Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 253 (2009) 31–46
www.elsevier.com/locate/entcs

An Ahead-of-time Yet Context-Sensitive Points-to Analysis for Java

Xin Li[1,2] Mizuhito Ogawa[3]

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

Abstract

Points-to analysis is a prerequisite of program verification and static analysis on Java programs. It is known that call graph is typically constructed on-the-fly when points-to analysis proceeds for a better precision. In this work, we propose an ahead-of-time yet context-sensitive points-to analysis for Java as all-in-one weighted pushdown model checking. The analysis is context-sensitive in the sense that, (i) method calls and returns match with each other (a.k.a., *valid paths*); and (ii) targets of dynamic dispatch are analyzed separately for different calling contexts (a.k.a., *context-sensitive call graph*). The insight of our approach is that, by encoding dataflow as weights, invalid control flows that violate Java semantics on dynamic dispatch are detected as those carrying conflicted dataflow. Our analysis is presented as field-sensitive and flow-sensitive. Flow-insensitivity is shown to be easily obtained as a hierarchy considering efficiency and concurrent behaviors. Due to the lack of control flow structure and the explicit stack-based design, program analysis on bytecode is not an easy matter. We implemented the analysis in the framework of Soot compiler, and utilized the Weighted PDS Library as the back-end analysis engine. The analysis works on Jimple, a typed three-address intermediate representation of bytecode supported by Soot. The results of the analysis can be encoded into the class file as attributes for the further analysis or verification on bytecode.

Keywords: Points-to Analysis, Weighted Pushdown Model Checking, Java

1 Introduction

Points-to analysis [3] for Java is to detect the set of heap objects, i.e., instances of classes or arrays, possibly referred to by reference variables at run-time. Many applications such as program understanding, program verification, and static analysis depend on points-to analysis to reason the underlying control/data flow of Java programs. Due to dynamic object-oriented features like *dynamic dispatch* [4], points-to

[1] We would like to thank anonymous reviewers for their valuable and thorough comments. This research is supported by the 21st Century COE program "Verifiable and Evolvable e-Society" of JAIST, funded by the Japanese Ministry of Education, Culture, Sports, Science and Technology.

[2] Email: li-xin@jaist.ac.jp

[3] Email: mizuhito@jaist.ac.jp

[4] In this paper, we limit our focus to single dynamic dispatch only. Multiple dynamic dispatch, e.g., reflection in Java, demands non-trivial extension and thus independent discussion.

doi:10.1016/j.entcs.2009.11.013

analysis is mutually dependent to call graph construction. Thus we have choices of constructing call graph either *on-the-fly* as the points-to sets of call site receivers are computed, or *ahead-of-time* based on syntactical information of the program such as CHA [21] and RTA (Rapid Type Analysis) [22]. The former essentially enjoys a higher precision and is the choice of most of points-to analysis algorithms.

This paper presents an ahead-of-time yet context-sensitive points-to analysis for Java as all-in-one WPDMC (weighted pushdown model checking). Though it is well understood that program analysis can be regarded as model checking of abstract interpretation [1], model checking based approach to a context-sensitive points-to analysis is not straightforward. We limit our focus to providing the following context-sensitivities in the analysis, such that (i) method calls and returns match with each other (a.k.a., *valid paths*), which is guaranteed by encoding the program as a pushdown system; and (ii) targets of dynamic dispatch are analyzed separately for different calling contexts (*a.k.a., context-sensitive call graph*). Our approach to (ii) is, by further encoding dataflow as weights, invalid control flows that violates Java semantics on dynamic dispatch are detected as those carrying conflicted dataflow. These context-sensitivities are recently shown to be crucial to a precise points-to analysis in practice [5,18], as illustrated by Example 1.1. Our analysis is also flow-sensitive and field-sensitive. Concerning efficiency and concurrent behaviors of Java programs, points-to analysis is typically designed as flow-insensitive. One smart idea is combining SSA (Static Single Assignment) and complete flow insensitivity [7]. We briefly discussed how to easily obtain flow-insensitivity as a hierarchy.

Example 1.1 We denote by o^l an abstract heap object that is allocated at the program line l, and by $\mapsto$ the mapping relation afterwards. An analysis will precisely compute $\{c \mapsto o^3, d \mapsto o^5\}$ if it obeys to valid paths, and will furthermore erroneously infer $\{c \mapsto o^5, d \mapsto o^3\}$ otherwise. An analysis will precisely compute $\{o^3.f \mapsto o^{15}, o^5.f \mapsto o^{20}\}$ if a context-sensitive call graph is constructed, and will furthermore erroneously infer $\{o^3.f \mapsto o^{20}, o^5.f \mapsto o^{15}\}$ otherwise.

```
1.  public class Main {
2.      public static void main(String[] args) {
3.          A a = new A();
4.          A c = foo(a);
5.          A b = new B();
6.          A d = foo(b);
7.      }
8.      public static A foo(A x) {
9.          x.set();
10.         return x; }
11. }
```

```
12. public class A {
13.     Object f;
14.     public void set() {
15.         this.f = new Integer(0);
16.     }
17. }
18. public class B extends A {
19.     public void set() {
20.         this.f = new String();
21.     }
22. }
```

Due to the lack of control flow structure and explicit operand stack-based design, static analysis on bytecode is not an easy matter. We thus design and implement the analysis as a sub-phase of the compilation procedure in the Soot framework. Soot is an open-source compilation/optimization framework for Java, which has been originally designed to simplify the process of developing new optimizations for Java bytecode and supports three kinds of intermediate representations of bytecode.

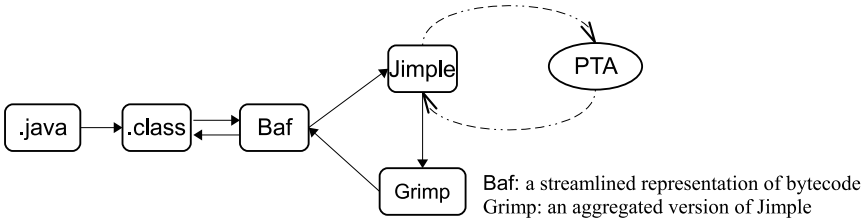


Fig. 1. Points-to Analysis on Jimple in the Soot Framework

As shown in Figure 1, the Java source code (bytecode) is firstly compiled into the Jimple [25] code, which is a typed three-address intermediate representation. Our analysis PTA is then performed on the Jimple code, and the analysis results can be encoded into the class file as attributes for any kind of later use. It can be useful for other occasions to perform complicated static analysis on an intermediate language like Jimple and annotate the class file with analysis results.

