Modular Stacking-based Context-Sensitive Program Analysis

Xin Li, Mizuhito Ogawa

2011-06-29, JAIST Technical Report IS-RR-2011-002

Modular Stacking-based Context-Sensitive Program Analysis

Xin Li and Mizuhito Ogawa

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

Abstract. This paper presents a systematic approach to scaling stacking-based context-sensitive program analysis yielded by weighted pushdown systems, and demonstrates its effectiveness by applications to Java points-to analysis. Our approach relies on the view of formulating stacking-based analysis as AGPs (abstract grammar problem), and reducing the analysis problem to fixed-point calculation over the equation system encoded by grammar productions. To characterize points-to analysis that does not assume an existing program control flow, we propose *Contracted* AGP, and a *symbolic sliding-window based* algorithm for it. The algorithm aims at reducing both time and memory costs, and its correctness is proved in terms of chaotic iteration. Finally, we instantiated the algorithm within Japot, a context-sensitive points-to analyzer for Java. Experiments show two-fold benefits of our approach in practice, with (i) scaling Japot to Dacapo benchmark suite, and (ii) a speed-up of 3x faster in average, given an adjustable memory budget.

1 Introduction

Context-sensitive program analysis is precise yet prohibitive to scale, e.g., enormous study has been devoted to scale up cloning-based points-to analysis [6]. Cloning-based analysis has an inherit limit to handle recursive procedures. However, as studied in [6], there are often many loops in the call graph, and each of which typically contains a large number of methods more than one thousand. As opposed to cloning-based approach, an alternative to context-sensitivity is *stacking-based* that models the program into a (weighted) pushdown system, and manages calling contexts with the finite yet unbounded pushdown stack [8]. This work is concerned with scaling stacking-based analysis as modular analysis, with retaining the original precision of the whole program analysis.

This work is motivated by our work on Japot [8], a stacking-based points-to analysis for Java yielded by weighted pushdown systems (WPDSs) [12]. As shown by previous empirical studies [8], the whole problem analysis in Japot could not scale to half applications of Dacapo Benchmark suite, due to memory overflow. To reduce the time cost, we attempted to carefully interleave the whole program analysis with local analysis in a heuristic manner, in which the whole program analysis is compulsory for ensuring soundness. Although the scalability issue remains, this technique brought us with 2x speed-up in average. A natural question is, whether it is possible to conduct local stacking-based analysis only, with preserving the original precision and soundness.

The idea of using local (or modular) analysis for scalability is not new [3]. That is, program parts are analyzed separately, and the analysis result on the whole program is obtained by composing these partial ones. A known difficulty for local analysis is that program parts are not completely independent. Generally, classic techniques consist in, (1) building a dependency graph of program parts before the analysis, and analyzing parts in their topological order after grouping loops; or (2) breaking such dependencies by providing each part with (often conservative) summary information of external parts on which it depends.

However, it turns out to be nontrivial to apply aforementioned techniques to stacking-based points-to analysis. As anticipated in [3] (§8.5), points-to analysis and call graph construction are mutually dependent, and the dependency graph expected by (1) is not known in advance. Moreover, it is known to be a challenging problem to compute precise, concise and efficient procedure summaries in modular (points-to) analysis [20]. In this work, we limit our focus to investigating the effects of language polymorphism like dynamic dispatch and field access on designing a precise modular stacking-based analysis. The dependency among parts exploited by (1) is detected on-the-fly when the analysis proceeds. The possibility of applying technique (2) is discussed as future work.

One key of our approach is the view of formulating stacking-based analysis as AGPs (abstract grammar problems) [12], and the analysis problem is reduced to computing the fixed-point of the equation system encoded from grammar productions. To model points-to analysis for which the program control flow is not assumed, we present a conceptual framework named *Contracted* AGP, as well as generic algorithms for effectively solving it. Our proposal guides a systematic design of precise modular stacking-based analysis.

The main results can be summarized as follows:

- We present a new problem named CAGP (Contracted AGP), and show that stacking-based points-to analysis for Java is a typical instance of it (§3)). The original AGP are special instances of CAGP.
- We generalize chaotic iteration to *chaotic contracted iteration* (CCI), and prove that CAGP can be solved by CCI (§4). This insight sheds light on the ensuring generic algorithm for CAGP.
- We present and prove a *symbolic sliding window based algorithm* for CCI, to reduce both (practical) time and memory costs (§5). By sliding window, the algorithm iteratively solves parts of the equation system. By symbolic, inputs of the equation system can be referred as symbolic names in local analysis and bound with concrete values in a lazy manner.
- We instantiate the aforementioned algorithm in Japot (§7). Empirical study showed that, this approach brings two-fold benefits in practice, by (i) scaling up Japot to Dacapo benchmark suite, and (ii) speeding up 3x faster in average, given an adjustable memory budget.

Besides, §2 presents weighted pushdown systems, and motives the problem to be addressed. §8 discusses related and future work.

2 Motivations

2.1 Weighted Pushdown System [12]

Definition 1. *A* ***pushdown system*** *(PDS) P is $(Q, \varGamma, \varDelta, q_0, \omega_0)$, where Q is a finite set of control locations, $\varGamma$ is a finite stack alphabet, and $\varDelta \subseteq Q \times \varGamma \times Q \times \varGamma^*$ is a finite set of transition rules, and $q_0 \in Q$ and $\omega_0 \in \varGamma^*$ are the initial control location and stack contents, respectively. A transition rule $(p, \gamma, q, \omega) \in \varDelta$ is written as $\langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle$. A* ***configuration*** *of P is a pair of $\langle q, \omega\rangle$ for $q \in Q$ and $\omega \in \varGamma^*$. A set of configurations C is* ***regular*** *if, for each configuration $\langle p, \omega\rangle \in C$, ω is regular. A relation $\Rightarrow$ on configurations is defined, such that $\langle p, \gamma\omega'\rangle \Rightarrow \langle q, \omega\omega'\rangle$for each $\omega' \in \varGamma^*$ if $\langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle$.*

A pushdown system can be normalized (or simulated) by a pushdown system for which $|\omega| \leq 2$ for each transition rule $\langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle$ [14]. In sequel, we always assume such normalized forms, and require the initial stack contents to be $\gamma_0 \in \varGamma$, without loss of generality.

Definition 2. *A* ***bounded idempotent semiring*** *S is $(D, \oplus, \otimes, \bar{0}\ , \bar{1})$, where $\bar{0}, \bar{1} \in D$, and*

1. *$(D, \oplus)$ is a commutative monoid with $\bar{0}$ as its unit element, and $\oplus$ is idempotent, i.e., $a \oplus a = a$ for all $a \in D$;*
2. *$(D, \otimes)$ is a monoid with $\mathbf{1}$ as the unit element;*
3. *$\otimes$ distributes over $\oplus$, i.e., for all $a, b, c \in D$, we have $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$;*
4. *for all $a \in D, a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$;*
5. *A partial ordering $\sqsubseteq$ is defined on D such that $a \sqsubseteq b$ iff $a \oplus b = a$ for all $a, b \in D$, , and there are no infinite descending chains in D.*

By Def. 2, we have that $\bar{0}$ is the greatest element. From the standpoint of abstract interpretation, PDSs model the (recursive) control flows of the program, weight elements encodes transfer functions, $\otimes$ corresponds to function composition, and $\oplus$ joins data flows.

Definition 3. *A* ***weighted pushdown system*** *(WPDS) W is a triplet (P, S, f), where $P = (Q, \varGamma, \varDelta, q_0, \gamma_0)$ is a pushdown system, $S = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is a bounded idempotent semiring, and $f \colon \varDelta \rightarrow D$ is a weight assignment function.*

Let $\sigma = [r_0, ..., r_k]$ with $r_i \in \varDelta$ for $0 \leq i \leq k$ be a sequence of pushdown transition rules. A value associated with σ is defined by $val(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$. Given $c,' c \in Q \times \varGamma^*$, we denote by $path(c, c')$ the set of transition sequences that transform configurations from c into c'.

