.10.5.2 with a Xeon 2×2.66 GHz Dual-Core processor, and 4GB RAM. Only one processor is used in the following experiments.

Table 6. Benchmark Statistics and Call Graph Generation

Benchmark	# Reachable Methods			# Statements	# Suite	# JDK
	`CHA`	`Japot`	Prec.↑	Japot		
soot-c	5460	5079	7%	83936		
sablecc-j	13,055	9004	31%	143140	Ashes	JDK 1.3.1_01
antlr	10,728	×				
bloat	12,928	11090	14%	194063		
chart	30,831	×				
jython	14,603	×				
pmd	12,485	×			Dacapo	JDK 1.5.0_13
hsqldb	9983	8394	16%	142629		
xalan	9977	8392	16%	141405		
luindex	10,596	8961	15%	152592		
lusearch	11,190	9580	14%	163958		
eclipse	12,703	×				

The sub-column titled `CHA` gives the number of reachable methods by the CHA (Class Hierarchy Analysis) of Spark in soot-2.3.0. The sub-column titled `Japot` gives results computed by our context-sensitive PTA and the "# Statements" column gives the number of Jimple statements that `Japot` analyzed. The "Prec.↑" sub-column shows how much improvement on precision can be obtained by Japot over CHA. Our proposals regarding program modelling alone does not yield high scalability. Applying *type filtering* on-the-fly as usual and ignoring differences of string constants are also essential to the scalability.

[2] As an approximation, return variables from any native methods and reflection calls can point to objects whose type allows, and a throw exception can be handled by any exception handler whose declared type allows as an over-approximation.

Table 7. An Acceleration on Efficiency

Benchmark	# $(\texttt{LE} \circ \texttt{GU})^*$ (sec.)	# $\texttt{GU}^*$ (sec.)	# Acceleration
soot-c	656	1591	59%
sablecc-j	1547	2785	44%
bloat	12339	41434	70%
hsqldb	1205	2910	59%
xalan	1321	2926	55%
luindex	1514	3880	61%
lusearch	1757	4057	57%

We studied the efficiency improvement of $(\texttt{LE} \circ \texttt{GU})^*$ over $\texttt{GU}^*$, and initial results are given in Table 7 [3]. The "# $(\texttt{LE}\circ\texttt{GU})^*$" column and the "# $\texttt{GU}^*$" column gives the time in seconds of performing these iterative schemes respectively. The "# Acceleration" column shows an acceleration in terms of $\frac{|\texttt{GU}^*|-|(\texttt{LE}\circ\texttt{GU})^*|}{|\texttt{GU}^*|}$, which shows that $(\texttt{LE} \circ \texttt{GU})^*$ is $2.5X$ faster in average than $\texttt{GU}^*$. We expect novel strategies of taking LEs can improve the practical efficiency even more.

7 Related Work

One of the pioneer work is Andersen's PTA for C [1]. It is a subset-based, flow-insensitive analysis encoded as constraint solving problems, such that object allocations and pointer assignments are described by subset constraints, e.g. $x = y$ induces $pta(y) \subseteq pta(x)$. The scalability of Andersen's analysis has been greatly improved by more efficient constraint solvers. Andersen's analysis was introduced to Java by using annotated constraints [12].

The first scalable cloning-based context-sensitive Java PTA is presented in [17], in which programs and analysis problems are encoded as rules in logic program Datalog. Calling contexts are cloned after merging loops as equivalent classes. The BDD (Binary Decision Diagram) based implementation, as well as approximation by collapsing recursions, enable the analysis to scale. As discussed in [6], there are usually rich and large loops within the call graph, and the loss of precision is potentially incurred after approximating recursions.

Reps, et al. present a general framework for program analysis based on CFL-reachability [10], in which a PTA for C is shown by formulating pointer assignments as productions of context-free grammars. Inspired by this work, Sridharan, et al. formulated Andersen's analysis for Java [14] as balanced-parentheses problems regarding field read and write. A novel refinement-based analysis [13] is based on context-insensitive analysis and recovers the precision on-demand by removing imprecise propagation of points-to sets as violating a grammar for

[3] Note that,data structures and program states cannot be shared between soot (in Java) and the back-end model checker (in C). These numbers include DISK IO time for exchanging information between these two parts via files.

balanced parentheses, regarding both heap access and method calls. It shows good precision and scalability with respect to downcast safety analysis.

