Conditional Weighted Pushdown Systems and Applications

Xin Li
School of Information Science,
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa, Japan, 923-1292
li-xin@jaist.ac.jp

Mizuhito Ogawa
School of Information Science,
Japan Advanced Institute of Science and Technology
Nomi, Ishikawa, Japan, 923-1292
mizuhito@jaist.ac.jp

Abstract

Pushdown systems are well understood as abstract models of programs with (recursive) procedures. Reps et al. recently extended pushdown systems into weighted pushdown systems, which serve as a generalized framework for solving certain kinds of meet-over-all-path problems in program analysis. In this paper, we extend weighted pushdown systems to conditional weighted pushdown systems, by further specifying conditions under which a pushdown transition rule can be applied, and show that model checking problems on conditional weighted pushdown systems can be reduced to those on weighted pushdown systems.

There are wider applications of conditional weighted pushdown systems when analyzing programs with objected-oriented features, for which weighted pushdown systems is not precise enough under a direct application. As an example, we lift a stacking-based points-to analysis for Java designed in the framework of weighted pushdown systems to a more precise counterpart in the framework of conditional weighted pushdown systems. In addition to the fundamental context-sensitivity in terms of valid paths, the lifted points-to analysis algorithm further enjoys context-sensitivity with respect to objected-oriented features, including call graph construction, heap abstraction, and heap access. These context-sensitive properties are shown to be crucial to the analysis precision in practice.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model Checking

General Terms Reliability, Verification

Keywords Weighted Pushdown Systems, Model Checking

1. Introduction

Pushdown systems (PDSs) [15] are well understood as abstract models of programs with (recursive) procedures. By encoding programs as pushdown systems, procedure calls and returns are guaranteed to be correctly paired with one another, which is called *valid paths*. A traditional context-sensitive program analysis is to compute precise analysis results, which are only (or as much as possible) involved with valid paths. We call program analysis *stack-based* if it is derived by an encoding of the program as pushdown systems or its variants.

PEPM'10, January 18–19, 2010, Madrid, Spain.

The notion of context-sensitivity bears a similarity to inline expansion, as if method calls are replaced with the body of all the callees. As such, the typical approach to obtain context-sensitivity is *cloning-based*, by creating a separate copy of a procedure for different calling contexts within a bounded call depth. It is well understood that, the cloning-based approach has an inherit limit on handling (recursive) procedure calls, and the common choice is to sacrifice context-sensitivity inside recursive procedure calls [6, 18], such as the well-known cloning-based points-to analysis for Java [18]. However, empirical study on practiced Java benchmarks shows that, more than one thousand of methods are typically contained within recursive procedures [19]. Taking Java points-to analysis as an instance, approximating recursions potentially threatens precision [8], and cloning-based points-to analysis cannot scale under deep context cloning.

Reps et al. recently extend pushdown systems into weighted pushdown systems (WPDSs), by associating a value with each pushdown transition rule. In particular, it is shown that classic Meet-Over-All-Path (MOVP) problems of certain kinds can be solved as model checking problems on weighted pushdown systems. Therefore, weighted pushdown systems is expected to serve as a generalized framework for yielding new algorithms of context-sensitive program analysis that naturally enjoy the context-sensitivity with respect to valid paths.

In this work, we extend weighted pushdown systems to conditional weighted pushdown systems (CWPDSs), by further associating each transition rule with a regular language that specifies conditions under which the transition rule can be applied. Such an extension of weighted pushdown systems is motivated by the finding that, weighted pushdown systems is not precise enough when analyzing objected-oriented programs like Java. We also show that model checking problems on CWPDSs can be reduced to model checking problems on WPDSs. Our solution is inspired by the extension of LTL model checking on pushdown systems with simple valuation to a counterpart with regular valuation.

Conditional weighted pushdown systems have a wider applications than those that weighted pushdown systems can directly applicable, by allowing an investigation on the calling contexts – the (dynamic) history of procedure calls – during the analysis. A scalable stacking-based points-to analysis for Java is proposed in the framework of weighted pushdown systems [10], which is context-sensitive in terms of valid paths. We lift this analysis to a more precise counterpart in the framework of conditional weighted pushdown systems. The lifted analysis further enjoys context-sensitivity regarding object-oriented features, such as call graph construction, heap abstraction, and heap access. These context-sensitive properties are shown to be essential to the precision of points-to analysis on large-scale programs [8, 16].

This paper makes the following contributions:

- We extend weighted pushdown systems to conditional weighted pushdown systems, and present model checking algorithms on it. Such an extension provides the ability of modelling wider application scenarios for which weighted pushdown systems are not directly applicable for enough precision.
- We present a stacking-based points-to analysis for Java, as a practical application of conditional weighted pushdown systems. In addition to valid paths, the lifted analysis further enjoys context-sensitivity with respect to heap abstraction, call graph construction, and heap access.

In the remainder of the paper: Section 2 gives a brief review on weighted pushdown systems and an application to stacking-based points-to analysis for Java. Motivations for extending weighted pushdown systems are discussed in Section 3, by illustrating occasions on which program analysis by weighted pushdown systems is not precise enough. We present conditional weighted pushdown systems and model checking algorithms on it in Section 4. A lifted stacking-based points-to analysis algorithm by conditional weighted pushdown systems is presented in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2. Preliminaries