The remainder of the paper is organized as follows: Section 2 briefly introduces weighted pushdown model checking. Section 3 formalizes our abstraction and modelling on heap operations. Section 4 presents our ahead-of-time points-to analysis as all-in-one weighted pushdown model checking. The skeleton of holding soundness property is given, and the prototype implementation is shown. Section 5 compares related work and Section 6 concludes this paper with a discussion on future work.

2 Background

2.1 Weighted Pushdown Model Checking

Definition 2.1 A **pushdown system** $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown automaton regardless of input, where Q is a finite set of states called control locations, and Γ is a finite set of stack alphabet, and $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$ is a finite set of transition rules, and $q_0 \in Q$ and $w_0 \in \Gamma^*$ are the initial control location and stack contents respectively. We denote the transition rule $((q_1, w_1), (q_2, w_2)) \in \Delta$ by $\langle q_1, w_1 \rangle \hookrightarrow \langle q_2, w_2 \rangle$. A **configuration** of P is a pair $\langle q, w \rangle$, where $q \in Q$ and $w \in \Gamma^*$. Δ defines the transition relation $\Rightarrow$ between pushdown configurations such that if $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$, then $\langle p, \gamma\omega' \rangle \Rightarrow \langle q, \omega\omega' \rangle$, for all $\omega' \in \Gamma^*$.

A pushdown system is a finite transition system carrying an unbounded stack. A weighted pushdown system extends a pushdown system by associating a weight to each transition rule. The weights come from a bounded idempotent semiring.

Definition 2.2 A **bounded idempotent semiring** $S = (D, \oplus, \otimes, 0, 1)$ consists of a set D $(0, 1 \in D)$ and two binary operations $\oplus$ and $\otimes$ on D such that

(i) $(D, \oplus)$ is a commutative monoid with 0 as the unit element, and $\oplus$ is idempotent, i.e., $a \oplus a = a$ for $a \in D$;

(ii) $(D, \otimes)$ is a monoid with 1 as the unit element;

(iii) $\otimes$ distributes over $\oplus$, i.e., $\forall a, b, c \in D$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$;

(iv) $\forall a \in D, a \otimes 0 = 0 \otimes a = 0$;

(v) The partial ordering $\sqsubseteq$ is defined on D such that $\forall a, b \in D, a \sqsubseteq b$ iff $a \oplus b = a$, and there are no infinite descending chains on D wrt $\sqsubseteq$.

Remark 2.3 As stated in Section 4.4 in [8], the distributivity of $\oplus$ can be loosened to $a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$. The associativity of $\otimes$ can be loosened too, as long as both $(a \otimes b) \otimes c$ and $a \otimes (b \otimes c)$ conservatively approximates the program execution when applied to program analysis.

Definition 2.4 A **weighted pushdown system** is a triple $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, and $f : \Delta \to D$ is a function that assigns a value from D to each rule of P.

Definition 2.5 Consider a weighted pushdown system $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, and $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring. Assume $\sigma = [r_0, ..., r_k]$ to be a sequence of pushdown transition rules, where $r_i \in \Delta (0 \leq i \leq k)$, and $v(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$. Let path($c$,$c'$) be the set of all rule sequences that transform configurations from c into c'. The **generalized pushdown reachability problem**(GPR) is to find

$$\delta(c, C) = \bigoplus \{v(\sigma) | \sigma \in path(c, c'), c' \in C\}$$

for $c \in Q \times \Gamma^*$ and a set $C (\subseteq Q \times \Gamma^*)$ of regular configurations.

The GPR can be easily extended to answer the classic "meet-over-all-valid-paths" problem in program analysis. Efficient algorithms for solving GPR are developed based on the property that the regular set of pushdown configurations is closed under forward and backward reachability [8]. There are two off-the-shelf implementations of weighted pushdown model checking algorithms, Weighted PDS Library [5], and WPDS+ [6]. We apply the former as the back-end analysis engine in the prototype implementation.

2.2 Program Analysis as WPDMC

When designing a program analysis as WPDMC, the intuition behind $\otimes$ and $\oplus$ is:

- A weight function models a transfer function which typically represents the data flow changes for one-step program execution;
- $f \oplus g$ represents the merging of data flow at the meet of two control flows;
- $f \otimes g$ represents the sequential composition of abstract state transformers;
- **1** implies that an execution step does not change the program state; and

[5] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/
[6] http://www.cs.wisc.edu/wpis/wpds++/

Table 1
Syntactical Notations for References

f	$\in$	$\mathcal{F}$	Name constants of instance fields
v	$\in$	RefVar	Local variables and static fields
$v[i]$	$\in$	RefArr	Array references ($i \in \mathbb{Z}_0^+$)
v, $v[i]$, $v.f$	$\in$	$\mathcal{V}_{ref}$	RefVar $\cup$ RefArr $\cup$ RefVar $\times$ $\mathcal{F}$
v, $o[i]$, $o.f$	$\in$	$\mathcal{V}_{diref}$	RefVar $\cup$ $\mathcal{O} \times \mathbb{Z}_0^+ \cup \mathcal{O} \times \mathcal{F}$

- **0** implies that the program execution is interrupted by an error.

Recall the usual encoding of programs as finite model checking, *program states*, i.e., the product of global variables, local variables and program execution points, are encoded as states of finite automata. For pushdown model checking, the pushdown stack can simulate the runtime stack of program execution. For instance, the pushdown stack can be encoded to store calling contexts for procedure calls, just like the program execution on stack machine. In this paper, we will follow the convention defined in Definition 2.6.

Definition 2.6 Define an interprocedural control flow graph $G = (N, E, n_0)$, where $N = N_i \cup N_c \cup N_e$ is the set of nodes, with N_i, N_c, N_e as the sets of internal nodes, call sites, and method exits, respectively. $E = E_i \cup E_c \cup E_e$ is the set of edges with $E_i \subseteq N_i \times N, E_c \subseteq N_c \times N_i, E_e \subseteq N_e \times N_i$, where E_i, E_c, and E_e are the sets of internal edges, call edges, and return edges, respectively. $n_0 \in N$ is the unique entry node of G. We denote by $N_r \subseteq N_i$ the set of return points of method calls. Let $\texttt{assign} : N_c \to N_r$ be the function that associates with each call site from N_c with a distinguished return point in N_r, $N_r = \{n_r \mid n_r = \texttt{assign}(n_c), n_c \in N_c\}$.