Definition 4. *Given a weighted pushdown system $W = (P, S, f)$, where $P = (Q, \varGamma, \varDelta, q_0, \gamma_0)$, and two regular sets of configurations $C, C' \subseteq Q \times \varGamma^*$, the* ***meet-over-all-valid-path*** *(MOVP) problem computes*

$$\mathit{MOVP}(C, C') = \bigoplus\{val(\sigma) \mid \sigma \in path(c, c'), c \in C, c' \in C'\}$$

Let $@(p, \gamma) = \{\langle p, \gamma\omega\rangle \mid \omega \in \varGamma^*\}$ for any $p \in Q, \gamma \in \varGamma$. The ***Head-MOVP*** *problem computes, for each $p \in Q, \gamma \in \varGamma$*

$$\mathit{HMOVP}(p, \gamma) = \mathit{MOVP}(\{\langle q_0, \gamma_0\rangle\}, @(p, \gamma))$$

2.2 Stacking-based Program Analysis by WPDSs

Points-to analysis infers a *points-to relation* $\mathcal{R}$ that maps each reference to the set of objects it may point to at runtime. We denote by *AbsRef* the set of abstract references, and by *AbsObj* the set of abstract heap objects, and let $\Diamond \in$ *AbsObj* denote the *null* reference. Thus $\mathcal{R}$ is a mapping from *AbsRef* to 2^{AbsObj}, and we write $r \mapsto \mathcal{R}(r)$. We adopt allocation site based abstractions on heap, i.e., a unique abstract heap object models concrete heap objects allocated at the same program line. Also, array members of any array reference o are not distinguished, and denoted by $[\![o]\!]$. Therefore, *AbsObj* is syntactically bounded, and can be represented as *pairs of allocation sites and runtime types.*

Definition 5. *Let $\mathcal{P}$ be the powerset constructor. We define a semiring $S_{pta} = (D_{pta}, \oplus, \otimes, \bar{0}, \bar{1})$, where $D_{pta} = \mathcal{P}(AbsObj \times AbsObj)$ is the set of all binary relation on AbsObj, $\bar{1}$ is the identity relation, $\bar{0}$ is the empty set, $\oplus$ is set union, and $d \otimes d' = d \circ d' \cup d' \circ d$, where $\circ$ denotes the relation composition.*

```
0.  public class Main {
1.    public static void main(String[] args){
2.      Object $x_1$ = new String();
3.      Object $x_2$ = new Integer(0);
4.      Object $y_1$ = $f(x_1)$;
5.      Object $y_2$ = $f(x_2)$;
6.    }
7.    public static Object $f$(Object $a$){
8.      if(false) $a$ = new Exception();
9.      return $a$;
10.   }
11. }
```

(a) A Java Code Snippet

$r_0 :$	$\langle \Lambda, \mathrm{main} \rangle \hookrightarrow \langle \Lambda, f\ \mathrm{main} \rangle$	id
$r_1 :$	$\langle \Lambda, \Psi \rangle \hookrightarrow \langle \Lambda, \mathrm{main} \rangle$	id
$r_2 :$	$\langle \Lambda, \mathrm{main} \rangle \hookrightarrow \langle x_1, \mathrm{main} \rangle$	$\{(\_, o_1)\}$
$r_3 :$	$\langle \Lambda, \mathrm{main} \rangle \hookrightarrow \langle x_2, \mathrm{main} \rangle$	$\{(\_, o_2)\}$
$r_4 :$	$\langle x_1, \mathrm{main} \rangle \hookrightarrow \langle \mathtt{arg}, f\ l_4 \rangle$	id
	$\langle \mathtt{ret}, l_4 \rangle \hookrightarrow \langle y_1, \mathrm{main} \rangle$	id
$r_0 :$	$\langle \mathtt{ret}, \mathrm{main} \rangle \hookrightarrow \langle y_1, \mathrm{main} \rangle$	id
$r_5 :$	$\langle x_2, \mathrm{main} \rangle \hookrightarrow \langle \mathtt{arg}, f\ l_5 \rangle$	id
	$\langle \mathtt{ret}, l_5 \rangle \hookrightarrow \langle y_2, \mathrm{main} \rangle$	id
$r_0 :$	$\langle \mathtt{ret}, \mathrm{main} \rangle \hookrightarrow \langle y_2, \mathrm{main} \rangle$	id
$r_8 :$	$\langle \Lambda, f \rangle \hookrightarrow \langle a, f \rangle$	$\{(\_, o_3)\}$
$r_9 :$	$\langle a, f \rangle \hookrightarrow \langle \mathtt{ret}, f \rangle$	id
$r_{10} :$	$\langle \mathtt{ret}, f \rangle \hookrightarrow \langle \mathtt{ret}, \epsilon \rangle$	id

(b) WPDS for the Code in (a)

Fig. 1. Stacking-based Points-to Analysis in a Nutshell. Abstract heap objects allocated on line 2, 3, 8 are $o_1 : (2, \mathtt{String})$, $o_2 : (3, \mathtt{Integer})$, $o_3 : (8, \mathtt{Exception})$, respectively. Λ and Ψ denote the abstract heap environment and dummy program entry, respectively. `ret` and `arg` denote return values and formal arguments, respectively. l_i denote the return point of the method call at line i.

We use the relational weight domain D_{pta} in our analysis. Note that, $\otimes_{pta}$ shows our choice of *flow-insensitivity.* Let $\{(\_, o)\}$ be a shorthand for $\{(i, o) \mid i \in$ *AbsObj*$\}$, and let *id* denote the identity relation $\{(i, i) \mid i \in$ *AbsObj*$\}$. Fig. 1 shows an Java code snippet and its WPDS encoding, where transitions $r_i (i > 0)$ correspond to statements at line i; Λ and Ψ correspond to the initial control location and stack contents, respectively; Λ holds the thread of the program control flow that is encoded in r_0.

As shown in the figure, variables are encoded as control locations, methods, as well as return points of procedure calls, are encoded the stack alphabet. By this encoding, we have for any reference $v \in \mathit{AbsRef}$ of the method m, $\mathcal{R}(v) = \{w(\Diamond) \mid w \in \mathsf{HMOVP}(v, m)\}$. The analysis on example in Fig. 1(a) precisely infers $y_1 \mapsto \{o_1, o_3\}$ and $y_2 \mapsto \{o_2, o_3\}$, whereas a context-insensitive analysis can not distinguish y_1 and y_2. Stacking-based analysis ensures that method calls correctly math with returns, a.k.a., *valid paths.*

Our previous empirical study [8] showed that, the aforementioned points-to analysis for Java (with handling language features of polymorphism) can not scale well. It turns out to be nontrivial to transform the analysis into a modular counterpart. As mentioned shortly in §1, points-to analysis does not assume an existing program control flow, e.g., effects of dynamic dispatch like "$y = x.f(a_1, ..., a_n)$" and field access like "$x.f = y$" could not be analyzed before $\mathcal{R}(x)$ is resolved. We **address the following problem** in this work, "*given a $k \in \mathbb{N}$, conduct a modular stacking-based analysis with analyzing at most k methods at once, without any loss of precision.*"

3 Contracted Abstract Grammar Problem

3.1 Abstract Grammar Problem (AGP) [12]

Let $\mathbb{N}$ be the set of natural numbers, and let $\mathcal{F}$ be a finite *ranked alphabet* associated with the arity function $\mathtt{arity} : \mathcal{F} \to \mathbb{N}$, and $\mathcal{F}_{(k)} = \{f \in \mathcal{F} \mid \mathtt{arity}(f) = k\}$ for all $k \in \mathbb{N}$. We denote by $\mathcal{T}(\mathcal{F})$ the set of *ground terms* over $\mathcal{F}$. Let $\langle D, \sqcap \rangle$ be a meet semilattice. An $\mathcal{F}$-**algebra** $\mathcal{A}$ is $(\langle D, \sqcap \rangle, [\cdot]_\mathcal{A})$, where $[\cdot]_\mathcal{A}$ associates each function $f \in \mathcal{F}$ with an interpretation $[f]_\mathcal{A} : D^{\mathtt{arity}(f)} \to D$. An *evaluation* on ground terms is $[\![\cdot]\!]_\mathcal{A} : \mathcal{T}(\mathcal{F}) \to D$, such that for each $t \in \mathcal{T}(\mathcal{F})$,

$$[\![t]\!]_\mathcal{A} = \begin{cases} [t]_\mathcal{A}, & \text{if } t \in \mathcal{F}_{(0)}; \\ [f]_\mathcal{A}([\![t_1]\!]_\mathcal{A}, ..., [\![t_k]\!]_\mathcal{A}), & \text{if } t = f(t_1, ..., t_k),\ f \in \mathcal{F}_{(k)}, \\ & \text{and } t_i \in \mathcal{T}(\mathcal{F}) \text{ for } 1 \leq i \leq k. \end{cases}$$

Definition 6. *An* **abstract grammar** $\mathcal{G} = (V, T, \Theta, \mathcal{F})$ *is a context-free grammar (V, T, Θ) over a finite ranked alphabet $\mathcal{F}$, where V is a finite set of nonterminals, $T = \mathcal{F} \uplus \{$"(", ")", ","$\}$ is a finite set of terminals, and Θ is a finite set of productions and each has the form $X_0 \to g_\theta(X_1, X_2, ..., X_k)$ for $g_\theta \in \mathcal{F}_{(k)}$.*

We define a relation $\Rightarrow_\mathcal{G}$ on $(V \cup T)^*$, such that $\omega Y \omega' \Rightarrow_\mathcal{G} \omega\alpha\omega'$ for any $\omega, \omega' \in (V \cup T)^*$ if $Y \to \alpha \in \Theta$. Let $\Rightarrow_\mathcal{G}^*$ be the reflexive and transitive closure of $\Rightarrow_\mathcal{G}$, and let $L(X)$ be the language generated by the non-terminal X, i.e., $L(X) = \{\omega \in T^* \mid X \Rightarrow_\mathcal{G}^* \omega\}$. In sequel, we assume $V = \{X_1, ..., X_n\}$ in $\mathcal{G}$. We denote by D^n the Cartesian product of D, and by $d[i]$ the i^{th} projection of any $d \in D^n$. We write $d[X_i]$ for $d[i]$ when it is clear from the context. We extend $\sqcap$ over D to D^n by $d \sqcap d' = (d[1] \sqcap d'[1], ..., d[n] \sqcap d'[n])$ for any $d, d' \in D^n$.