2.1 Weighted Pushdown Systems

DEFINITION 1. *A **pushdown system** is a five-tuple $P = (Q, \Gamma, \Delta, q_0, \omega_0)$, where Q is a finite set of states called control locations, Γ is a finite stack alphabet, and $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma^*$ is a finite set of transition rules, and $q_0 \in Q$ and $\omega_0 \in \Gamma^*$ are the initial control location and stack contents, respectively. A transition rule $(p, \gamma, q, \omega) \in \Delta$ is 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 $\omega \in \Gamma^*$. A computation relation $\Rightarrow$ on configurations is defined such that $\langle p, \gamma\omega'\rangle \Rightarrow \langle q, \omega\omega'\rangle$ for all $\omega' \in \Gamma^*$ if there exists a transition rule $r : \langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle \in \Delta$, written as $\langle p, \gamma\omega'\rangle \overset{r}{\Rightarrow} \langle q, \omega\omega'\rangle$. The reflective and transitive closure of $\Rightarrow$ is denoted by $\Rightarrow^*$, and we write $c \overset{\sigma}{\Rightarrow}^n c'$ if $c \overset{r_1}{\Rightarrow} c_1 \overset{r_2}{\Rightarrow} ...c_n \overset{r_n}{\Rightarrow} c'$ for any $n \in \mathbb{N}$ and $c, c', c_i \in Q \times \Gamma^*$ with $1 \leq i \leq n$ and $\sigma = [r_1, r_2, ..., r_n]$. Given a set of configurations C, we define $pre^*(C) = \{c' \mid c' \Rightarrow^* c,$ for each $c \in C, c' \in Q \times \Gamma^*\}$, and define $post^*(C) = \{c' \mid c \Rightarrow^* c',$ for each $c \in C, c' \in Q \times \Gamma^*\}$.*

A pushdown system can be normalized (or simulated) by a pushdown system for which $|\omega| \leq 2$ for each transition rule $\langle p, \gamma\rangle \hookrightarrow \langle q, \omega\rangle$ [15]. In sequel, pushdown systems under consideration are assumed to be normalized as above.

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

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

By Def. 5, we have $\mathbf{0}$ is the greatest element.

DEFINITION 3. *A **weighted pushdown system** is a triplet $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, \omega_0)$ is a pushdown system, $S = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a bounded idempotent semiring, and $f : \Delta \to D$ is a weight assignment function.*

When developing program analysis in the framework of WPDSs, the bounded idempotent semiring is typically used to model data flows of the program. Typically, a weight element encodes the traditional program transformer, i.e., the changes of program states (in the abstract domain); $f \oplus g$ combines data flows at the meet of control flows; $f \otimes g$ composes sequential control flows; $\mathbf{1}$ denotes identity function, and $\mathbf{0}$ implies program errors.

DEFINITION 4. *Given a weighted pushdown system $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$. Assume $\sigma = [r_0, ..., r_k]$ be a sequence of pushdown transition rules for $r_i \in \Delta$ with $0 \leq i \leq k$, and $\mathtt{val}(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$. Given sets of regular configurations $C, C' \subseteq Q \times \Gamma^*$, for each configuration $c \in Q \times \Gamma^*$,*

- *the Generalized Pushdown Successor (GPS) problem is to find $\text{GPS}(c, C) = \bigoplus\{\mathtt{val}(\sigma) \mid c' \overset{\sigma}{\Rightarrow}^* c, c' \in C\}$;*
- *the Generalized Pushdown Predecessor (GPP) problem is to find $\text{GPP}(c, C) = \bigoplus\{\mathtt{val}(\sigma) \mid c \overset{\sigma}{\Rightarrow}^* c', c' \in C\}$;*
- *the Meet-Over-All-Valid-Path (MOVP) problem is to find $\text{MOVP}(C, C') = \bigoplus\{\mathtt{val}(\sigma) \mid c \overset{\sigma}{\Rightarrow}^* c', c \in C, c' \in C'\}$.*

Based on the finding that a regular set of configurations is closed under forward and backward reachability, efficient algorithms for solving the GPS and GPP problems are proposed using $\mathcal{P}$-automaton techniques [13]. MOVP problems can be solved based on the results of solving either GPS or GPP problems. We are aware of two off-the-shelf implementations of weighted pushdown model checking, Weighted PDS Library [1] and WPDS++ [2].

2.2 Stacking-based Points-to Analysis by WPDSs

We show how to yield program analysis by weighted pushdown model checking, using the points-to analysis algorithm in [10] as an example. Points-to analysis is a prerequisite of precise program analysis on objected-oriented programs. It aims at computing a points-to relation (denoted by $\mapsto$) that maps a variable of reference type to the set of objects it may point to at runtime. In particular, a context-sensitive points-to analysis distinguishes the calling contexts in which a points-to relation is valid.

We apply a context-insensitive abstraction on heap, such that a unique abstract heap object models concrete heap objects allocated at the same heap allocation site. Thus, the number of abstract heap objects are syntactically bounded to be finite. The set of abstract heap objects is denoted by `Obj`, and the set of reference variables (in the abstract domain) is denoted by `Ref`. In contrast to the cloning-based approach, calling contexts are entirely managed by the pushdown stack.

DEFINITION 5. *Let $\mathcal{P}$ be the powerset constructor. We define $D_1 = \{\lambda x.s \mid s \in \mathcal{P}(\mathtt{Obj})\}$ and $D_2 = \{\lambda x.x \cup s \mid s \in \mathcal{P}(\mathtt{Obj})\}$, and a bounded idempotent semiring $S = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, such that*

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

To be self-contained, Definition 5 recalls the bounded idempotent semiring used in the analysis, where $\lambda x.s \in D_1$ means that a reference points to the set of abstract heap objects s; and $\lambda x.x \cup s \in D_2$ means that a reference may keep unchanged along some paths and point to s along others. It is easy to see that, both the distributivity of $\otimes$ over $\oplus$ and the associativity of $\otimes$ hold.