Definition 2.7 The encoding of an interprocedural control flow graph $G = (N, E, n_0)$ as a pushdown system $P = (Q, \Gamma, \Delta, q_0, w_0)$ is defined as follows

- Q is a singleton set denoted by $\{\cdot\}$;
- $\Gamma = N$ with $w_0 = n_0$;
- Δ is constructed as follows,

$$\langle \cdot, n_i \rangle \hookrightarrow \langle \cdot, n_i' \rangle \qquad \text{if } (n_i, n_i') \in E_i$$

$$\langle \cdot, n_i \rangle \hookrightarrow \langle \cdot, n_c n_r \rangle \qquad \text{if } (n_i, n_c) \in E_c, \text{ and } n_r = \texttt{assign}(n_c) \in N_r$$

$$\langle \cdot, n_e \rangle \hookrightarrow \langle \cdot, \epsilon \rangle \qquad \text{if } (n_e, n_i) \in E_e$$

3 Modelling and Abstraction

3.1 Semantics of Heap Operations

Definition 3.1 Define $\mathcal{O}$ be the set of **heap objects** in the concrete domain, where $\top_o \in \mathcal{O}$ is the greatest element and represents any objects; $\bot_o \in \mathcal{O}$ is the least element and represents no objects (i.e., *null* reference). Elements in $\mathcal{O}$ except $\top_o$ and $\bot_o$ are incomparable.

We take Jimple, a three-address intermediate representation of Java, as our

target language, since it is syntactically much simpler than either Java or Bytecode. Table 1 prepares notations for, (i) the set of references $\mathcal{V}_{ref}$ that is syntactically allowed in Jimple; and (ii) the set of references $\mathcal{V}_{diref}$ in the semantic domain of *heap environments* (Definition 3.2). Static fields are treated in the same way with local variables, since they can be syntactically identified as well and we limit our focus to single-thread Java programs in the presentation.

Definition 3.2 A **heap environment** $\mathtt{henv}$ is a mapping from $\mathcal{V}_{diref}$ to $\mathcal{O}$. The set of heap environments is denoted by Henv^{con}. The evaluation function $\mathtt{eval}^{con} : \text{Henv}^{con} \to \mathcal{V}_{ref} \to \mathcal{O}$ on reference variables is defined as:

$$\mathtt{eval}^{con}(\mathtt{henv},\, v) = \mathtt{henv}(v)$$

$$\mathtt{eval}^{con}(\mathtt{henv},\, v[i]) = \mathtt{henv}(\mathtt{henv}(v)[i])$$

$$\mathtt{eval}^{con}(\mathtt{henv},\, v.f) = \mathtt{henv}(\mathtt{henv}(v).f)$$

Let $\mathtt{h}_{init}$ be the initial heap environment such that, for each $r \in \mathcal{V}_{diref}$, $\mathtt{eval}^{con}(\mathtt{h}_{init}, r) = \bot_o$ (*null* reference).

Let $\mathtt{Loc}$ be the set of program locations. Since we only consider single thread Java program here, the next program location at each execution step is uniquely determined. We informally refer it as $\mathtt{next}(l)$ for $l \in \mathtt{Loc}$. Later it will be discussed how to leverage the analysis to a flow-insensitive counterpart regarding concurrency.

Definition 3.3 Define an transition system $\mathcal{OS} = (States, \mathtt{s}_{init}, \to)$ to represent the Java semantics on heap, where

- $States \subseteq (\mathtt{Loc} \times \text{Henv}^{con})$ is a set of pairs of a program location and a heap environment,
- $\mathtt{s}_{init}$ is the initial state, which is a pair of the program entry l_0 and $\mathtt{h}_{init}$;
- $\to \;\subseteq States \times States$ is the set of operational semantic rules, and $\to^*$ denotes the transitive closure of $\to$.

A transition rule $\langle l, \mathtt{henv}\rangle \to \langle \mathtt{next}(l), \tau(\mathtt{henv})\rangle$ for typical pointer assignment statements at $l \in \mathtt{Loc}$ is shown in Table 2, where

- the function $\nu(\mathtt{henv}, \mathrm{T})$ generates a fresh heap object of type T in $\mathcal{O}$; and
- for $r,\ r' \in \mathcal{V}_{diref}$, $o \in \mathcal{O}$,

$$(\mathtt{henv} \odot [r \mapsto o])r' = \begin{cases} o & \text{if } r = r' \\ \mathtt{henv}(r) & \text{otherwise} \end{cases}$$

Definition 3.4 The **composition of heap environment transformers** is defined by the standard η-expansion, such that, for $\text{exph}_1, \text{exph}_2 \in \text{ExpHenv}$,

$$\begin{aligned}(\lambda\mathtt{henv}.\ \text{exph}_2) \circ (\lambda\mathtt{henv}.\ \text{exph}_1) &=_\eta \lambda\mathtt{h}.\ (\lambda\mathtt{henv}.\ \text{exph}_2)(\lambda\mathtt{henv}.\ \text{exph}_1)\mathtt{h} \\ &=_\beta \lambda\mathtt{h}.\ \text{exph}_2[\mathtt{henv} := \text{exph}_1[\mathtt{henv} := \mathtt{h}]]\end{aligned}$$

The notation $E[h := E']$ means the expression E with E' substituted for free occurrences of h.

Table 2
Heap Environment Transformer

Statement	Heap Environment Transformer
$x = \text{new T}$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \nu(\texttt{henv}, \text{T})]$
$x = y$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(y)]$
$x = y[n]$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(\texttt{henv}(y)[n])]$
$x[n] = y$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [\texttt{henv}(x)[n] \mapsto \texttt{henv}(y)]$
$x = y.f$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(\texttt{henv}(y).f)]$
$y.f = x$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [\texttt{henv}(y).f \mapsto \texttt{henv}(x)]$
$x = \text{return } y$	$\tau = \lambda\texttt{henv}.\texttt{henv} \odot [x \mapsto \texttt{henv}(y)]$
$x.m(r_0, ..., r_l)$	$\tau = \tau_0 \circ \tau_1 \circ ... \circ \tau_k$
	where $\tau_1, ..., \tau_k \in \texttt{Fun}$,
	$\texttt{Fun} = \{\lambda\texttt{henv}.\texttt{henv} \odot [\texttt{arg}_i \mapsto \texttt{henv}(m_i)] \mid$
	$r_i (0 \leq i \leq l)$ are arguments of reference type $\}$
	$\tau_0 = \lambda\texttt{henv}.\texttt{henv} \odot [\texttt{this} \mapsto \texttt{henv}(x)]$
	The method m to be invoked is from the class with which the
	type of $\texttt{henv}(x)$ is `compatible` with respect to the method m
	(Definition 3.5, i.e., the basic procedure of dynamic dispatch)

Definition 3.5 Let $\mathcal{M}$ be the set of method identifiers, and let $\mathcal{T}$ be the set of reference types. Let $\texttt{any} = \texttt{type}(\top_o)$ and $\texttt{none} = \texttt{type}(\bot_o)$. For $t, t' \in \mathcal{T} \setminus \{\texttt{any}, \texttt{none}\}$ and $m \in \mathcal{M}$, t' `conflicts` with t with respect to the method m if and only if, (i) $t' \neq t$ and t' does not inherit t, or (ii) t' inherits t with redefining m. Otherwise, we say t' is `compatible` with t with respect to the method m. Furthermore, t is `compatible` with `any`, for each t in $\mathcal{T}$ and vice versa; `none conflicts` with t, for each t in $\mathcal{T}$ and vice versa.