Definition 7. *Given an abstract grammar $\mathcal{G} = (V, T, \Theta, \mathcal{F})$ with $V = \{X_1, ..., X_n\}$, and an $\mathcal{F}$-**algebra** $\mathcal{A} = (\langle D, \sqcap \rangle, [\cdot]_\mathcal{A})$ in which D has the greatest element $\top$. The*

abstract grammar problem *(AGP) is to compute $val(\Theta) \in D^n$, such that for each $1 \leq i \leq n$,*

$$val(\Theta)[i] = \begin{cases} \top & \text{if } L(X_i) = \emptyset; \\ \bigsqcap_{\omega \in L(X_i)} [\![\omega]\!]_{\mathcal{A}} & \text{if } L(X_i) \neq \emptyset. \end{cases}$$

For each production $X_{i_0} \longrightarrow g_\theta(X_{i_1}, X_{i_2}, ..., X_{i_k})$ where $i_0 < ... < i_k$ with $i_j \in \{1, ..., n\}$ and $0 \leq j \leq k$, we extend $[g_\theta]_{\mathcal{A}}$ to be an n-ary function on D^n with overloading g_θ, such that for each $1 \leq t \leq n$, and $d \in D^n$,

$$g_\theta(d)[t] = \begin{cases} [g_\theta]_{\mathcal{A}}(d[i_1], d[i_2], ..., d[i_k]) & \text{if } t = i_0; \\ d[t] & \text{otherwise.} \end{cases}$$

We say that g_θ is *dependent* of $X_{i_1}, X_{i_2}, ..., X_{i_k}$, and the set of g_θ is denoted by $\mathcal{F}(\Theta)$. We are able to define a function $f(\Theta)$ on D^n by

$$f(\Theta) = \lambda X.X \sqcap \bigsqcap_{X_{i_0} \rightarrow g_\theta(X_{i_1}, X_{i_2}, ..., X_{i_k})} g_\theta(X)$$

It is established in [10] that $val(\Theta)$ is the maximum fixed point of f, if $[g_\theta]_{\mathcal{A}}$ is *distributive* (and thus monotonic) for each $g_\theta \in \mathcal{F}$. [1]

(0) $N_{(q_0,\gamma_0,q_f)} \longrightarrow g_0(\epsilon)$
$\quad [g_0] = \bar{1}$
(1) $N_{(p',\gamma',q')} \longrightarrow g_1(N_{(p,\gamma,q)}C_{(q,\gamma',q')})$ $\qquad r = \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta$
$\quad [g_1] = \lambda x.\lambda y.y \otimes x \otimes f(r)$, and $q, q' \in \mathcal{Q}, \gamma' \in \Gamma$
(2) $N_{(p',\gamma',q)} \longrightarrow g_2(N_{(p,\gamma,q)})$ $\qquad r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma' \rangle \in \Delta$
$\quad [g_2] = \lambda x.x \otimes f(r)$, and $q \in \mathcal{Q}$
(3) $N_{(p',\gamma',q_{p',\gamma'})} \longrightarrow g_3(\epsilon)$ $\qquad r = \langle p, \gamma \rangle \hookrightarrow \langle p', \gamma'\gamma'' \rangle \in \Delta$
$\quad [g_3] = \bar{1}$
(3′) $C_{(q_{p',\gamma'},\gamma'',q)} \longrightarrow g_3'(N_{(p,\gamma,q)})$
$\quad [g_3'] = \lambda x.x \otimes f(r)$, and $q \in \mathcal{Q}$
(4) $Head_{(p,\gamma)} \longrightarrow g_4(N_{(p,\gamma,q)}Dummy_{(q,q_f)})$
$\quad [g_4] = \lambda x.\lambda y.y \otimes x$, and $p \in Q, \gamma \in \Gamma$, $q \in \mathcal{Q}$
(4′) $Head_{(p,\gamma)} \longrightarrow g_4'(N_{(p,\gamma,q_f)})$
$\quad [g_4'] = \lambda x.x$, and $p \in P, \gamma \in \Gamma$
(5) $Dummy_{(q,q_f)} \longrightarrow g_5(C_{(q,\gamma,q')}Dummy_{(q',q_f)})$
$\quad [g_5] = \lambda x.\lambda y.y \otimes x$, and $q' \in \mathcal{Q}$, and $\gamma \in \Gamma$
(5′) $Dummy_{(q,q_f)} \longrightarrow g_5'(C_{(q,\gamma,q_f)})$
$\quad [g_5'] = \lambda x.x$, and $q \in \mathcal{Q}$, $\gamma \in \Gamma$

Fig. 2. An Abstract Grammar Problem for Solving the Head-MOVP Problem. $\mathcal{Q} = \{q_{p,r} \mid p \in Q, \gamma \in \Gamma\}$, and q_f is a fresh symbol.

Program analysis problems on WPDSs can be reduced to AGPs [12]. We give an instance of AGP for solving the Head-MOVP problem in Figure 2, where (1) – (3′) are production schemes encoded from pushdown transitions, and (4) – (5′) and (0) are those specific to the Head-MOVP problem ((0) corresponds to the

[1] [10] works on the function $f'(\Theta) = \lambda X. \bigsqcap_{g_\theta \in \mathcal{F}(\Theta)} g_\theta(X)$. It is easy to see that $f'(\Theta)$ coincides with $f(\Theta)$ on the maximum fixed point.

initial configuration $\langle p_0, \gamma_0 \rangle$, and (4)$-$(5$'$) corresponds to the target configuration and are derived from (1) $-$ (3$'$)). We have for each $p \in Q$ and $\gamma \in \Gamma$,

$$\mathtt{HMOVP}(p, \gamma) = val(\Theta)[Head_{(p,\gamma)}]$$

3.2 Contracted AGPs and Applications

Due to language polymorphism like dynamic dispatch and field access, the program control flow is not known in advance, and is discovered on-the-fly when points-to analysis proceeds. We present *Contracted* AGP in Def. 8, to offer some insights into the cyclic dependency between control flow and data flow analysis.

Definition 8. *Given an abstract grammar $\mathcal{G} = (V, T, \Theta, \mathcal{F})$, and an $\mathcal{F}$-algebra $\mathcal{A} = (\langle D, \sqcap \rangle, [\cdot]_{\mathcal{A}})$ in which D has the greatest element $\top$. A function $\mathcal{E} : D^n \rightarrow \mathcal{P}(\Theta)$ is a* ***contract function*** *if it is anti-monotonic. We define a function K on $\mathcal{P}(\Theta)$ by $K = \lambda x.x \cup \mathcal{E}(val(x))$. Let Θ^* be the least fixed point of K. The* ***contracted abstract grammar problem*** *(CAGP) is to compute $val(\Theta^*)$.*

Remark 1. It is not hard to see that K is monotonic, and its least fixed point Θ^* exists and can be computed as the limit of the iterates $K^{(0)}(\emptyset), K^{(1)}(\emptyset), \ldots$ by Kleen's theorem. However, such an approach faces the scalability problem since the underlying grammar problem keeps enlarged.

Java Points-to Analysis as Solving CAGPs

We use the abstraction and modelling discussed in §2.2, and further assume a typed three-address form of the target language. Given the abstraction, the possible objects that a reference variable may point to is known as long as its declared type permits. We prepare a function $\mathsf{allObj} : AbsRef \rightarrow 2^{AbsObj}$ by, for each $v \in AbsRef$, $\mathsf{allObj}(v) = \{(l, \mathtt{T}) \in AbsObj \mid \mathtt{T}$ is a subtype of $v's$ declared type$\}$. Let $\mathcal{R}_\perp$ be a points-to relation such that $\mathcal{R}_\perp(r) = \emptyset$ for all $r \in AbsRef$. Given $os \subseteq AbsObj$, and $r \in AbsRef$, for each $r' \in AbsRef$, $\mathcal{R}[r \mapsto os](r') = os$ if $r' = r$, and $\mathcal{R}[r \mapsto os](r') = \mathcal{R}(r)$ otherwise. Moreover, the union of $\mathcal{R}$ and $\mathcal{R}'$ is defined as, for each $r \in AbsRef$, $\mathcal{R} \cup \mathcal{R}'(r) = \mathcal{R}(r) \cup \mathcal{R}'(r)$.