EXAMPLE 1. *A Java code snippet is given in Fig. 1(a), and its encoding as weighted pushdown systems is given in Fig. 1(b), where o^l denotes an abstract heap object allocated at line l; `Henv` and `sp`*

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

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

```
0.  public class Main {
1.    public static void main(String[] args) {
2.        Object x1 = new String();
3.        Object x2 = new Object();
4.        Object y1 = f1(x1);
5.        Object y2 = f1(x2);
6.        System.out.println(y1.equals(y2));
7.    }
8.    public static Object f1(Object a) {
9.        return a;
10.   }
11. }
```

(a) A Java Code Snippet

$$
\begin{array}{rll}
r_0: & \langle \texttt{Henv}, \texttt{sp}\rangle \hookrightarrow \langle \texttt{Henv}, \text{main}\rangle & \texttt{id} \\
r_2: & \langle \texttt{Henv}, \text{main}\rangle \hookrightarrow \langle x_1, \text{main}\rangle & \lambda x.\{o^2\} \\
r_3: & \langle \texttt{Henv}, \text{main}\rangle \hookrightarrow \langle x_2, \text{main}\rangle & \lambda x.\{o^3\} \\
r_4: & \langle x_1, \text{main}\rangle \hookrightarrow \langle \texttt{arg}_{f_1}, f_1\ l_4\rangle & \texttt{id} \\
r'_4: & \langle \texttt{ret}_{f_1}, l_4\rangle \hookrightarrow \langle y_1, \text{main}\rangle & \texttt{id} \\
r_5: & \langle x_2, \text{main}\rangle \hookrightarrow \langle \texttt{arg}_{f_1}, f_1\ l_5\rangle & \texttt{id} \\
r'_5: & \langle \texttt{ret}_{f_1}, l_5\rangle \hookrightarrow \langle y_2, \text{main}\rangle & \texttt{id} \\
r_8: & \langle \texttt{arg}_{f_1}, f_1\rangle \hookrightarrow \langle a, f_1\rangle & \texttt{id} \\
r_9: & \langle a, f_1\rangle \hookrightarrow \langle \texttt{ret}_{f_1}, f_1\rangle & \texttt{id} \\
r_{10}: & \langle \texttt{ret}_{f_1}, f_1\rangle \hookrightarrow \langle \texttt{ret}_{f_1}, \epsilon\rangle & \texttt{id}
\end{array}
$$

(b) Weighted Pushdown System Encoding

Figure 1. A Java Code Snippet for Showing Valid Paths

are fresh symbols to denote the abstract heap environment and the program entry point, respectively; `ret` *and* `arg` *are fresh variables to denote return values and formal arguments, respectively. These variables are indexed with their method scope if necessary.*

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. For any variable $v \in$ `Ref`, $\text{MOVP}(C, @(v))(v)$ *returns the set of objects that v may point to, where $C = \{\langle \texttt{Henv}, \texttt{sp}\rangle\}$ and $@(p) = \{\langle p, \omega\rangle \mid \omega \in \Gamma^*\}$. The points-to analysis precisely infers $\{y_1 \mapsto o^2\}$ and $\{y_2 \mapsto o^3\}$. In contrast, an context-insensitive analysis would infer mix them. This example illustrates the featured power of stacking-based program analysis regarding valid paths.*

3. Motivations

We use Example 2 to illustrate typical occasions on which the aforementioned points-to analysis by WPDSs is not precise enough.

EXAMPLE 2. *We consider the following analysis clients on the Java code snippet in Fig. 2:*

- *whether variables c and d are aliased, i.e., whether these variables point to the same objects at runtime;*
- *whether downcasts at line 6 and 10 are safe, i.e., whether the returned value of calling get() can only point to Integer objects at line 6 and String objects at line 10, respectively.*

We can conclude that at runtime, (i) c and d are not aliased, and (ii) both downcasts at line 6 and 10 are safe. However, we cannot infer the correct answer by performing the aforementioned stacking-based points-to analysis, which imprecisely infers that, any of variables c, d, int_i, and str_s may point to $\{o^{21}, o^{31}\}$.

The imprecision is due to the fact that, some kinds of data flows are sensitive to the calling history because of object-oriented features, but such dependency is missed when the program is modelled

```
1.  public class Main {
2.    public static void main(String[] args) {
3.      A a = new A();
4.      Object c = foo(a);
5.      Object i = A.get(a);
6.      Integer int_i = (Integer) i;
7.      A b = new B();
8.      Object d = foo(b);
9.      Object s = B.get(b);
10.     String str_s = (String) s;
11.   }
12.   public static Object foo(A x) {
13.     return x.set();
14.   }
15. }
16. public class A {
17.   Object f;
18.   Object[] arr;
19.   A() { this.arr = new Object[1]; }
20.   public Object set() {
21.     this.f = new Integer(5);
22.     this.arr[0] = this.f;
23.     return this.f;
24.   }
25.   public static Object get(A x) {
26.     return x.arr[0];
27.   }
28. }
29. public class B extends A {
30.   public Object set() {
31.     this.f = new String();
32.     this.arr[0] = this.f;
33.     return this.f;
34.   }
35. }
```

Figure 2. A Java Code Snippet for Ilustrating the Needs of Conditional Extension of Weighted Pushdown Systems

as weighted pushdown systems. For instance, the failure of proving downcast safety for this example is attributed to, (i) context-insensitive heap abstraction (at line 19), such that o^3.arr and o^7.arr point to the same object o^{19}; and (ii) mixing field read at line 26 under different calling contexts.

Fig. 3 illustrates the imprecision caused by the construction of a context-insensitive call graph regarding the program fragments. Each call edge is labelled with (i), and its corresponding return edge is labelled with $(i)'$, for $1 \leq i \leq 4$. There are two valid control flows (1)-(2)-$(2)'$-$(1)'$ and (3)-(4)-$(4)'$-$(3)'$ starting from the main() method. By encoding the program as (weighted) pushdown systems, it is guaranteed that any call and return edges are correctly paired with one another. However, invalid control flows of (1)-(4)-$(4)'$-$(1)'$ and (3)-(2)-$(2)'$-$(3)'$ still remain when producing the analysis results, because control flows along edges (2) and (4) depend on the type of objects x may point to at line 13, i.e., obeying to the semantics of dynamic dispatch.