3.2 Abstraction

There are varieties of infinities to be abstracted away for a tractable analysis, such as the nesting of array structures, method invocations, field reference, and the number of allocated heap objects. We take the following abstractions in the analysis,

- An unique abstract heap object models objects allocated at each heap allocation site, and is identified by its type and program line number (Definition 3.6). Therefore, the number of abstract heap objects are syntactically bounded;
- The indices of arrays are ignored, such that members of an array are not distinguished. We denote by $[\![v]\!]$ the representative for array references $v[i]$. References with nested $[\![\_]\!]$ refer to multi-array access. We denote $\{[\![o]\!] \mid o \in \texttt{Obj}\}$ by $[\![\texttt{Obj}]\!]$.

Note that, after abstracting heap objects to be a finite set, the nesting of either field references or array references are correspondingly finite yet unbounded. Since local variables have a unique counterpart representation in the analysis, we will reuse notations in Table 1 afterwards when it is clear from the context.

Definition 3.6 Define **abstract heap objects** $\texttt{Obj} = \{\lceil t, l \rceil \mid t \in \mathcal{T}, l \in \texttt{Loc}\} \cup \{\top_{\texttt{obj}}, \bot_{\texttt{obj}}\}$, where $\top_{\texttt{obj}}$ is the greatest element and $\bot_{\texttt{obj}}$ is the least element. Other elements in `Obj` except $\top_{\texttt{obj}}$ and $\bot_{\texttt{obj}}$ are incomparable.

Definition 3.7 An **abstract heap environment** `henv` is a mapping from $\mathcal{V}_{diref}$ to $\mathcal{P}(\texttt{Obj})$, where $\mathcal{P}$ is the powerset operator. The set of **abstract heap environ-**

Table 3
Abstract Heap Environment Transformer ExpFun

Statement	Abstract Heap Environment Transformer
$x = \text{new T}$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \lceil t, l \rceil]$
$x = y$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(y)]$
$x = y[n]$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}([\![\texttt{henv}(y)]\!])]$
$x[n] = y$	$\lambda\texttt{henv}.\texttt{henv} \bullet [[\![\texttt{henv}(x)]\!] \mapsto \texttt{henv}(y)]$
$x = y.f$	$\lambda\texttt{henv}.\texttt{henv} \bullet [x \mapsto \texttt{henv}(\texttt{henv}(y).f)]$
$y.f = x$	$\lambda\texttt{henv}.\texttt{henv} \bullet [\texttt{henv}(y).f \mapsto \texttt{henv}(x)]$

ments is denoted by Henv^{abs}. The evaluation function $\texttt{eval}^{abs} : \text{Henv}^{abs} \to \mathcal{V}_{ref} \to \mathcal{P}(\texttt{Obj})$ on reference variables in the abstract domain is defined as:

$$\texttt{eval}^{abs}(\texttt{henv}, v) = \texttt{henv}(v)$$

$$\texttt{eval}^{abs}(\texttt{henv}, [\![v]\!]) = \{\texttt{henv}([\![o]\!]) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, v)\}$$

$$\texttt{eval}^{abs}(\texttt{henv}, v.f) = \{\texttt{henv}(o.f) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, v)\}$$

Let $\texttt{h}_{init}$ be the abstract initial heap environment such that $\texttt{eval}^{abs}(\texttt{h}_{init}, r) = \bot_{\texttt{obj}}$ for each $r \in \mathcal{V}_{ref}$.

Abstract heap environment transformers for typical pointer assignment statements are shown in Table 3, where $\lceil t, l \rceil \in \texttt{Obj}$, and for $r, r' \in \mathcal{V}_{diref}$, $o \in \texttt{Obj}$

$$(\texttt{henv} \bullet [r \mapsto o])r' = \begin{cases} \{o\} & \text{if } r = r' \notin [\![\texttt{Obj}]\!] \\ \texttt{henv}(r') \cup \{o\} & \text{if } r = r' \in [\![\texttt{Obj}]\!] \\ \texttt{henv}(r') & \text{otherwise} \end{cases}$$

4 Points-to Analysis as WPDMC

4.1 The Design of Weight Space

By encoding the program as a pushdown system, we are provided with context-sensitivity regarding valid pathes. To construct a context-sensitive call graph during the analysis, we enrich the notion of valid paths, such that valid paths that violate type requirements of dynamic dispatch are also regraded as invalid. By encoding dataflow as weights, an invalid control flow is detected as that carrying conflicted dataflow, and combining weights along the control flow will result in the weight **0**.

Definition 4.1 Define **abstract heap environment transformers** ExpFun as,

ExpFun	::=	$\lambda\texttt{henv}.$ ExpHenv
ExpHenv	::=	$\texttt{henv}$ | ExpHenv $\bullet$ ExpMap
ExpMap	::=	[Expf $\mapsto$ Expt]
Expf	::=	v | Expt.f | Arrf
Expt	::=	$\lceil t, l \rceil$ | $\texttt{henv}(v)$ | $\texttt{henv}(\text{Expt}.f)$ | Arrt
Arrf	::=	$[\![\text{Expt}]\!]$
Arrt	::=	$\texttt{henv}([\![\text{Expt}]\!])$

For this purpose, the basic weight functions, i.e., the abstract heap environment transformers (Definition 4.1), are extended by pairing path constraints (Definition 4.2). We denote by $(s, t \uparrow m)$ a path constraint $(s, t, m) \in \texttt{PathCons}$, which intends

Table 4
Abstract Heap Environment Transformer with Path Constraints

$x.m(r_0, ..., r_l)$	$(\tau_0 \circ \tau_1 \circ ... \circ \tau_k,\ \{(x, t \uparrow m)\})$ where $\tau_1, ..., \tau_k \in \texttt{Fun}$, $\texttt{Fun} = \{\lambda\texttt{henv.henv} \bullet [\texttt{arg}_i \mapsto \texttt{henv}(r_i)] \mid$ $r_i (0 \leq i \leq l)$ are arguments of reference type $\}$ $\tau_0 = \lambda\texttt{henv.henv} \bullet [\texttt{this} \mapsto \texttt{henv}(x)]$

that the dynamic dispatch of a call edge demands the runtime type of the heap object pointed to by s to be *compatible* with the type t w.r.t. the method m. This judgement on types should exactly obey to (such as Definition 3.5) or soundly approximates the Java semantics for dynamic dispatch.