Let $\mathcal{L}$ be the set of distinguished program line numbers, and let $\mathcal{M}$ be the set of methods. A *labelled call graph* G is (N, E), where $N \subseteq \mathcal{M}$ is the set of methods, and $E \subseteq \mathcal{M} \times \mathcal{L} \times \mathcal{M}$ is the set of call edges. We define a labeling function $\mathtt{rc}$ from the call edges to points-to relations, such that $\mathtt{rc}(e)$ specifies possible conditions (i.e., points-to relations on runtime types of the receiver object) that enable the call relation e. We assume an imprecise call graph syntactically built, e.g., by CHA (Class Hierarchy Analysis) [4], and $\mathtt{rc}$ is decided by the language semantics, for each $e = (m', l, m) \in E$,

$$\mathtt{rc}(e) = \begin{cases} \{\mathcal{R}_\perp[x \mapsto \{o\}] \mid o \in \mathsf{allObj}(x), & \text{if } l \text{ is a dynamic dispatch with} \\ \qquad\qquad x \mapsto \{o\} \text{ enables } e\}, & x \text{ being the implicit parameter;} \\ \\ \{\mathcal{R}_\perp\}, & \text{if } l \text{ is a static method invocation.} \end{cases}$$

Let $m_0 \in N$ be the initial node. For each $m \in \mathcal{M}$, we define conditions for m being reachable from m_0, denoted by $\mathsf{RC}(m)$, by

$$\mathsf{RC}(m) = \begin{cases} \{\mathcal{R}_\perp\}, & \text{if } m = m_0; \\ \{\mathcal{R}' \cup \mathcal{R} \mid \mathcal{R}' \in \mathsf{RC}(m'), \mathcal{R} \in \mathtt{rc}(e), e = (m', l, m) \in E\}, & \text{otherwise.} \end{cases}$$

Fig. 3 shows the encoding of a WPDS $W = (P, S, f)$ with $P = (Q, \Gamma, \Delta, q_0, \gamma_0)$, given G. We define a labelling function $\mathtt{rc}$ from Δ to points-to relations, such that for any transition $r \in \Delta$, $\mathtt{rc}(r)$ specifies runtime types of the base variable (i.e., x in $x.f(r_1, ..., r_n)$, or $x.f$, or $x[i]$ for any $i \in \mathbb{N}$) that enable r. To reduce the exponential growth of transitions, we distinguish a global reference (i.e., array or field reference) in each method as if it is a local one, and the unified result is referred from Λ on demand [8]. Our analysis is context-sensitive and field-sensitive (and flow-insensitive). Context-sensitive and field-sensitive analysis is undecidable [11]. Instead of matching field (resp. array) read and write during the context-sensitive analysis, we first cast aliasing in field (resp. array) reference.

$$\begin{aligned}
\mathcal{I}[\![x = \text{new T}]\!] &= \{\langle \Lambda, m\rangle \underset{c}{\overset{\{(\_,(l,\mathrm{T}))\}}{\hookrightarrow}} \langle x, m\rangle\} \\
\mathcal{I}[\![x = y]\!] &= \{\langle y, m\rangle \underset{c}{\hookrightarrow} \langle x, m\rangle\} \\
\mathcal{I}[\![x := (\mathrm{T})y]\!] &= \{\langle y, m\rangle \underset{c}{\hookrightarrow} \langle x, m\rangle\} \\
\mathcal{I}[\![x := \text{@this}:\ \mathrm{T}]\!] &= \{\langle \mathtt{this}, m\rangle \underset{c}{\hookrightarrow} \langle x, m\rangle\} \\
\mathcal{I}[\![x := \text{@parameter}_k : \mathrm{T}]\!] &= \{\langle \mathtt{arg}_k, m\rangle \underset{c}{\hookrightarrow} \langle x, m\rangle\} \\
\mathcal{I}[\![\text{return } x]\!] &= \{\langle x, m\rangle \underset{c}{\hookrightarrow} \langle \mathtt{ret}, m\rangle\} \\
\mathcal{I}[\![y.f = x]\!] &= \{\langle x, m\rangle \underset{c \cup \{\mathcal{R}_\perp[y \mapsto \{o\}]\}}{\hookrightarrow} \langle o.f, m\rangle \mid o \in \mathsf{allObj}(y)\} \\
\mathcal{I}[\![x = y[i]]\!] &= \{\langle [\![o]\!], m\rangle \underset{c \cup \{\mathcal{R}_\perp[y \mapsto \{o\}]\}}{\hookrightarrow} \langle x, m\rangle \mid o \in \mathsf{allObj}(y)\} \cup A_g, \text{ where} \\
A_g &= \{\langle \Lambda, m\rangle \underset{c \cup \{\mathcal{R}_\perp[[\![o]\!] \mapsto \{o'\}]\}}{\overset{\{(\_,o')\}}{\hookrightarrow}} \langle [\![o]\!], m\rangle \mid o' \in \mathsf{allObj}([\![o]\!])\} \\
\mathcal{I}[\![z = r_0.f(r_1, ..., r_n)]\!] &= A_c \cup A_r, \text{ where } (m, l, m') \in E,\ c' = \mathtt{rc}((m, l, m')) \\
A_c &= \{\langle r_0, m\rangle \underset{c \cup c'}{\hookrightarrow} \langle \mathtt{this}, m'\ l\rangle\} \cup \\
&\quad \bigcup_{0 \le i \le n} \{\langle r_i, m\rangle \underset{c \cup c'}{\hookrightarrow} \langle \mathtt{arg}_i, m'\ l\rangle\} \\
A_r &= \{\langle \mathtt{ret}, m'\rangle \underset{c}{\hookrightarrow} \langle \mathtt{ret}, l\rangle\} \cup \{\langle \mathtt{ret}, l\rangle \underset{c}{\hookrightarrow} \langle z, m\rangle\}.
\end{aligned}$$

Fig. 3. $\mathcal{I}[\![\_]\!]$ generates W for a statement at $l \in \mathcal{L}$ of the method m, where $c = \mathsf{RC}(m)$. For any $r : \langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle \in \Delta$, we write $\langle p, \gamma\rangle \underset{\mathtt{rc}(r)}{\overset{f(r)}{\hookrightarrow}} \langle q, \omega\rangle$, and omit $f(r)$ if it is $\bar{1}$.

Let Θ_{pta} be the abstract grammar generated for the program (Fig. 3 and Fig. 2), and let n be the number of non-terminals in Θ_{pta}. We define the contract function $\mathcal{E}$ by, for each $d \in D^n_{pta}$, and for any $\theta \in \Theta_{pta}$, $\theta \in \mathcal{E}(d)$ if either

- (i) there exists a transition $r \in \Delta$ and $\mathcal{R} \in \mathtt{rc}(r)$, such that, for each $r \in$ *AbsRef*, $\bigoplus_{\gamma \in \Gamma} d[Head_{(r,\gamma)}] \sqsubseteq \mathcal{R}(r)$, and (ii) θ is a production generated from r by (1)-(3') in Fig. 2; or
- θ is any production generated by (0) or (4) – (5′) in Fig. 2.

4 Chaotic Contracted Iteration

Definition 9. *Given a set D, and an element $d \in D$, and a finite set of functions $F = \{f_1, ..., f_n\}$ on D.*

- *A* ***run*** *is an infinite sequence of functions in F.*
- *An* ***iteration*** *of F associated with a run $f_{i_1}, f_{i_2}, ...$ and starting with d is an infinite sequence of values $d_0, d_1, ...,$ such that $d_0 = d$ and $d_j = f_{i_j}(d_{j-1})$.*
- *A run $f_{i_1}, f_{i_2}, ...$ is* ***fair*** *if, for each $m \geq 0$, $F \subseteq \bigcup\limits_{j>m} \{f_{i_j}\}$.*
- *An iteration of F is* ***chaotic*** *if it is associated with a fair run.*

Definition 10. *Given a partially ordered set $(D, \sqsubseteq)$. A function f on D is* ***inflationary*** *if $f(x) \sqsupseteq x$ and* ***downward inflationary*** *if $f(x) \sqsubseteq x$, for each $x \in D$, respectively.*

In sequel we fix the following notations,

- $\langle D, \sqcap \rangle$ for a meet semi-lattice with the greatest element $\top$ and the descending chain condition (i.e., there are no infinite descending chains in D),
- $x \sqsubseteq y$ when $x \sqcap y = x$ for $x, y \in D$, and
- $F = \{f_1, ..., f_n\}$ for a finite set of monotonic and downward inflationary functions on D.

Theorem 1 (Chaotic Iteration [2] [1]). *The limit of any chaotic iteration of F exists and coincides with $\prod\limits_{j=0}^{\infty} f^j(\top)$, where $f = \lambda x. \prod\limits_{i=0}^{n} f_i(x)$.*

Instead of starting from $\top$, we can iteratively apply the function f starting at $d \in D$, and its limit $\prod\limits_{j=0}^{\infty} f^j(d)$ uniquely exists. We refer it by $\mathsf{fix}(f, d)$.

Definition 11. *Let $d \in D$.*

- *A function $\phi : D \to 2^F$ is a* ***contract function*** *if ϕ is anti-monotonic.*
- *A* ***contracted iteration****(CI) of F associated with a run $f_{i_1}, f_{i_2}, ...$ and starting with d is an infinite sequence of values $d_0, d_1, ...,$ such that $d_0 = d$ and $d_j = f_{i_j}(d_{j-1})$ where $i_j \in \phi(d_{j-1})$.*
- *A run $f_{i_1}, f_{i_2}, ...$ is* ***fair*** *for a contracted iteration $d_0, d_1, ...$ if, for each $m > 0$, $\phi(d_j) \subseteq \bigcup\limits_{j>m} \{i_j\}$.*
- *A contracted iteration of F is* ***chaotic*** *if it is associated with a fair run.*

Definition 12. *Let $S = \{f_1, ..., f_k\} \subseteq F$. We define,*

$$\begin{cases} \mathcal{F}un(S) = \lambda x. \prod\limits_{f \in S} f(x), & \text{and} \\ \mathcal{S}er(S) = \{f_{i_1} \circ f_{i_2} \circ ... \circ f_{i_k} \mid (i_1, i_2, ..., i_k) \text{ is a permutation of } \{1, ..., k\}\} \end{cases}$$

[2] Theorem 1 holds for a cpo, in which the direction of the ordering is reversed [1].