SPARK[5] is a widely-used test-bed for experimenting with Java PTA. It supports both equality and subset-based analysis, provides various algorithms for call graph construction, such as CHA, RTA(Rapid Type Analysis), and on-the-fly algorithms, as well as variations on field-sensitivity. The BDD-based implementation of the subset-based algorithms further improves the efficiency

One stream of research examines calling contexts in terms of sequences of objects on which methods are invoked, called *object-sensitivity* [9]. Similar to call-site strings based approach, the sequence of receiver objects can be unbounded and demands proper approximations, like k-CFA. [6] indicates that object-sensitivity excels at precision and is more likely to scale. Last but not the least, existing practiced Java PTA as discussed above, are cloning-based for context-sensitivity and have restrictions on handling recursive procedure calls.

In contrast to points-to analysis with call graph constructed on-the-fly, an ahead-of-time points-to analysis is proposed as one run of weighted pushdown model checking [7]. The notion of valid paths are enriched with further obeying to the Java semantics on dynamic dispatch. In particular, invalid control flows that violate Java semantics on dynamic dispatch are detected as those carrying conflicted data flows. The analysis enjoys context-sensitivities regarding both call graph construction and valid paths.

Last but not least, WPDSs are extended to conditional weighted pushdown systems (CWPDSs), by further associating each transition rule with a regular language that specifies conditions under which the transition rule can be applied [8]. There are wider applications of CWPDs when analyzing programs with objected-oriented features, for which WPDSs are not precise enough under a direct application. It is also shown that, the model checking problem on CWPDSs can be reduced to model checking problems on WPDSs.

8 Conclusions

We presented a scalable stacking-based context-sensitive points-to analysis for Java. The algorithm is derived as model checking problems on WPDSs, and no restriction is placed on (recursive) procedure calls. A two-staged iterative procedure is further proposed to effectively accelerate the analysis, supported by introducing summary transition rules. Our new iteration schemes shows the potential of an incremental points-to analysis, and we are extending the current setting with performing local explorations only.

9 Acknowledgments

This research is supported by STARC (Semiconductor Technology Academic Research Center). We thank Dr. Stefan for help us using the Weighted PDS Library. We also thank anonymous reviewers for their valuable comments.

References

1. L. Andersen. Program analysis and specialization for the c programming language. *PhD thesis*, 1994.
2. S. M. Blackburn, R. Garner, C. Hoffman, and et.al. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006.
3. A. Lal and T. W. Reps. Improving pushdown system model checking. In *18th International Conference on Computer Aided Verification, CAV06*, volume 4144 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 2006.
4. A. Lal and T. W. Reps. Solving multiple dataflow queries using wpdss. In *15th International Symposium on Static Analysis, SAS 2008*, volume 5079 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2008.
5. O. Lhoták and L. Hendren. Scaling java points-to analysis using spark. In *Proceedings of the 12th International Conference on Compiler Construction*.
6. O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th International Conference*, volume 3923 of *LNCS*, pages 47–64, Vienna, Mar. 2006. Springer.
7. X. Li and M. Ogawa. An ahead-of-time yet context-sensitive points-to analysis for Java. In *Proceedings of BYTECODE'09. ENTCS17798*, York, Mar. 2009. Elsevier.
8. X. Li and M. Ogawa. Conditional weighted pushdown systems and applications. In *Proceedings of PEPM'10*, to appear.
9. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.
10. T. Reps. Program analysis via graph reachability. In *ILPS '97: Proceedings of the 1997 international symposium on Logic programming*, pages 5–19, Cambridge, MA, USA, 1997. MIT Press.
11. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.
12. A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. *SIGPLAN Not.*, 36(11):43–55, 2001.
13. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. volume 41, pages 387–400, New York, NY, USA, 2006. ACM.
14. M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for java. *SIGPLAN Not.*, 40(10):59–76, 2005.
15. R. Vallée-Rai. Ashes suite collection. *http://www.sable.mcgill.ca/ashes*.
16. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.
17. J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004)*, pages 131–144, 2004.
18. G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 225–236, New York, NY, USA, 2008. ACM.