Definition 4.2 Define a set of **path constraints** $\texttt{PathCons} \subseteq \mathcal{V} \times \mathcal{T} \times \mathcal{M}$, where $\mathcal{V} ::= v \mid \mathcal{V}.f \mid [\![\mathcal{V}]\!]$ is the set of references that syntactically allows nested field references and array structures.

The evaluation function $eval^{abs}$ is extended as $eval^{abs} : \text{Henv}^{abs} \to \mathcal{V} \cup \texttt{Obj} \to \mathcal{P}(\texttt{Obj})$, such that for $\texttt{henv} \in \text{Henv}^{abs}$, $o \in \texttt{Obj}$

$$\texttt{eval}^{abs}(\texttt{henv}, o) = \{o\} \qquad \texttt{eval}^{abs}(\texttt{henv}, v) = \texttt{henv}(v)$$

$$\texttt{eval}^{abs}(\texttt{henv}, \mathcal{V}.f) = \{\texttt{henv}(o.f) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, \mathcal{V})\}$$

$$\texttt{eval}^{abs}(\texttt{henv}, [\![\mathcal{V}]\!]) = \{\texttt{henv}([\![o]\!]) \mid o \in \texttt{eval}^{abs}(\texttt{henv}, \mathcal{V})\}$$

Definition 4.3 Define $\texttt{Ref}$: ExpFun $\to \mathcal{V} \to \mathcal{P}(\text{Expt})$ such that for $\tau = \lambda\texttt{henv}.$ ExpHenv $\bullet$ $[\texttt{vf} \mapsto \texttt{vt}]$ and $\tau' = \lambda\texttt{henv}.$ ExpHenv $\in$ ExpFun,

$$\texttt{Ref}(\tau, v) = \begin{cases} \{\texttt{vt}\} & \text{if } \texttt{vf} = v \notin \text{Arrf} \\ \texttt{Ref}(\tau', v) \cup \{\texttt{vt}\} & \text{if } \texttt{vf} = v \in \text{Arrf} \\ \texttt{Ref}(\tau', v) & \text{otherwise} \end{cases}$$

$$\texttt{Ref}(\tau, \mathcal{V}.f) = \{\texttt{Ref}(\tau, \texttt{vt}'.f) \mid \texttt{vt}' \in \texttt{Ref}(\tau, \mathcal{V})\}$$

$$\texttt{Ref}(\tau, [\![\mathcal{V}]\!]) = \{\texttt{Ref}(\tau, [\![\texttt{vt}']\!]) \mid \texttt{vt}' \in \texttt{Ref}(\tau, \mathcal{V})\}$$

Table 4 shows the abstraction of virtual method invocations. The heap environment transformer for the virtual call edge is paired with a singleton set, which specifies the expected runtime type t for the call site receiver to follow this call path. Transformers for other program statements are paired with an empty set $\emptyset$ initially.

Definition 4.4 Define $\texttt{Ref}^{-1} : \mathcal{P}(\text{Expt}) \to \mathcal{P}(\mathcal{V} \cup \texttt{Obj})$ such that $\texttt{Ref}^{-1}(\mathbb{V}) = \bigcup_{\texttt{vt} \in \mathbb{V}} \texttt{Ref}^{-1}(\{\texttt{vt}\})$ for $\mathbb{V} \subseteq \text{Expt}$, where

$$\texttt{Ref}^{-1}(\{\lceil t, l \rceil\}) = \{\lceil t, l \rceil\}$$

$$\texttt{Ref}^{-1}(\{\texttt{henv}(v)\}) = \{v\}$$

$$\texttt{Ref}^{-1}(\{\texttt{henv}(\text{Expt}.f)\}) = \{\texttt{Ref}^{-1}(\{\text{Expt}\}).f\}$$

$$\texttt{Ref}^{-1}(\{\texttt{henv}([\![\text{Expt}]\!])\}) = \{[\![\texttt{Ref}^{-1}(\{\text{Expt}\})]\!]\}$$

Definition 4.5 Define $\texttt{trace} : \text{ExpFun} \to \mathcal{V} \to \mathcal{P}(\mathcal{V} \cup \texttt{Obj})$ such that $\texttt{trace} = \texttt{Ref}^{-1} \circ \texttt{Ref}$.

Example 4.6 Let $\tau = \lambda\texttt{henv.henv} \bullet [x \mapsto \texttt{henv}(y)] \bullet [\texttt{henv}(z).f \mapsto o]$. Then $\texttt{trace}(\tau, x.f) = y.f$, $\texttt{trace}(\tau, z.f) = o$, and $\texttt{trace}(\tau, y) = y$.

Definition 4.7 Let $c \subseteq \texttt{PathCons}$ and $\tau \in$ Expfun. For $(s, t \uparrow m) \in c$,

$$\texttt{judge}(c, \tau) = \begin{cases} \texttt{error} & \text{if there exits } (s, t \uparrow m) \in c \text{ s.t.} \\ & \texttt{judge}(\{(s, t \uparrow m)\}, \tau) = \texttt{error} \\ \bigcup_{(s,t\uparrow m)\in c} \texttt{judge}(\{(s, t \uparrow m)\}, \tau) & \text{otherwise} \end{cases}$$

$$\texttt{judge}(\{(s, t \uparrow m)\}, \tau)$$
$$= \begin{cases} \texttt{error} & \text{if } \texttt{trace}(\tau, s) \subseteq \texttt{Obj} \text{ and for all } o \in \texttt{trace}(\tau, s), \\ & \quad \texttt{type}(o) \texttt{ conflicts with } t \text{ w.r.t. } m \\ \phi & \text{if there exists } o \in \texttt{trace}(\tau, s) \text{ for } o \in \texttt{Obj} \text{ and} \\ & \quad \texttt{type}(o) \text{ is } \texttt{compatible} \text{ with } t \text{ w.r.t. } m \\ \bigcup_{s' \in \texttt{trace}(\tau,s)} \{(s', t \uparrow m)\} & \text{otherwise} \end{cases}$$

Definition 4.7 defines judgements on path constraints when composing the extended weights.

- The first case says, `error` returns if the current abstract heap environment is known not to satisfy the path constraints on s. This case results in the weight $\mathbf{0}$ and the related control flow is thus excluded from the analysis result.
- The second case says, a known satisfied constraint will not be included into the newly generated path constraints for efficiency.
- The last case says, new path constraints are generated when the judgement on path constraints is pending at the moment.