Lemma 1. *Let $S \subseteq F$. For each $x \in D$ and $h \in \mathcal{S}er(S)$, $\mathcal{F}un(S)(x) \sqsupseteq h(x)$.*

Proof. Since each function in F is downward inflationary, for each $1 \leq j \leq k$, $x \in D$, and $h \in \mathcal{S}\mathtt{er}(S)$, $f_{i_j}(x) \sqsupseteq h(x)$ holds. □

Theorem 2 (Chaotic Contracted Iteration). *Given a contract function $\phi : D \to 2^F$. The limit of any chaotic contracted iteration of F starting with $d \in D$ exists and coincides with $\mathsf{fix}(\tau_d, d)$, where the function τ_d on D is defined by $\tau_d = \lambda z.\mathsf{fix}(g_z, d)$ and $g_z = \mathcal{F}un(\phi(z))$.*

Proof. The theorem follows from the following steps.

(1) Consider any given chaotic contracted iteration $d_0, d_1, d_2, ...$ of F starting with d. Since each function in F is downward inflationary, we have $d_i \sqsupseteq d_{i+1}$ for $i \geq 0$. Moreover, since there are no infinite descending chains in D, we have that the limit of any chaotic contracted iteration of F exists.

(2) Consider any $d_1, d_2 \in D$ with $d_1 \sqsubseteq d_2$. Since ϕ is anti-monotonic and $\phi(d_1) \supseteq \phi(d_2)$, we have $g_{d_1}(x) \sqsubseteq g_{d_2}(x)$ for all $x \in D$ and $\mathsf{fix}(g_{d_1}, d) \sqsubseteq \mathsf{fix}(g_{d_2}, d)$. We have $\tau_d(d_1) \sqsubseteq \tau_d(d_2)$; thus τ_d is monotonic. Since g_d is downward inflationary, $\tau_d(d) \sqsubseteq d$ by definition of τ_d. Therefore, we have that $d \sqsupseteq \tau_d(d) \sqsupseteq \tau_d^2(d) \sqsupseteq ...$ and the limit $\mathsf{fix}(\tau_d, d)$ exists, since there are no infinite descending chains in D.

(3) Consider any given chaotic contracted iteration $a_0, a_1, a_2, ...$ of F associated with the run $f_{i_1}, f_{i_2}, ...$ and starting with d, and the descending sequence $b_0, b_1, b_2, ...$ where $b_0 = d$ and $b_j = \tau_d(b_{j-1})$ for $j \geq 1$. To prove they converge at the same limit follows the following claims:

- $\forall j.\ a_j \sqsupseteq b_j$.
  By induction on j. (i) Obvious when $j = 0$, $a_0 = b_0 = d$. (ii) Assume that $a_j \sqsupseteq b_j$ for some $j > 0$, then we have

$$\begin{aligned} a_{j+1} &= f_{i_j}(a_j) \qquad (\text{where } f_{i_j} \in \phi(a_j)) \\ &\sqsupseteq f_{i_j}(b_j) \sqsupseteq f_{i_j}(b_{j+1}) \\ &\sqsupseteq g_{b_j}(b_{j+1}) \qquad (a_j \sqsupseteq b_j \Rightarrow f_{i_j} \in \phi(b_j)) \\ &= b_{j+1} \end{aligned}$$

- $\forall j\ \exists j'.\ b_j \sqsupseteq a_{j'}$.
  By induction on j. (i) Obvious when $j = 0$, $a_0 = b_0 = d$. (ii) Assume that, for $j > 0$, there exists $j' > 0$ such that $b_j \sqsupseteq a_{j'}$. Let k is the smallest number that satisfies $g^k_{b_j}(d) = g^{k+c}_{b_j}(d)$ for each $c \geq 0$. By definition, we have

$$b_{j+1} = \tau_d(b_j) = \mathsf{fix}(g_{b_j}, d) = {g_{b_j}}^k(d).$$

  Since $d = b_0 \sqsupseteq b_j$, we have

$$\begin{aligned} {g_{b_j}}^k(d) \sqsupseteq {g_{b_j}}^k(b_j) \sqsupseteq {g_{b_j}}^k(a_{j'}) &= \mathcal{F}\mathrm{un}(\phi(b_j))^k(a_{j'}) \\ &\sqsupseteq h_1 \circ ... \circ h_k(a_{j'}) \qquad (\text{by Lemma 1}) \end{aligned}$$

  where $h_i \in \mathcal{S}\mathtt{er}(\phi(b_j))$ for each $1 \leq i \leq k$.

Consider the suffix $f_{i_{j'+1}}, f_{i_{j'+2}}, ...$ of the run associated with the chaotic contracted iteration $a_0, a_1, ...$ For each $h \in \mathcal{S}\mathtt{er}(\phi(b_j))$, fairness implies the existence of $t_h \in \mathbb{N}$ such that $h_1 \circ ... \circ h_k$ is embedded into $f_{i_{t_h}} \circ \cdots \circ f_{i_{j'+2}} \circ f_{i_{j'+1}}$. Let t_H be the smallest t_h. We have $h_1 \circ ... \circ h_k(a_{j'}) \sqsupseteq f_{i_{t_H}} \circ \cdots \circ f_{i_{j'+2}} \circ f_{i_{j'+1}}(a_{j'})$ due to downward inflationary, and thus $b_{j+1} \sqsupseteq a_{t_H}$. □

Theorem 2 says that we can solve CAGPs in terms of CCIs, by instantiating ϕ with $\mathcal{E}$, and F with $\mathcal{F}(\Theta)$.

5 A Symbolic Sliding Window based Algorithm for CCI

As shown in the proof of Theorem 2, $\mathsf{fix}(\tau_d, d)$ would have faster convergence than a contracted chaotic iteration. However, it is prohibitive to scale when $\phi(z)$ becomes larger and larger for ϕ is anti-monotonic. To reduce memory costs, we present two techniques for effectively solving CCI, a sliding window based algorithm and further acceleration by symbolic computation.

5.1 A Sliding Window based Algorithm

Algo. 1 gives a sliding window based CCI algorithm that will continue forever. We maintain a boolean variable *checked* for each function f in F, and refer it by f*.checked*. We use the *iteration* number (starting with 1) enclosed with parenthesis as the superscript of variables in the algorithm, to refer their value at the entry of the loop (line 2) in that iteration, e.g., $d^{(i)}, F_r^{(i)}, F_w^{(i)}, f.checked^{(i)}$. Note that by replacing the condition "$++iteration > 0$" of the while loop (line 2) with the negation of "$f.checked = 1$ *for each* $f \in F_r$", we obtain a terminating algorithm that computes the same limit.

Definition 13. *For an n-ary function f over D^n, f is independent of the i-th variable if, for each $d_1, ..., d_{i-1}, d_{i+1}, ..., d_n, d, d' \in D$,*

$$f(d_1, ..., d_{i-1}, d, d_{i+1}, ..., d_n) = f(d_1, ..., d_{i-1}, d', d_{i+1}, ..., d_n).$$

In Algo. 1, we assume n variables $X_1, ..., X_n$ such that X_i is always the i-th variable of each function. $DepVar(f)$ denotes a set of variables such that its complement consists of independent variables of f.

Definition 14. *A function $\mathcal{S}ch : 2^F \times \mathbb{N} \to 2^F$ is a **scheduler** for Algorithm 1 if, for each $F' \subseteq F$ and $i \in \mathbb{N}$, $\mathcal{S}ch(F', i) \subseteq F' \cap \{f \in F \mid f.checked^{(i)} = 0\}$. A scheduler is **fair** if, for each $i > 0$, there exists $f \in F_r^{(i)}$ with $f.checked^{(i)} = 0$, then there exists $j \geq i$ with $f \in F_w^{(j)}$.*

Remark 2. By sliding window, we refer to parts of equations $\mathcal{F}_w$ in the iterates. Although $\mathcal{F}_w$ is defined (Def. 14) to take functions from $\{f \mid f.checked == 0\}$, it is safe to include any functions $\{f \mid f.checked == 1\}$, as revealed in the proof

Algorithm 1: A Sliding Window based CCI Algorithm

```
$F_r := \emptyset$; $d := (\top, ..., \top) \in D^n$; $iteration := 0$;
while $(++iteration > 0)$ do
    $F_r := \phi(d)$;
    $F_w := \mathcal{S}\mathrm{ch}(F_r, iteration)$ ;
    $d' := \mathsf{fix}(\mathcal{F}\mathrm{un}(F_w), d)$ ;
    foreach $f$ in $F_w$ do $f.checked := 1$;
    $UpdatedVar := \{X_t \mid d[t] \neq d'[t]$ for each $1 \leq t \leq n\}$ ;
    $NewRules := \phi(d') \setminus F_r$ ;
    foreach $f$ in $F_r \setminus F_w$ do
        if $DepVar(f) \cap UpdatedVar \neq \emptyset$ then $f.checked := 0$;
    end
    foreach $f$ in $NewRules$ do $f.checked := 0$;
    $d := d'$;
end
```

for Theorem 3. Algo. 1 says that sliding window are freely chosen under fairness. As the most rewarding result, we are enabled to impose any given threshold k on $|\mathcal{F}_w|$, i.e., $|\mathcal{F}_w| < k$, without violating its sound convergence. It works for the occasion where the memory budget is limited.