4. Conditional Weighted Pushdown Model Checking

4.1 Conditional Weighted Pushdown Systems

We extend pushdown systems to conditional pushdown systems, by further associating each transition rule with conditions that specify

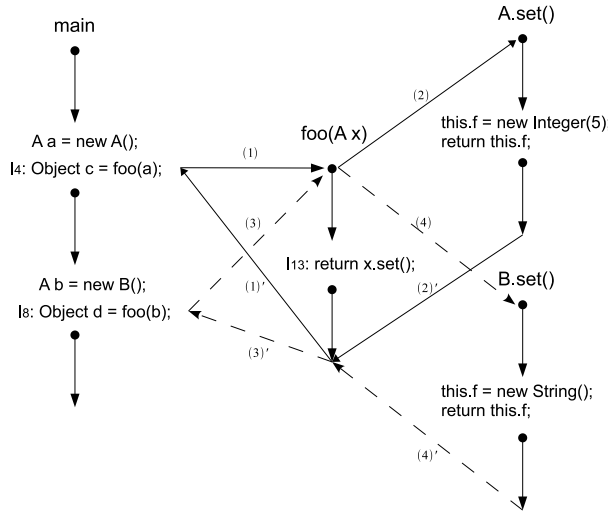

Figure 3. Illustrating Context-Insensitive Call Graph Construction

when this rule can be applied, and correspondingly lift weighted pushdown systems to conditional weighted pushdown systems.

DEFINITION 6. *A **conditional pushdown system** is a 6-tuple $P_c = (Q, \Gamma, \Delta_c, \mathcal{C}, q_0, \omega_0)$, where Q is a finite set of control locations, Γ is a finite stack alphabet, $\mathcal{C}$ is a finite set of regular languages over Γ, $\Delta_c \subseteq Q \times \Gamma \times \mathcal{C} \times Q \times \Gamma^*$ is the set of transition rules, and $q_0 \in Q$ and $\omega_0 \in \Gamma^*$ are the initial control location and stack contents, respectively. A transition rule $(p, \gamma, L, q, \omega) \in \Delta_c$ is written as $\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \omega\rangle$. A computation relation $\Rightarrow_c$ on configurations is defined such that $\langle p, \gamma\omega'\rangle \Rightarrow_c \langle q, \omega\omega'\rangle$ for all $\omega' \in \Gamma^*$ if there exists a transition rule $\langle p, \gamma\rangle \overset{L}{\hookrightarrow} \langle q, \omega\rangle$ and $\omega' \in L$, written as $\langle p, \gamma\omega'\rangle \overset{r}{\Rightarrow}_c \langle q, \omega\omega'\rangle$. The reflective and transitive closure of $\Rightarrow_c$ is denoted by $\Rightarrow_c^*$, and for any $n \in \mathbb{N}$, we write $c \overset{\sigma}{\Rightarrow}_c^n c'$ if $c \overset{r_1}{\Rightarrow}_c c_1 \overset{r_2}{\Rightarrow}_c ...c_n \overset{r_n}{\Rightarrow}_c c'$ for $c, c', c_i \in Q \times \Gamma^*$ with $1 \leq i \leq n$ and $\sigma = [r_1, r_2, ..., r_n]$.*

Note that, for the sake of efficiency, we do not play a condition on the whole stack, because the topmost stack symbol is already specified by the underlying pushdown transition rule.

DEFINITION 7. *A **conditional weighted pushdown system** is a triplet $W_c = (P_c, S, f)$, where $P_c = (Q, \Gamma, \mathcal{C}, \Delta, q_0, \gamma_0)$ is a conditional pushdown system, $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, and $f : \Delta \rightarrow D$ is a function that designates a weight to each pushdown transition rule.*

We lift model checking problems in Definition 4 to conditional weighted pushdown system, denoted by GPS_c, GPP_c and MOVP_c, respectively, which can be solved by a reduction to model checking problems on weighted pushdown systems.

Given a regular language L, a *condition automaton* with respect to L is a deterministic finite state automaton, denoted by $A = (S, \Sigma, \delta, \dot{s}, F)$, where S is the finite set of states, Σ is the finite input alphabet, $\delta : S \times \Sigma \rightarrow S$ is the total transition function, $\dot{s} \in S$ is the initial state, and $F \subseteq S$ is the set of final states. The language recognized by the automaton A is denoted by $\mathcal{L}(A)$. Without loss of generality, condition automata under consideration have no ϵ transitions. We extend δ to be the type of $S \times \Sigma^* \rightarrow S$ in the standard way, such that for each $s \in S$, $\delta(s, \varepsilon) = s$ and $\delta(s, \omega\alpha) = \delta(\delta(s, \omega), \alpha)$ for each $\omega \in \Sigma^*$ and $\alpha \in \Sigma$.

Assume condition automata $A_i = (S_i, \Sigma, \delta_i, \dot{s}_i, F_i)$ with $i \in \{1, 2\}$. We define the product automaton A of A_1 and A_2, denoted by $A_1 \times A_2$, such that $A_1 \times A_2 = (S_1 \times S_2, \Sigma, \delta, (\dot{s}_1, \dot{s}_2), F)$, where $\delta(s_1, s_2) = (\delta_1(s_1), \delta_2(s_2))$ for $s_i \in S_i$ with $i \in \{1, 2\}$, and $F = F_1 \times S_2 \cup S_1 \times F_2$. Let $A = \{A_1, ..., A_n\}$ be a finite set of condition automata that have the same input alphabet. We denote by $\Pi_{1 \leq i \leq n} A_i$ the product $A_1 \times A_2 \times ... \times A_n$, and by $\mathbf{s}_i$ the i^{th} component of a state $\mathbf{s}$ from $\Pi_{1 \leq i \leq n} A_i$, i.e., $\mathbf{s}_i \in A_i$. It is easy to see that, $\Pi_{1 \leq i \leq n} A_i$ is a condition automaton.

DEFINITION 8. *Let ϕ be a function that designates a condition automaton A with respect to any given regular language L, such that (i) A has minimal states for $\mathcal{L}(A) = L$; and (ii) for regular languages L, L' (over the same alphabet), condition automata $\phi(L)$ and $\phi(L')$ are identical if $L = L'$, otherwise the states of $\phi(L)$ and $\phi(L')$ are disjoint.*

We also denote by $\text{REV}(\omega)$ the reverse of a word ω from some language, and denote by L^R the set $\{\text{REV}(\omega) \mid \omega \in L\}$ of all reversed words from L.