Definition 4.8 Define a semiring $S_e = (D_e, \oplus_e, \otimes_e, 0_e, 1_e)$, such that

- $D_e = \mathcal{P}(\mathbb{D})$, where $\mathbb{D} = \{(f, c) \mid f \in ExpFun,\ c \subseteq \texttt{PathCons}\}$
- $0_e = \emptyset$ and $1_e = \{(\lambda\texttt{henv.henv}, \emptyset)\}$
- $w_1 \otimes_e w_2 = \{\ p_1 \otimes p_2 \mid p_1 = (\texttt{func}_1, c_1) \in w_1, p_2 = (\texttt{func}_2, c_2) \in w_2\ \}$

$$(\texttt{func}_1, c_1) \otimes (\texttt{func}_2, c_2) = \begin{cases} 0_e & \text{if } \texttt{jpc} = \texttt{error} \\ (\texttt{func}_2 \circ \texttt{func}_1, c_1 \cup \texttt{jpc}) & \text{otherwise} \end{cases}$$

  where $\texttt{jpc} = \texttt{judge}(c_2, \texttt{func}_1)$, $w_1, w_2 \in D_e$.
- $w_1 \oplus_e w_2 = w_1 \cup w_2$ for $w_1, w_2 \in D_e$

Remark 4.9 Both the associativity of $\otimes$ and the distributivity of $\oplus$ over $\otimes$ hold. Since the nesting of field references and array structures is finite yet unbounded, a bound can be given on their nested depth for efficiency. That is,

a field or array reference nested deeper than the given bound will be regarded as pointing to anywhere(i.e., $\top_{\texttt{obj}}$), as illustrated in Example 4.10. As a result, $(w_0 \otimes w_1) \otimes w_2 \sqsubseteq w_0 \otimes (w_1 \otimes w_2)$ for $w_0, w_1, w_2 \in \mathbb{D}$.

Example 4.10 If we limit the nesting of field references to the depth 1, the analysis of the Java code fragment "$x.f = w;\ y = x.f;\ z = y.g;$" returns the weight $\lambda\texttt{henv.henv} \bullet [\texttt{henv}(x).f \mapsto \texttt{henv}(w)] \bullet [y \mapsto \texttt{henv}(w)] \bullet [z \mapsto \texttt{henv}(\texttt{henv}(w).g)]$ by $(w_0 \otimes w_1) \otimes w_2$, and $\lambda\texttt{henv.henv} \bullet [\texttt{henv}(x).f \mapsto \texttt{henv}(w)] \bullet [y \mapsto \texttt{henv}(w)] \bullet [z \mapsto \top_{\texttt{obj}}]$ by $w_0 \otimes (w_1 \otimes w_2)$.

Remark 4.11 The points-to analysis presented above is flow-sensitive. It is easy to obtain parameterized flow-sensitivity as a hierarchy by loosening the following dimensions in the weight space design, (i) whether the points-to target of a reference is changed by a new assignment on it. For this purpose, $\bullet$ is reinterpreted as the union extension on maps for all references; and (ii) whether the ordering of the composition of heap environment transformers is kept on a sequence of program codes. Apart from (i), the $\otimes_e$ operation on weights w_1, w_2 is extended as $w_1 \otimes_e w_2 = \{\lambda\texttt{henv.henv} \bullet p_1 \otimes p_2(\texttt{henv}) \bullet p_2 \otimes p_1(\texttt{henv}) \mid p_1 \in w_1, p_2 \in w_2\}$.

Definition 4.12 For a program starting with the entry point $l_0 \in \texttt{Loc}$, let $W = (P, S_e, f)$ be the weighted pushdown system encoded from it by Definition 2.6, and let $\texttt{Ret}$ be the set of return points introduced for method invocations. The points-to analysis on the reference $r \in \mathcal{V}_{ref}$ at the program point $l \in \texttt{Loc} \cup \texttt{Ret}$ is defined as

$$\texttt{pta}(r, C) = eval^{abs}(\delta(c, C)(\texttt{h}_{init}), r)$$

where $\delta(c, C)$ is from Definition 2.5 with $c = \langle \cdot, l_0 \rangle$ and $C = \langle \cdot, l.(\texttt{Ret})^* \rangle$.

We take $C = \langle \cdot, l.(\texttt{Ret})^* \rangle$ to represent all possible pushdown configurations as an approximation, when l is the top-most stack symbol. Therefore, $\texttt{pta}$ computes points-to information along all paths leading from the program's entry point to the program point l of concern.

4.2 Soundness

Since $\oplus$ operation conservatively combines all possible dataflow in the analysis, we turn to the following two steps to show that our analysis is sound (Theorem 4.18), (i) the analysis on any sequential execution path infers sound points-to results based on abstract interpretation (Theorem 4.16), and (ii) if some control flow is removed during the analysis, it is invalid indeed in the concrete execution, which is witnessed by Lemma 4.17.

Definition 4.13 Let $\texttt{type} : \mathcal{O} \to \mathcal{T}$ and $\texttt{loc} : \mathcal{O} \to \texttt{Loc}$ be functions that return the type and the allocation site of a heap object, respectively. The **abstraction on heap objects** $\alpha : \mathcal{O} \to \texttt{Obj}$ is defined as follows,

- $\alpha(o) = (t, l)$ for $o \in \mathcal{O} \setminus \{\top_o, \bot_o\}$, $t = \texttt{type}(o) \in \mathcal{T}$, $l = \texttt{loc}(o) \in \texttt{Loc}$; and
- $\alpha(\top_o) = \top_{\texttt{obj}}$ and $\alpha(\bot_o) = \bot_{\texttt{obj}}$

The concretization is denoted by $\gamma = \alpha^{-1} : \texttt{Obj} \to \mathcal{P}(\mathcal{O})$. The powerset extensions of α and γ are denoted by $\alpha_o : \mathcal{P}(\mathcal{O}) \to \mathcal{P}(\texttt{Obj})$ and $\gamma_o : \mathcal{P}(\texttt{Obj}) \to \mathcal{P}(\mathcal{O})$.

Definition 4.14 For the program entry l_0 and the program point $l \in \texttt{Loc} \cup \texttt{Ret}$, let $\langle l_0, \texttt{h}_{init}\rangle \to^* \langle l, \texttt{henv}\rangle$. For $r \in \mathcal{V}_{ref}$ at l and $C = \langle \cdot, l.(\texttt{Ret})^*\rangle$, $\texttt{pta}(r, C)$ is *sound* if $\alpha_0(\texttt{eval}^{con}(\texttt{henv}, r)) \subseteq \texttt{pta}(r, C)$.

Definition 4.15 For abstract environment transformers $f_1, f_2 \in \text{ExpFun}$, $x \in \mathcal{V}_{ref}$ and $\texttt{henv} \in \text{Henv}^{abs}$, $f_1 \succcurlyeq f_2$ if $\texttt{eval}^{abs}(f_1(\texttt{henv}), x) \supseteq \texttt{eval}^{abs}(f_2(\texttt{henv}), x)$.