In sequel we prove the correct convergence of Algo. 1.

Definition 15. *Given a contract function $\phi : D \rightarrow 2^F$.*

- *A* ***batched run*** *is an infinite sequence of sets of functions from 2^F.*
- *A* ***batched contracted iteration*** *(BCI) of F starting with $d \in D$ is an infinite sequence of values $d_0, d_1, ...$ associated with a batched run $F_1, F_2, ...$ such that $d_0 = d$, and for each $j > 0$, $d_j = \mathsf{fix}(g_j, d_{j-1})$ where $g_j = \mathcal{F}un(F_j)$, and $F_j \subseteq \phi(d_{j-1})$.*
- *A batched run $F_1, F_2, ...$ is fair for a batched contracted iteration $d_0, d_1, ...$ if each $i \geq 0$, $\phi(d_i) \subseteq \bigcup\limits_{j>i} F_j$.*
- *A BCI is chaotic if it is associated with a fair batched run.*

Lemma 2 (Chaotic BCI). *Given a contract function $\phi : D \rightarrow 2^F$. The limit of any chaotic BCI of F starting with $d \in D$ exists and coincides with the limit of any chaotic contracted iteration of F starting with d.*

Proof. For a chaotic BCI of F starting with $d \in D$, we can construct a CCI of F starting with d by element-wise enumeration over each set of functions F_j in the batched run. Since any chaotic contracted iteration of F stabilizes at the same limit by Theorem 2, the theorem holds. □

Theorem 3. *Assume running Algorithm 1 with a fair scheduler. We have*

1. *The limit of $d^{(1)}, d^{(2)}, ...$ reaches at $d^{(i)}$ if $f.checked^{(i)} = 1$ for each $f \in F_r^{(i)}$.*

2. *The limit of $d^{(1)}, d^{(2)}, \ldots$ exists and there exists a chaotic BCI of F starting with $(\top, \ldots, \top) \in D^n$, such that two limits coincide.*

Proof. 1. $d^{(i)}$ is the limit, since the loop invariant that "*for each f, if $f.checked^{(j)} = 1$ then $f(d^{(j)}) = d^{(j)}$*" holds (at line 2).

2. Since all functions in F are downward inflationary, we have $d^{(1)} \sqsupseteq d^{(2)} \sqsupseteq \ldots$. The limit of $d^{(1)}, d^{(2)}, \ldots$ exists, otherwise the descending chain condition is violated. We define a BCI $d_0, d_1, \ldots$ of F associated with a batched run $F_1, F_2, \ldots$ and $d_0 = (\top, \ldots, \top) \in D^n$ such that $F_{2m-1} = \overline{F}_w^{(m)}$ and $F_{2m} = F_w^{(m)}$, where $\overline{F}_w^{(m)} = \{f \in F_r^{(m)} \mid f.checked^{(m)} = 1\}$ for $m > 0$.

(1) Since "*for each f, $f.checked^{(m)} = 1$ implies $f(d^{(m)}) = d^{(m)}$*" is a loop invariant (at line 2), $d_{2m} = d_{2m+1} = d^{(m)}$. Thus, the limit of $d^{(1)}, d^{(2)}, \ldots$ coincides with the limit of $d_0, d_1, \ldots$.

(2) We show $F_1, F_2, \ldots$ is fair, i.e., for each $i \geq 0$, $\phi(d_i) \subseteq \bigcup_{j>i} F_j$. Since $d_{2m} = d_{2m+1} = d^{(m)}$, $\phi(d_{2m}) = \phi(d_{2m+1}) = \phi(d^{(m)}) = F_r^{(m+1)}$. For each $f \in F_r^{(m+1)}$, if $f.checked^{(m+1)} = 1$, $f \in \overline{F}_w^{(m+1)} = F_{2m+3}$; otherwise, by the definition of fair scheduler, there exists $m' \geq m+1$ with $f \in F_w^{(m')} = F_{2m'}$. □

5.2 Acceleration by Symbolic Computation

As the application size grows, the points-to sets of reference variables could be enormous. To prevent local analyses from operating on such huge data, we introduce symbolic computation technique, to further improve Algo. 1. The idea is to allow (arbitrary subset of) inputs of the equation system to be referred as symbolic names in the fixed point calculation, and the analysis result is obtained by binding symbolic names to their concrete values lazily. This approach anticipates that the cost of substitution would be paid off on the whole.

To show that such a symbolic algorithm will soundly converge as Algo. 1, we limit our discussion to the type of equation systems derived from stacking-based program analysis by WPDSs, as shown in Fig. 2, and base our presentation in terms of formal power series [13] [3]. Let Σ be an alphabet, and let S be a set. Mappings r from Σ^* to D are called (formal) power series. For $\omega \in \Sigma^*$, we write (r, ω) for $r(\omega)$, and r is written as a sum $r = \sum_{\omega \in \Sigma^*} (r, \omega)\omega$. The collection of all power series r as defined above is denoted by $D\langle\langle \Sigma^* \rangle\rangle$. For $r, r' \in D\langle\langle \Sigma^* \rangle\rangle$, the sum $r + r'$ and the product $r \cdot r'$ are power series in $D\langle\langle \Sigma^* \rangle\rangle$ defined as

$$(r + r', \omega) = (r, \omega) + (r', \omega) \text{ and } (r \cdot r', \omega) = \textstyle\sum_{\omega_1 \omega_2 = \omega} (r, \omega_1)(r', \omega_2)$$

In the rest of this section, we assume that (i) D is a bounded idempotent semiring $(D, +, \cdot, \bar{0}, \bar{1})$, (ii) D is finite with the greatest element $\top$, and (iii) $\cdot$ is idempotent and commutative. We further assume that (iv) *the concatenation of words is idempotent and commutative*, i.e., for any $w, w' \in \Sigma^*$, $ww' = w'w$ and $ww = w$. Then, it is easy to check that, $(D\langle\langle \Sigma^* \rangle\rangle, +, \cdot, \bar{0}, \bar{1})$ is also a bounded

[3] An alternative could be in terms of symbolic execution that re-interprets the language constructs over symbolic values.

idempotent semiring. Note that the idempotent laws and commutativity laws in (iii) and (iv) ensure termination of the following symbolic computation.

Let $\mathcal{X} = \{X_1, ..., X_n\}$ be a finite set of variables. We define a set of functions *Func* in which each function has the form $\lambda X_1, ..., X_n.(X_1 + r_1, ..., X_n + r_n)$, where $r_1, ..., r_n \in D\langle\langle \mathcal{X}^* \rangle\rangle$. Note that, the function scheme $\lambda X.X + r$ satisfies downward inflationary property. A fresh distinguished symbolic value s_i is assigned to each variable $X_i \in \mathcal{X}$ $(1 \le i \le n)$. The set of symbolic values $\{s_1, ..., s_n\}$ is denoted by $Sym\mathcal{X}$. For $d \in D^n$ and $seeds \subseteq \{1, ..., n\}$. We denote by

- $\sigma_{seeds,d}$ the ground substitution such that, for each $1 \le i \le n$,
$$\sigma_{seeds,d}(s_i) = \begin{cases} d[i] & \text{if } i \in seeds; \\ \top & \text{otherwise.} \end{cases}$$
- d_{seeds} the value from $(D\langle\langle Sym\mathcal{X}^* \rangle\rangle)^n$ such that, for each $1 \le i \le n$,
$$d_{seeds}[i] = \begin{cases} s_i & \text{if } i \in seeds; \\ d[i] & \text{otherwise.} \end{cases}$$

Lemma 3. *For any function $f \in Func$, any $d \in D^n$, and any $seeds \subseteq \{1, ..., n\}$, we have* $\mathsf{fix}(f, d_{seeds})\sigma_{seeds,d} = \mathsf{fix}(f, d)$.

Proof. The proof is straightforward based on assumptions above. □

Theorem 4. *Given $seeds \subseteq \{1, ..., n\}$ and $F \subseteq Func$. Assume replacing line 5 in Algo. 1 with $d' := \mathsf{fix}(\mathcal{F}un(F_w), d_{seeds})\sigma_{seeds,d}$ and running Algo. 1 with a fair scheduler.*

1. *The limit of $d^{(1)}, d^{(2)}, ...$ reaches at $d^{(i)}$ if $f.checked = 1$ for all $f \in F$.*
2. *The limit of $d^{(1)}, d^{(2)}, ...$ exists and there exists a chaotic BCI of F starting with $(\top, ..., \top) \in D^n$, such that two limits coincide.* □

6 Modular Stacking-based Points-to Analysis

We implemented a modular stacking-based points-to analysis, by instantiating the algorithm in §5. Our analysis consists of two modules: the front-end points-to analysis engine implemented in Java, and the back-end WPDS engine implemented in C. Those two modules communicated with each other by writing and reading files shared on the disk, e.g., the front-end engine generates and outputs the WPDS model on the disk for each local analysis, and the back-end engine loads the partial model, calculates the fixed-point, and outputs the intermediate values of variables to some files on the disk. These values are later read-off by the front-end analysis to enlarge the underlying problem to be solved (i.e., ϕ) and to decide the sliding window for the next iteration.