4.2 Solving Conditional WPDMC

As shown in Fig. 4, Algo. TRANS translates a conditional weighted pushdown system to a corresponding weighted pushdown system, by extending the stack alphabet. The idea of the translation is to synchronize the underlying (weighted) pushdown system and the product automaton $\Pi_{0 \leq i \leq n} A_i = (\widehat{S}, \Gamma, \widehat{\delta}, \dot{\mathbf{s}}, \widehat{F})$, with reading the stack symbols bottom-up. In sequel, we fix a CWPDS $W_c = (P_c, S, f)$ with $P_c = (Q, \Gamma, \mathcal{C}, \Delta, q_0, \gamma_0)$, and let $W_t = (P, S, f')$ be the WPDS translated from W_c by Algo. TRANS with $P = (Q, \Gamma', \Delta, q_0, \gamma_0')$.

DEFINITION 9. *A configuration $\langle q, (\gamma_n, \mathbf{s}_n)(\gamma_{n-1}, \mathbf{s}_{n-1})...(\gamma_0, \mathbf{s}_0)\rangle$ of W_t is **consistent** if $\mathbf{s}_0 = \dot{\mathbf{s}}$ and $\widehat{\delta}(\mathbf{s}_i, \gamma_i) = \mathbf{s}_{i+1}$ for $0 \leq i < n$. The set of consistent configurations of W_t is denoted by $\mathtt{Confs}$.*

DEFINITION 10. *We define a function $\rho : \mathtt{Confs} \rightarrow (Q \times \Gamma)^*$ that maps each consistent configuration of W_t to some configuration of W_c, such that for any $\langle q, (\gamma_n, \mathbf{s}_n)(\gamma_{n-1}, \mathbf{s}_{n-1})...(\gamma_0, \mathbf{s}_0)\rangle \in \mathtt{Confs}$, $\rho(\langle q, (\gamma_n, \mathbf{s}_n)(\gamma_{n-1}, \mathbf{s}_{n-1})...(\gamma_0, \mathbf{s}_0)\rangle) = \langle q, \gamma_n\gamma_{n-1}...\gamma_0\rangle$.*

LEMMA 1. *ρ is bijective.*

Proof By the definition of ρ and consistent configurations, ρ is easily seen to be injective, based on the fact that the product automaton $\Pi_{1 \leq i \leq n} A_i$ is deterministic. Furthermore, ρ can be proved surjective, because $\widehat{\delta}$ is total. □

According to Lemma 1, we are able to define the inverse of ρ, denoted by ρ^{-1}.

LEMMA 2. *Given a computation sequence $c_s \overset{\sigma}{\Rightarrow}_c^* c_t$ of W_c. Let $c_s' = \rho^{-1}(c_s)$ and $c_t' = \rho^{-1}(c_t)$. There exists σ' of W_t such that $c_s' \overset{\sigma'}{\Rightarrow}^* c_t'$ with $\mathtt{val}(\sigma) = \mathtt{val}(\sigma')$.*

Proof By an induction on the transition steps m of $c_s \Rightarrow_c^m c_t$. When $m = 0$, the proof is trivial. Assume that the lemma holds for $m = k$, i.e., for $c_s \overset{\sigma}{\Rightarrow}_c^k c_t$, there exists σ' such that $c_s' \overset{\sigma'}{\Rightarrow}^* c_t'$ with $\mathtt{val}(\sigma) = \mathtt{val}(\sigma')$ of W_t. Let $c_t = \langle q, \gamma_n\gamma_{n-1}...\gamma_0\rangle$, and $c_t' = \rho^{-1}(c_t) = \langle q, (\gamma_n, \mathbf{s}_n)(\gamma_{n-1}, \mathbf{s}_{n-1})... (\gamma_0, \mathbf{s}_0)\rangle$ with $\mathbf{s}_0 = \dot{\mathbf{s}}$.

Assume that there exists a transition rule $r : \langle q, \gamma_n\rangle \overset{L_i}{\hookrightarrow} \langle p, \omega\rangle \in \Delta_c$, such that $c_t \overset{r}{\Rightarrow}_c c$ for some configuration $c \in Q \times \Gamma$. By the definition of $\Rightarrow_c$, we have $c = \langle p, \omega\gamma_{n-1}...\gamma_0\rangle$ and $\gamma_0...\gamma_{n-1} \in \mathcal{L}(A_i)$, where $A_i = \phi(L_i^R)$. Since $\widehat{\delta}(\mathbf{s}_0, \gamma_0...\gamma_{n-1})$ = $\mathbf{s}_n$, we know $(\mathbf{s}_n)_i$ is a final state of A_i. By Algo. TRANS, there

exists $r' : \langle q, (\gamma_n, \mathbf{s}_n)\rangle \hookrightarrow \langle p, \omega'\rangle \in \Delta'$, such that $f(r') = f(r)$. Since $\mathtt{val}(\sigma{\cdot}r) = \mathtt{val}(\sigma) \otimes f(r)$, we have $\mathtt{val}(\sigma{\cdot}r) = \mathtt{val}(\sigma'{\cdot}r')$ by induction hypothesis.