Theorem 4.16 *For a Jimple statement $s \in Stmt$, let f be the heap environment transformer of s, and f^{abs} be the abstract heap environment transformer of s. Then, $f^{abs} \succcurlyeq \alpha_o \circ f \circ \gamma_o$.*

Theorem 4.16 is proved by a case analysis on the Jimple statement s.

Lemma 4.17 *For $w_1, w_2 \in \mathbb{D}$, and $(\tau, c) \in w_1, (\tau', c') \in w_2$, $\texttt{henv} \in \text{Henv}^{abs}$, and $(s, t \uparrow m) \in c'$,*

$$\bigcup_{s' \in \texttt{trace}(\tau, s)} \texttt{eval}^{abs}(\texttt{henv}, s') \supseteq \texttt{eval}^{abs}(\tau(\texttt{henv}), s)$$

Lemma 4.17 says that, the result of back-tracing by `trace` soundly comprise all the contributed path constraints. As illustrated in Figure 2, the analysis is performed on an operational transition sequence, where "$\bullet \longrightarrow \bullet$" represents an operational transition in one step, and w_1 and w_2, respectively, denote the resulting weight by composing transitions marked with the dotted line. By Lemma 4.17, to check the points-to targets of s on $\tau(\texttt{henv})$ amounts to check that of all $s' \in \texttt{trace}(\tau, s)$ on $\texttt{henv}$.

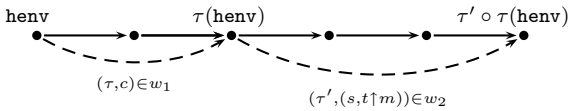


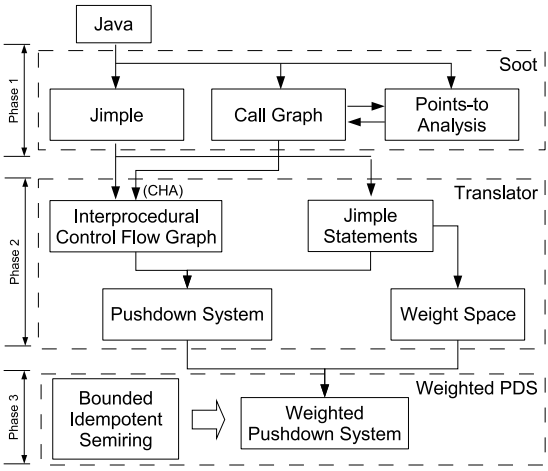
Fig. 2. Sound Tracing on Path Constrains

Theorem 4.18 *(**Soundness**) For $r \in \mathcal{V}_{ref}$ at $l \in \texttt{Loc} \cup \texttt{Ret}$ and $C = \langle \cdot, l.(\texttt{Ret})^*\rangle$, $\texttt{pta}(r, C)$ is sound.*

4.3 Prototype Implementation

We implemented the analysis algorithm as a prototype in the Soot framework. As shown in Figure 3, it starts off preprocessing from Java sources (or bytecode) to Jimple codes by Soot. Soot provides facilities of call graph construction and points-to analysis at various levels of precision. We borrow the most imprecise analysis CHA (Class Hierarchy Analysis) [21] to produce a preliminary call graph for the ahead-of-time analysis. A weighted pushdown system designed for the ahead-of-time points-to analysis is then constructed from the Jimple code. The analysis is finally performed by calling the Weighted PDS Library on the model, during

which the invalid control flows are removed from the analysis results on-demand. Note that, during the encoding of programs as a weighted pushdown system, extra variables will be introduced in RefVar to denote *formal parameters* and *return values* of reference type. For program statements whose execution does not change heap states, their corresponding heap environment transformers are thus identity function λ`henv`.`henv`, such as the conditional branching statement.




```
f0:
[(scope0_type0_args -> scope0_type0_arg_0);
(scope0_type1_$r0 -> A:2@1);
(scope0_type1_a -> A:2@1);
(scope0_type1_c -> A:2@1);
(scope0_type2_$r1 -> B:21@2);
(scope3_this -> B:21@2);
(scope3_type2_this -> B:21@2);
(scope1_this -> B:21@2);
(scope1_type1_this -> D:21@2);
(scope0_type1_b -> B:21@2);
(scope2_type1_arg_0 -> B:21@2);
(scope2_type1_x -> B:21@2);
(scope5_this -> B:21@2);
(scope5_type1_this -> B:21@2);
(scope5_type4_$r0 -> java.lang.Integer:null@3);
(r_ref -> B:21@2);
(scope0_type1_d -> B:21@2);
(scope0_type3_$r2 -> scope4_type3_out);
(scope0_type3_$r3 -> scope4_type3_out);
(A:2@1.scope7_type5_f -> java.lang.Integer:null@3);
(B:21@2.scope7_type5_f -> java.lang.Integer:null@3);
f1:
[(scope0_type0_args -> scope0_type0_arg_0);
(scope0_type1_$r0 -> A:2@1);
(scope0_type1_a -> A:2@1);
(scope5_this -> A:2@1);
(scope5_type1_this -> A:2@1);
(scope5_type4_$r0 -> java.lang.Integer:null@3);
(scope0_type1_c -> A:2@1);
(scope0_type2_$r1 -> B:21@2);
(scope3_this -> B:21@2);
(scope3_type2_this -> B:21@2);
(scope1_this -> B:21@2);
(scope1_type1_this -> D:21@2);
(scope0_type1_b -> B:21@2);
(scope2_type1_arg_0 -> B:21@2);
(scope2_type1_x -> B:21@2);
(scope6_this -> B:21@2);
(scope6_type2_this -> B:21@2);
(scope6_type0_$r0 -> java.lang.String:null@4);
(r_ref -> B:21@2);
(scope0_type1_d -> B:21@2);
(scope0_type3_$r2 -> scope4_type3_out);
(scope0_type3_$r3 -> scope4_type3_out);
(A:2@1.scope7_type5_f -> java.lang.Integer:null@3);
(B:21@2.scope7_type5_f -> java.lang.String:null@4);
```

Fig. 3. The Prototype Framework and Running Profiles

Definition 3.5 defines rules for judging whether a type t `conflicts` or `compatible` with a type t'. For simplicity at the first stage, the case (ii) is not provided in the prototype implementation and will be included in the later version. The right-hand-side of Figure 3 shows the points-to result of analyzing Example 1.1. The analysis returns two abstract heap environment transformers. f_1 gives the precise dataflow summary of this program following the control flow "$4 \to 9 \to 15 \to 6 \to 9 \to 20$", which precisely infers that $\{o^3 \mapsto o^{15}, o^5 \mapsto o^{20}\}$. f_0 is a dataflow summary of the invalid call path "$6 \to 9 \to 15$" due to excluding the case (ii) above.