6.1 An Adapted Algorithm for CCI

When analyzing realistic applications, the induced variables (i.e., nonterminals of CAGPs) turn out to be enormous. First, we organize the analysis *method-wise*. The compound grammar productions generated from each method and all its *incoming* edges such as method calls and returns, is treated as an atomic function in the chaotic iteration. Then, we adapt the analysis to a *demand-driven* manner, such that only nonterminals $Head_{(p,\gamma)}$ are observed by the front-end analysis for their concrete values in the iterates, where $p \in AbsRef$ is any variable that is either sensitive (Def. 16) or shared among methods. To ensure soundness, we introduce an extra step of convergence check in the iterates. Such a choice turns out to be cost-effective by empirical studies, as shown in §7.

Definition 16. *For any $i \in \{1, ..., n\}$, we say that X_i is* ***sensitive*** *if, there exists $d, d' \in D^n$, with $d[i] \neq d'[i]$, and $d[j] = d[j]$ for any $j \neq i \in \{1, ..., n\}$, such that $\phi(d) \neq \phi(d')$. The indexes for all sensitive variables is denoted by $SenVar$.*

Definition 17. *Given any $ObsVar \subseteq \{1, ..., n\}$, $\tilde{d} \in (D\langle\langle Sym\mathcal{X}^*\rangle\rangle)^n$, $d \in D^n$, and $seeds \subseteq \{1, ..., n\}$. For each $1 \leq i \leq n$,*

$$\tilde{d}(ObsVar, \sigma_{seeds,d})[i] = \begin{cases} \tilde{d}[i]\sigma_{seeds,d} & \textit{if } i \in ObsVar; \\ \tilde{d}[i] & \textit{otherwise.} \end{cases}$$

and given $Var \subseteq \{1, ..., n\}$, for each $1 \leq i \leq n$,

$$(\tilde{d}\mid_{Var})[i] = \begin{cases} \tilde{d}[i], \textit{ if } i \in Var; \\ \top, \quad \textit{otherwise.} \end{cases}$$

Remark 3. For any $d \in (D\langle\langle Sym\mathcal{X}^*\rangle\rangle)^n$, we have $\phi(d) = \phi(d\mid_{SenVar})$.

In Algo. 2, we assume $ObsVar \supseteq SenVar$ and $ObsVar \supseteq seeds$ as input conditions. In contrast with Algo. 1, it differs in that, (i) at line 6, only observed variables are bound to concrete values in each iteration; (ii) at line 8, only variables from $ObsVar$ are observed; (iii) at line 14-20, an extra convergence check (i.e., $\forall f \in F. f(d) = d$) is introduced when $f.checked = 1$ for all $f \in F$. By replacing the condition "$++iteration > 0$" of the while loop (line 3) with the negation of "$f.checked == 1$ *for each* $f \in F_r$ and $mayStabilized == 1$", we obtain a terminating algorithm that computes the same limit as that of any chaotic BCI.

Theorem 5. *Assume running Algorithm 2 with a fair scheduler. We have*

1. *The limit of $d^{(1)}, d^{(2)}, ...$ reaches at $d^{(i)}$ if $f.checked = 1$ for all $f \in F$ and $mayStabilized = 1$.*
2. *The limit of $d^{(1)}, d^{(2)}, ...$ exists and there exists a chaotic BCI of F starting with $(\top, ..., \top) \in D^n$, such that its limit coincides with $d^{(i)}\sigma_{seeds,d^{(i)}}$.*

Proof. After adding the explicit convergence check, it becomes easy to anticipate the soundness. We give a proof sketch below.

1. $d^{(i)}$ is the limit, since the loop invariant "for each f, if $f.checked^{(j)} = 1$ and $mayStabilized^{(j)} = 1$, then $f(d^{(j)}) = d^{(j)}$" holds (at line 3).

2. The limit exists otherwise the descending chain property of $(D\langle\langle Sym\mathcal{X}^*\rangle\rangle)$ is violated. We define a BCI d_0, d_1, .. of F associated with a batched run F_1, F_2, ... and $d_0 = (\top, ..., \top) \in D^n$, such that $F_i = \bigcup_{1\leq k\leq i} \mathcal{F}_w^{(k)} \cup \{f \in F_r^{(i)} \mid f.checked = 1\}$. To show the two limits coincides, (i) $d_i \sqsubseteq d^{(i)}\sigma_{seeds,d^{(i)}}$ is obvious; (ii) $\forall i.\exists j.\ d_i \sqsupseteq d^{(j)}\sigma_{seeds,d^{(j)}}$ is proved by induction on i, based on Lemma 1, fairness of the scheduler, and Lemma 3. □

```
Algorithm 2: An Adapted Symbolic Sliding Window based Algorithm
$F_r := \emptyset$; $d := (\top, ..., \top) \in D^n$; $iteration := 0$; $mayStabilized := 0$;
foreach $f \in F$ do $f.checked := 0$;
while $(++iteration > 0)$ do
    $F_r := \phi(d\,|_{SenVar})$ ;
    $F_w := \mathcal{S}\mathrm{ch}(F_r, iteration)$ ;
    $\underline{d' := \mathsf{fix}(\mathcal{F}\mathrm{un}(F_w), d_{seeds})(ObsVar, \sigma_{seeds,d})}$;
    foreach $f$ in $F_w$ do $f.checked := 1$;
    $\underline{UpdatedVar := \{X_t \mid d[t] \neq d'[t], t \in ObsVar\}}$ ;
    $\underline{NewRules := \phi(d') \setminus F_r}$ ;
    foreach $f$ in $F_r \setminus F_w$ do
        if $DepVar(f) \cap UpdatedVar \neq \emptyset$ then $f.checked := 0$;
    end
    foreach $f$ in $NewRules$ do $f.checked := 0$;
    $\underline{\textbf{if } UpdatedVar \neq \emptyset \textbf{ then } mayStabilized := 0}$;
    if $f.checked = 1$ for each $f \in F_r$ then
        if $mayStabilized = 0$ then
            $mayStabilized := 1$;
            foreach $f \in F_r$ do $f.checked := 0$
        end
    end
    $d := d'$;
end
```

6.2 Correspondence with Language Features

ObsVar: Based on the aforementioned method-wise decomposition of the program and the grammar scheme and encodings in §3, the set of observed variables $ObsVar$ consists of all nonterminals $Head_{(v,m)}$, where $v \in AbsRef$ is any of the following variables of the method m,

1. all base variables (i.e., variables like x in $x.f(r_1, ..., r_n)$, or $x.f$, or $x[i]$ for any $i \in \mathbb{N}$) that correspond to $SenVar$; and
2. all variables shared among methods including (i) real arguments of any method invocation, (ii) return variables `ret` of each method if it has, and (ii) global variables; and
3. formal arguments $\mathtt{arg}_i$ of each method.

Moreover, the set of unobservable variables (that are shared among methods) contains nonterminals $N_{(p,\gamma,q)}$ where p is any return variable `ret`. The intermediate values of observed variables are stored in the memory, whereas the unobservable ones are stored on the disk. Although nonterminals induced by formal arguments (item 3) are neither sensitive nor shared among methods (under the aforementioned method-wise decomposition), they are observed for an easy experimental configuration to be shown afterwards.

$\mathcal{S}\texttt{ch}$: The analysis maintains a *workset* that contains all methods to be analyzed in the later iterates (i.e., any function f with $f.checked == 0$). The *workset* is updated as follows, for any method m, m is included into *workset* if $m \notin workset$, and

- m is a newly detected reachable method after resolving virtual calls; or
- the base variable x of any field or array reference like $x.f$ or $x[i]$ within m has been updated in the previous iteration; or
- some global variable that is referred within m has been updated in the previous iteration; or
- at any call site of some method m', real arguments passed to m has been updated in the previous iteration; or
- at any call site of m that calls some method m', the return variable of m' has been updated in the previous iteration.

Note that, the first two items correspond to line 13 in Algo. 2, and others correspond to line 11. Given $k \in \mathbb{N}$. We take each sliding window $\mathcal{F}_w$ from *workset* with $|\mathcal{F}_w| \leq k$. Besides, for an easy experimental configuration, we choose to always analyze m and m' above involved in the last two items together. Such a choice also leads to the following way of specifying symbolic seeds.

seeds: All nonterminals $Head_{(v,m)}$ are taken as symbolic *seeds*, where $v \in AbsRef$ is (i) either global variables or (ii) formal arguments $\texttt{arg}_i$ of each method. To supply the fixed-point computation with symbolic names, we make use of the grammar scheme of CAGPs in Fig. 2, and introduce extra transitions as follows for each method m in the sliding window,

$$r : \langle \Lambda, \Psi \rangle \hookrightarrow \langle \texttt{arg}_i, m\ \bar{\gamma} \rangle \text{ and } f(r) = s_{\texttt{arg}_i}$$

$$r : \langle \Lambda, m \rangle \hookrightarrow \langle \texttt{glob}, m \rangle \text{ and } f(r) = s_{\texttt{glob}}$$

where $\texttt{arg}_i$ is the i^{th} formal argument of m for $i \geq 0$, and $\texttt{glob}$ is any global variable that is referred within m, and $s_{\texttt{arg}_i}$ and $s_{\texttt{glob}}$ are symbolic names for $Head_{(\texttt{arg}_i,m)}$ and $Head_{(\texttt{glob},m)}$, respectively. $\bar{\gamma}$ is a fresh stack symbol. These rules plus rule (0) in Fig. 2 equally set the initial value of $val(\Theta)[Head_{(\texttt{arg}_i,m)}]$ to be $s_{\texttt{arg}_i}$. It goes for global variables.