Furthermore, there exists $c'_t \overset{r'}{\Rightarrow} c'$ with $c' = \langle p, \omega'(\gamma_{n-1}, \mathbf{s}_{n-1}) \ldots(\gamma_0, \mathbf{s}_0)\rangle$. It is not hard to see that $c' = \rho^{-1}(c)$, by a case analysis on ω. For instance, if $\omega = \gamma'_n \in \Gamma$, then $\omega' = (\gamma'_n, \mathbf{s}_n) \in \Gamma'$ by the algorithm construction and thus $c' = \rho^{-1}(c)$. We can conclude the lemma holds when $m = k + 1$, and thus prove the lemma. □

LEMMA 3. *Given a computation sequence $c_s \overset{\sigma}{\Rightarrow}^* c_t$ of W_t where c_s and c_t are consistent. Let $c'_s = \rho(c_s)$ and $c'_t = \rho(c_t)$. There exists σ' of W_c such that $c'_s \overset{\sigma'}{\Rightarrow}^*_c c'_t$ with $\mathtt{val}(\sigma) = \mathtt{val}(\sigma')$.*

Proof Similarly to the proof of Lemma 2, by an induction on the transition steps m of $c_s \overset{\sigma}{\Rightarrow}^m c_t$. □

THEOREM 1. *Given configurations $C_s, C_t \subseteq Q \times \Gamma^*$ of W_c, and configurations $C'_s, C'_t \subseteq Q \times (\Gamma')^*$ of W_t, such that $C'_s = \{\rho^{-1}(c) \mid c \in C_s\}$ and $C'_t = \{\rho^{-1}(c) \mid c \in C_t\}$. We have*

$$\text{MOVP}_c(C_s, C_t) = \text{MOVP}(C'_s, C'_t)$$

Proof Because $\oplus$ is idempotent and commutative, the proof is straightforward by Lemma 2 and 3, according to the definition of the MOVP problems. □

4.3 Discussions

Consider the translation algorithm in Fig. 4, it is easily seen that $|\Gamma'| \leq |\Gamma| \times |\widehat{S}|$ and $|\Delta'| \leq |\Delta| \times |\widehat{S}|$, because the product automaton $\Pi_{1\leq i\leq n}\, A_i$ is deterministic by construction. Let $|H|$ be the length of the longest descending chain of the weighted domain that may occur in the problem instance. Let $\mathcal{Q}$ and $\rightarrow_0$ be states and transitions of the $\mathcal{P}$-automaton accepting the given regular set of configurations C, respectively. According to the complexity results of solving WPDSs [13], by extending the stack alphabet, (i) the time complexity of computing $\text{pre}^*(C)$ for W_c is $\mathcal{O}(|\widehat{S}|\ |\mathcal{Q}|^2\ |\Delta|\ |H|)$, and (ii) the time complexity of computing $\text{post}^*(C)$ is $\mathcal{O}((|\widehat{S}|^2\ |Q|\ |\Delta|\ n_2 + |\widehat{S}|\ |Q|\ |\Delta|\ n_1 + |Q|\ n_0)\ |H|)$, where $n_2 = |\{\langle p', \gamma'\rangle \mid \langle p, \gamma\rangle \overset{A}{\hookrightarrow} \langle p', \gamma'\gamma''\rangle \in \Delta\}|$, $n_1 = |\mathcal{Q}\backslash Q|$, and $n_0 = |\ \rightarrow_0\ |$. We can conclude that the time complexity increase is linear in $|\widehat{S}|$ for solving $\text{pre}^*(C)$, and polynomial in $|\widehat{S}|$ for solving $\text{post}^*(C)$, after extending WPDSs to CWPDSs.

There is an alternative approach to conducting the synchronization algorithm in Fig. 4, by extending control locations of the underlying pushdown system. The construction resembles the extension on the stack alphabet. First, a product automaton $\Pi_{1\leq i\leq n}\, A_i = (\widehat{S}, \Gamma, \widehat{\delta}, \dot{\mathbf{s}}, \widehat{F})$ over all regular conditions is built, and then for each transition rule $r : \langle p, \gamma\rangle \overset{A_i}{\hookrightarrow} \langle q, \omega\rangle \in \Delta$ with $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$, the transition rules Δ' of the translated WPDS W_t are generated as follows: for each $\mathbf{s} \in \widehat{S}$, Δ' consists of

$$\begin{cases} \langle (p, \mathbf{s}), \gamma\rangle \hookrightarrow \langle (q, \mathbf{s}), \nu\rangle, & \text{if } r : \langle p, \gamma\rangle \overset{A_i}{\hookrightarrow} \langle q, \nu\rangle \\ & \text{and } \mathbf{s}_i \in F_i; \\ \langle (p, \mathbf{s}), \gamma\rangle \hookrightarrow \langle (q, \mathbf{s}), \epsilon\rangle, & \text{if } r : \langle p, \gamma\rangle \overset{A_i}{\hookrightarrow} \langle q, \epsilon\rangle \\ & \text{and } \mathbf{s}_i \in F_i; \\ \langle (p, \mathbf{s}), \gamma\rangle \hookrightarrow \langle (q, \mathbf{t}), \alpha\beta\rangle, & \text{if } r : \langle p, \gamma\rangle \overset{A_i}{\hookrightarrow} \langle q, \alpha\beta\rangle \\ & \text{and } \mathbf{s}_i \in F_i \text{ and } \mathbf{t} = \widehat{\delta}(\mathbf{s}, \beta). \end{cases}$$

Let $W_t = (P, S, f')$ with $P = (Q', \Gamma, \Delta', q'_0, \gamma_0)$. We know $|Q'| \leq |Q| \times |\widehat{S}|$ and $|\Delta'| \leq |\Delta| \times |\widehat{S}|$, because the product automaton $\Pi_{1\leq i\leq n}\, A_i$ is deterministic. By extending control locations, (i) the time complexity of computing $\text{pre}^*(C)$ for W_c is $\mathcal{O}(|\widehat{S}|^3\ |\mathcal{Q}|^2\ |\Delta|\ |H|)$, and (ii) the time complexity of computing $\text{post}^*(C)$ is $\mathcal{O}((|\widehat{S}|^3\ |Q|\ |\Delta|\ n_2 + |\widehat{S}|^2\ |Q|\ |\Delta|\ n_1 + |\widehat{S}|\ |Q|\ n_0)\ |H|)$. Therefore, the translation by extending the stack alphabet is more efficient than the translation by extending control locations in principle.

5. Applications

CWPDSs provide the facility of modelling wider application scenarios than those that WPDSs are directly applicable. These application scenarios beyond the scope of WPDSs are characterized by a dependency on the dynamic calling histories of the program. Such a well-known application is analyzing programs with local security checks on the state of the stack frames at runtime, e.g., the *stack inspection* mechanism in the Java sandbox model for security.