5 Related Work

Points-to analysis for Java has been an active field over the past decade. We limit our discussion primarily to recent advances especially related to context-sensitive points-to analysis.

One of the pioneer work in this field is Andersen's points-to analysis for C [13]. It is a subset-based, flow-insensitive analysis implemented via constraint solving, 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 [14,15]. Andersen's analysis was introduced to Java by using annotated constraints [16].

Reps, et al. present a general framework for program analysis based on CFL-reachability [11]. A points-to analysis for C is shown by formulating pointer assignments as productions of context-free grammars. Borrowing this view, Sridharan, et al. formulated Andersen's analysis for Java in a demand-driven manner [17]. The analysis targets on applications with small time and memory budgets. A key insight of their algorithm is that a field read action is supposed to be preceded by a field write action, so-called balanced-parentheses problem. An improved context-sensitive analysis is later proposed by refining call paths as a balanced-parentheses problem as well [18]. The lost precision is retained by further refinement procedures. The demand-driven strategy, as well as the refinement-based algorithm makes this analysis scale.

A scalable context-sensitive points-to analysis for Java is presented in [19]. Programs and analyses are encoded as the set of rules in logic programs Datalog. The context-sensitivity is obtained by cloning a method for each calling context, and by regarding loops as equivalent classes. The BDD (Binary Decision Diagram) based implementation, as well as approximation by collapsing recursions, make the analysis scale. As shown in [5], there are usually rich and large loops within the call graph, and thus much precision is lost by collapsing loops.

Spark[23] is a widely-used testbed for experimenting with points-to analysis for Java. It supports both equality and subset-based analysis, provides various algorithms for call graph construction (such as CHA, RTA, and an on-the-fly algorithm), and enables variations on field-sensitivity. The BDD-based implementation of the subset-based algorithms further improves the efficiency of operations on points-to sets [24]. Our analysis also borrows its CHA for a preliminary call graph. A recent empirical study compares precision of subset-based points-to analyses with various abstractions on context-sensitivity [5].

One stream of research examines calling contexts in terms of sequences of objects on which methods are invoked, called *object-sensitivity* [20]. Similar to call-site strings based approach, the sequence of receiver objects can be unbounded and demands proper approximations, like k-CFA [6]. [5] also concludes that a context-sensitive points-to analysis in terms of object-sensitivity excels at precision and is even more likely to scale by experimental studies.

Concerning scalability for context-sensitive points-to analysis, some analysis utilizes BDD as the underlying data structure [23,19], others only compute results that sufficiently meet the client's needs, so-called *client-driven* and *demand-driven* manner [4,18]. These strategies are also applicable to our analysis in this paper.

6 Conclusions

This paper presents context-sensitive points-to analysis for Java as all-in-one weighted pushdown model checking. The notion of valid paths are enriched such

that dataflow along each valid path need further satisfy type requirements for dynamic dispatch. The ahead-of-time analysis is formalized as one run of weighted pushdown model checking, which enjoys context-sensitivities regarding both call graph and valid paths. The proposed points-to analysis is implemented as a prototype, with Soot as the preprocessor from Java to Jimple and Weighted PDS library as the model checking engine.

The time complexity in general case specific to our analysis is $\Theta(|\Delta| \cdot |\mathbb{D}| \cdot |T_{\oplus}| \cdot |T_{\otimes}|)$. $|\mathbb{D}|$ is the cardinality of the weight space. $|\Delta|$ is up to the program size by encoding. $|T_{\oplus}|$ and $|T_{\otimes}|$ are the prices for each weight operation. At present, the tentative experiments are restricted to small examples, due to the weight package is implemented based on linked list for a fast prototyping. Our next step is to prepare a weight package based on CrocoPat [26], a high level BBD package.

References

[1] D. A. Schmidt. Data flow analysis is model checking of abstract interpretation. *In Proceedings of the Twenty Fifth Annual Symposium on Principles of Programming Languages*, pages 38-48. ACM Press, 1998.

[2] Cousot, P. and Cousot, R., Abstract Interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints, *Proc. 4th ACM Symposium on Principles of Programming Languages*, pp.238–252, 1977.

[3] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *the International Conference on Compiler Construction* (CC'03), pages 126-137, 2003.

[4] S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *the 10^{th} International Static Analysis Symposium* (SAS), San Diego, CA, June 2003.

[5] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In the *15th International Conference on Compiler Construction* (CC 2006). LNCS volume 3923, Pages 47-64, 2006.

[6] Olin Shivers. Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie Mellon University, May 1991. CMU-CS-91-145.

[7] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve. flow-insensitive pointer analysis. In *SIGPLAN 98 Conference on Programming Language Design and Implementation*, pages 97-105, June 1998.

[8] Reps, T., Schwoon, S., Jha, S., and Melski, D., Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming, 58(1C2):206–263, October 2005.

[9] Sagiv, M., Reps, T., and Horwitz, S., Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167 (1996), 131–170.

[10] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In the *8th International Conference on Concurrency Theory* (CONCUR'97), volume 1243 of LNCS, pages 135-150. Springer-Verlag, 1997.

[11] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701C726, November/December 1998.

[12] Jim Alves-Foss (Ed.). Formal syntax and semantics of Java. LNCS 1523 Springer 1999.

[13] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.

[14] M. Fändrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI),Montreal, Canada, June 1998.

[15] N. Heintze and O. Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), Snowbird, Utah, June 2001.

[16] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), Tampa Bay, Florida, October 2001.

[17] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), 2005.

[18] M. Sridharan and R. Bodik. Refinement-Based Context-Sensitive Points-To Analysis for Java. In the *Proceedings of ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation* (PLDI 2006).

[19] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. Proceedings of the *ACM SIGPLAN Conference on Programming Language Design and Implementation*(PLDI), 2004.

[20] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *the International Symposium on Software Testing and Analysis*, pages 1-11, 2002.

[21] Dean, J., Grove, D., and Chambers, C. Optimization of object-oriented programs using static class hierarchy analysis. In Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95) (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77-101.

[22] Bacon, D. F. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[23] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Proceedings of the 12th International Conference on Compiler Construction* (CC), pages 153-169, April 2003.

[24] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), San Diego, CA, June 2003.

[25] R. Vallee, Phong, C. Etienne, G. Laurie, H. Patrick, and L. Vijay: Soot - a Java bytecode optimization framework, Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research 1999 (CASCON '99), Ontario, Canada, November 1999.

[26] D. Beyer, A. Noack, C. Lewerentz. Efficient Relational Calculation for Software Analysis. IEEE Transactions on Software Engineering (TSE), 31(2):137-149, 2005.