ϕ: Our points-to analysis builds the call graph on-the-fly when the analysis proceeds. After any sensitive variables are updated in the previous iteration, the underlying problem to be resolved is updated in two ways,

- new reachable methods are resolved according to the updated values of the base variables for dynamic dispatch, as well as static methods that can be triggered in the current iteration; and
- new grammar productions for each method are generated according to the updated values of base variables for any field or array references referred within this method.

7 Experiments

We are aware of two implementations of WPDSs: Weighted PDS Library [4] and WPDS++ [5], and used the former in our experiments. We instantiated the algorithm presented in §5 within our Java points-to analyzer Japot (implemented in Java), with using Soot2.3.0 [18] for preprocessing from Java programs to Jimple codes, and the Weighted PDS Library as the oracle for computing the fixed point of the equation system. We evaluate Japot on the Ashes benchmark suite [17] and the DaCapo benchmark suite [2] (the "#App." column in Table 1). These applications are de facto benchmarks when evaluating Java points-to analysis. We analyze DaCapo benchmark with JDK 1.5, and Ashes benchmarks for which JDK 1.3 suffices. 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. A 2GB RAM is set for Java virtual machine when running Japot.

To measure the performance of points-to analysis, we take *call graph generation* in terms of reachable methods as client analysis. Table 1 shows the preliminary experimental results. The number of reachable methods is given in the "# Methods" column with taking Java libraries into account. The sub-column " `CHA`" is the result by conducting CHA of Spark in soot-2.3.0. The sub-column "`Japot`" gives results computed by our context-sensitive analysis, and the "# Statements" column gives the number of Jimple statements that `Japot` analyzed.

# App.	# `WPA`(s)	# `SSWA`(s)			# Acc.	# Methods		# Stmts
		win(10^4)	win(5000)	win(3000)		`CHA`	Japot	(Japot)
soot-c	1751	605	607	676	2.9	5460	5079	83,936
sablecc-j	2785	1030	1087	1274	2.6	13,055	9004	143,140
antlr	×	1401	1506	1832	∞	10,728	9133	156,913
bloat	41434	11704	12323	↺	3.4	12,928	11,090	194,063
chart	_	_	_	_	_	30,831	_	_
jython	×	9581	10305	↺	∞	14,603	12,033	202,326
pmd	×	1611	1696	1584	∞	12,485	10,406	180,170
hsqldb	2910	1037	1119	1239	2.8	9983	8394	142,629
xalan	2926	1064	1085	1255	2.7	9977	8392	141,415
luindex	3880	1329	1292	1511	3.0	10,596	8961	152,592
lusearch	4057	1369	1441	1678	2.8	11,190	9580	163,958
eclipse	×	1955	2042	2555	∞	12,703	10,404	179,179

Table 1. Scaling Japot by Symbolic Sliding Window based Algorithm. ↺: time out at 4 hours. ×: memory overflow on WPDS. _ : memory overflow on JVM.

The "# `WPA`" column gives the time in seconds of the original whole program analysis, i.e., solving contracted AGPs by computing the limit of Kleen chains.

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

[5] http://www.cs.wisc.edu/wpis/wpds++/index.php

The "# SSWA" column gives the time of the symbolic sliding window based analysis. To measure the analysis given an adjustable memory budget, we set a bound k on the number of methods that can be taken in each sliding window. The sub-column "win(10^4)", "win(5000)" and "win(3000)" gives the analysis time for $k \in \{10000, 5000, 3000\}$, respectively. As shown in the "# Acc." column, over all the experiments we performed, SSWA provided us an average 3X speedup over WPA (when $k = 5000$). Interestingly, it works almost the same when $k = 10000$ and $k = 5000$, which means that we can safely reduce the memory without affecting the performance a lot. Moreover, the analysis works well for most of applications when the memory budget is extremely shrunk (when $k = 3000$).

8 Related and Future Work

There has been a host of work on points-to analysis. A long standing difficulty in the realm is that, context-sensitivity is crucial to the analysis precision, whereas prohibitive to scalability. To our knowledge, practical points-to analysis do not handle recursive procedure calls in a context-sensitive way. On the one hand, they took cloning-based approach that has an inherit limit on analyzing recursions. On the other hand, it will become a bottleneck to precisely handle recursions. After all, scalability remains a problem even after approximating recursive procedures.

The first scalable cloning-based context-sensitive Java points-to analysis is presented in [19], where programs and analysis problems are encoded as logic rules in Datalog. The BDD (Binary Decision Diagram) based implementation, as well as approximations by collapsing loops in the call graph, enable the analysis to scale. Manu et al proposed refinement-based techniques that explores CFL-reachability to effectively detect unrealizable dataflow over pointer assignments [15, 16]. The demand- and client-driven manner serves as the key to their scalability. Loops in the call graph is again collapsed for decidability. As discussed in [6], there are often rich and large loops within the call graph, and the loss of precision is incurred after approximating recursions.

Our attempt is to design a scalable stacking-based points-to analysis for Java that precisely handles recursive procedures. We place no restrictions on recursions, by managing the program calling contexts with the unbounded pushdown stack. The work is an enhancement of our previous work [8] that tried the local analysis methods to reduce the time costs in a heuristic manner, and relied on the whole program analysis for soundness. In this work, we presented a systematic approach of scaling the stacking-based analysis as a modular analysis, with reducing both time and memory costs yet retaining precision. There are a host of work on modular analysis. We refer to [3] for an elegant review.

We didn't compare with existing cloning points-to analyzers, because at present, context-sensitivity with respect to object-oriented features, such as context-sensitive heap abstraction and context-sensitive call graph, are not supported in Japot. To provide those features demands Conditional WPDS [7]. However, techniques proposed in this work is applicable to the lifted analysis since analysis problems on Conditional WPDS is reduced to that on WPDS.

Akash et al [5] proposed a technique to improve the running time for (weighted) pushdown model checking. In their approach, dataflow equations of intra-graphs are grouped as regular expressions, and the update on those expressions are incrementally propagated through shared interfaces interprocedurally. The efficiency gain makes use of Tarjan's algorithm for efficiently computing regular expressions. Their work is a complementary to ours, and can be plugged-in as a back-end for Weighted PDS Library. Our analysis can be also regarded as incremental for only affected program parts are taken as the next sliding window.

In this paper we limit our focus to the effects of unknown program control flow on designing a precise modular stacking-based analysis. As future work, smarter iteration strategies can be introduced when choosing sliding windows, and proper procedure summaries could be generated to avoid revisiting some program parts in the iterates for a better efficiency. We anticipate that generating context-sensitive summaries for stacking-based analysis demands Conditional WPDSs. Besides, since analysis that assume a program control flow are specific cases of CAGPs, we plan to apply techniques in this paper to taint-like analysis for detecting web vulnerabilities [9]. Last but not least, modularity opens the possibility of parallization. We hope to further scale up Japot by parallization.

References

1. K. R. Apt. The essence of constraint propagation. *Theor. Comput. Sci.*, 221:179–210, June 1999.
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. P. Cousot and R. Cousot. Modular static program analysis. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 159–178, London, UK, 2002. Springer-Verlag.
4. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 77–101, London, UK, UK, 1995. Springer-Verlag.
5. A. Lal and T. W. Reps. Improving pushdown system model checking. In *CAV'06: Proceedings of the 18th International Conference on Computer Aided Verification*, pages 343–357, 2006.
6. O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *CC'06: Proceedings of the 15th International Conference on Compiler Construction*, volume 3923 of *LNCS*, pages 47–64, Vienna, Mar. 2006. Springer.
7. X. Li and M. Ogawa. Conditional weighted pushdown systems and applications. In *Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '10, pages 141–150, New York, NY, USA, 2010. ACM.
8. X. Li and M. Ogawa. Stacking-based context-sensitive points-to analysis for java. In *Proceedings of the 5th international Haifa verification conference on Hardware and software: verification and testing*, HVC'09, pages 133–149, Berlin, Heidelberg, 2011. Springer-Verlag.

9. X. Li, D. Shannon, I. Ghosh, M. Ogawa, S. P. Rajan, and S. Khurshid. Context-sensitive relevancy analysis for efficient symbolic execution. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 36–52, Berlin, Heidelberg, 2008. Springer-Verlag.
10. G. Ramalingam. *Bounded Incremental Computation.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
11. T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22:162–186, January 2000.
12. 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.
13. G. Rozenberg and A. Salomaa, editors. *Handbook of formal languages, vol. 1: word, language, grammar.* Springer-Verlag New York, Inc., New York, NY, USA, 1997.
14. S. Schwoon. *Model-Checking Pushdown Systems.* PhD thesis, 2002.
15. 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.
16. M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. *SIGPLAN Not.*, 40(10):59–76, 2005.
17. R. Vallée-Rai. Ashes suite collection. *http://www.sable.mcgill.ca/ashes.*
18. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: is tt feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.
19. J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.
20. G. Yorsh, E. Yahav, and S. Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 221–234, New York, NY, USA, 2008. ACM.