Taking the simple stack inspection in Netscape 3.0 as an instance, when a sensitive operation is to be launched, the stack inspection system will check the calling stack in a top-down manner: (i) the access would be allowed if a procedure with enabled privileges is found, and (ii) an exception would be thrown otherwise. The stack inspection properties can be modelled as regular languages, and it is straightforward to yield algorithms for program analysis with stack inspection by CWPDSs.

In the following of this section, we present a new practical application of CWPDSs to program analysis with object-oriented features, using points-to analysis as an example. Points-to analysis for object-oriented languages is a challenging task, because call graph construction and points-to analysis are inter-dependent due to dynamic language features like late binding.

5.1 Lifting Stacking-based Points-to Analysis by CWPDSs

We lift the stacking-based context-sensitive points-to analysis by weighted pushdown systems [9] to a more precise counterpart in the framework of conditional weighted pushdown systems. The lifted analysis is also field-sensitive and flow-insensitive, with call graph constructed on-the-fly.

Our analysis is an iterative procedure that computes two global data structures: the call graph $\mathcal{G}$ and the points-to relation $\mathbf{R}$ of the program. Initially, both the call graph and points-to relation are empty sets. The analysis starts with analyzing the program entry points (denoted by M_0), and updates $\mathcal{G}$ and $\mathbf{R}$ until convergence. In each iterative cycle,

- First, the program (i.e., the reachable methods detected so far) is encoded as conditional weighted pushdown systems;
- Second, points-to information on the partial program is detected by conditional weighted pushdown model checking;
- Third, obeying to the semantics of Java virtual machine, new call edges, as well as new reachable methods, are (potentially) discovered according to the updated points-to information.

Abstraction

DEFINITION 11. *We denote by $\mathcal{C}$ the set of classes, and denote by Ψ the set of method signatures. A method is identified by a pair of its enclosing class $C \in \mathcal{C}$ and method signature $\psi \in \Psi$, denoted by $C.\psi$. The set of method identifiers is denoted by $\mathcal{C}.\Psi \subseteq \mathcal{C} \times \Psi$.*

DEFINITION 12. *Let $\mathcal{L}$ be the set of program line numbers, and let $\mathtt{RetPoint} \subseteq \mathcal{C}.\Psi \times \mathcal{L}$ be the set of return points of method invocations. We define $\mathbb{C} = (\mathcal{C}.\Psi \times \{\_\}).\mathtt{RetPoint}^*$ as the set of abstract calling contexts, where "$\_$" is a fresh symbol that indicates anywhere. For $(m, \_)\omega \in \mathbb{C}$ with $\omega = (m_n, l_n)\ldots(m_1, l_1)(m_0, l_0)$, we define $\mathtt{tail}((m, \_)\omega) = \omega$, and extend this this function to a set of calling contexts element-wise.*

A return point $(m, l) \in \texttt{RetPoint}$ refers to a method invocation site at line l in the method m. Being flow-insensitive, a calling context $(m, \_)(m_n, l_n)...(m_1, l_1)(m_0, l_0) \in \mathbb{C}$ is with respect to the method m that is called most recently, rather than specific program execution points.

DEFINITION 13. *Let $C \subseteq \mathbb{C}$. We denote by $\sim$ an equivalence relation on C which partitions C into disjoint (nonempty) equivalence classes with a finite index. The set of all equivalence classes in C with respect to $\sim$ is denoted by $C/_\sim$.*

DEFINITION 14. *A context-sensitive call graph of a program is $\mathcal{G} = (M, \boldsymbol{E})$, where $M \subseteq \mathcal{C}.\Psi$ and $\boldsymbol{E} \subseteq M \times \mathcal{L} \times (\mathcal{P}(\mathbb{C}) \cup \{\_\}) \times M$. An element $(m, l, L, m') \in \boldsymbol{E}$ is a context-sensitive call edge, written as $m \overset{l,L}{\rightarrow} m'$. We define the set of calling contexts of a method m as $\text{ACC}(m) = \{(m, \_)(m_n, l_n)...(m_1, l_1)(m_0, l_0) \in \mathbb{C} \mid m_0 \overset{l_0,L_0}{\rightarrow} m_1 \overset{l_1,L_1}{\rightarrow} ... \; m_n \overset{l_n,L_n}{\rightarrow} m\}$.*

DEFINITION 15. *We define the set of abstract heap objects as $\mathit{Obj} = (\mathcal{L} \cup \{\_\}) \times \mathcal{C}.\Psi \times \mathcal{C} \times (\mathcal{P}(\mathbb{C}))$, and define $\boldsymbol{R} : \mathit{Ref} \times \mathcal{P}(\mathbb{C}) \rightarrow 2^{\mathit{Obj}}$ be the function that stores the points-to relation.*

In Java, a heap object is a dynamically created instance of either a class or an array. Reference variables are typically local variables, method parameters, array references, and static or instance fields of reference types. Fields and array references can be regarded as global variables. The abstraction on heap objects in Def. 15 is context-sensitive, and $\texttt{Obj}$ is finite because $\mathbb{C}/_\sim$ has a finite index.

Furthermore, we take an over-abstraction on arrays such that indices of an array is ignored, i.e., members of an array are not distinguished. We denote by $[\![o]\!]$ the unique representative for all members of an array instance o. After bounding the set of abstract heap objects to be finite, the nesting of arrays and field references become finite correspondingly. In contrast to cloning-based approach, there is a unique abstract reference for each local reference variable in the analysis, and global references are cloned for methods inside which they are referred to.

Modelling

Let $\texttt{Stmt}$ be the set of program statements. We denote by $\mathcal{A}[\![\_]\!] : \texttt{Stmt} \rightarrow \mathcal{P}(\hookrightarrow)$ the function that translates the program into transition rules of conditional weighted pushdown systems. Fig. 5 gives the translation schemes on statements at line $l (\in \mathcal{L})$ in the method $C.\psi$, and these statements do not contain explicit method invocations. For simplicity, we omit the weight associated with a transition rule r if $f(r) = \texttt{id}$. In the traditional way of modelling a program as a pushdown system, global variables are explicitly passed as parameters along procedure calls and returns. In contrast, we model the heap memory as the global data structure (i.e., **R**). The heap memory provides global references with cached data flows (i.e., A_g, A_f), when they are locally referred inside methods (only necessary for field read).

Table 6 gives translation schemes on statements that contains explicit method invocations, where A_c denotes method calls, A_r denotes method returns, and A_t denotes data flows from return points to the corresponding calling procedures. Note that, the heap environment $\texttt{Henv}$ of the program is explicitly passed (as a parameter) among any procedure calls and returns, to hold the thread control of the program.

Let $W_c = (P_c, S, f)$ be the conditional weighted pushdown system encoded from the target program, where $P_c = (Q, \Gamma, \mathcal{C}, \Delta, q_0, \omega_0\})$. The set of control locations Q is encoded from $\texttt{Ref} \cup \{\texttt{Henv}\}$. The stack alphabet Γ is encoded from $\mathcal{C}.\Psi \cup \texttt{RetPoint}$. The choice of the bounded idempotent semiring is the same as S in Def. 5. Let $q_0 = \texttt{Henv}$ and $w_0 = \texttt{sp}$. A set δ of dummy transition rules leading to the actual program entry points is introduced, such that

$$
\begin{array}{l}
\mathcal{A}[\![z = r_0.f(r_1, ..., r_n)]\!] \;=\; A_c \cup A_r \cup A_t \\
\text{where } A_c = \{\langle r_0, C.\psi\rangle \overset{\texttt{tail}(L)}{\hookrightarrow} \langle \texttt{this}^{C'.\psi'}, C'.\psi' \; \texttt{rp}\rangle\} \\
\qquad \cup \{\langle \texttt{Henv}, C.\psi\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, C'.\psi' \; \texttt{rp}\rangle\} \\
\qquad \cup \bigcup_{r_i \in \texttt{Ref}} \{\langle r_i, C.\psi\rangle \overset{\texttt{tail}(L)}{\hookrightarrow} \langle \texttt{arg}_i{}^{C'.\psi'}, C'.\psi' \; \texttt{rp}\rangle\} \\
\qquad A_r = \{\langle \texttt{ret}^{C'.\psi'}, C'.\psi'\rangle \overset{\texttt{tail}(L)}{\hookrightarrow} \langle \texttt{ret}^{C'.\psi'}, \texttt{rp}\rangle\} \\
\qquad \cup \{\langle \texttt{Henv}, C'.\psi'\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, \texttt{rp}\rangle\} \\
\qquad A_t = \{\langle \texttt{ret}^{C'.\psi'}, \texttt{rp}\rangle \overset{\texttt{tail}(L)}{\hookrightarrow} \langle z, C.\psi\rangle\} \\
\qquad \cup \{\langle \texttt{Henv}, \texttt{rp}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, C.\psi\rangle\} \\
\qquad \text{for each } C'.\psi' \text{ and } L \text{ with } (C.\psi, l, L, C'.\psi') \in \mathbf{E}, \text{ and} \\
\qquad \psi' \text{ is the method signature of } f, \text{ and } \texttt{rp} = (C.\psi, l).
\end{array}
$$

$$
\begin{array}{l}
\mathcal{A}[\![z = C'.f(r_1, ..., r_n)]\!] \;=\; A_c \cup A_r \cup A_t \\
\text{where } A_c = \{\langle \texttt{Henv}, C.\psi\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, C'.\psi' \; \texttt{rp}\rangle\} \\
\qquad \cup \bigcup_{r_i \in \texttt{Ref}} \{\langle r_i, C.\psi\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{arg}_i{}^{C'.\psi'}, C'.\psi' \; \texttt{rp}\rangle\} \\
\qquad A_r = \{\langle \texttt{ret}^{C'.\psi'}, C'.\psi'\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{ret}^{C'.\psi'}, \texttt{rp}\rangle\} \\
\qquad \cup \{\langle \texttt{Henv}, C'.\psi'\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, \texttt{rp}\rangle\} \\
\qquad A_t = \{\langle \texttt{ret}^{C'.\psi'}, \texttt{rp}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle z, C.\psi\rangle\} \\
\qquad \cup \{\langle \texttt{Henv}, \texttt{rp}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, C.\psi\rangle\} \\
\qquad \text{for each } C'.\psi' \text{ with } (C.\psi, l, \_, C'.\psi') \in \mathbf{E}, \text{ and} \\
\qquad \psi' \text{ is the method signature of } f, \text{ and } \texttt{rp} = (C.\psi, l).
\end{array}
$$

Figure 6. $\mathcal{A}[\![\_]\!] : \texttt{Stmt} \rightarrow \mathcal{P}(\hookrightarrow)$

$\delta = \{\langle \texttt{Henv}, \texttt{sp}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, C.\psi\rangle \mid C.\psi \in M_0\}$, and $f(r) = \texttt{id}$ for each r from δ.

Analyzing

Points-to information is detected as model checking problems on the CWPDS encoded from the target program, as given in Def. 16.

DEFINITION 16. *Let $W_c = (P_c, S, f)$ be the conditional weighted pushdown system encoded from the target program, where $P_c = (Q, \Gamma, \mathcal{C}, \Delta, q_0, \omega_0\})$. For any reference $v \in \mathit{Ref}$ in the method m,*

$$\boldsymbol{R}(v, L) = \text{MOVP}_c(C, L)(v)$$

for each $L \in \text{ACC}(m)/_\sim$, where $C = \{\langle q_0, \omega_0\rangle\}$.

Solving the MOVP_c problem of W_c is reduced to solving the MOVP problem of the WPDS W_t translated from W_c by the algorithm in Fig. 4. To solve $\text{MOVP}(C_s, C_t)$ on W_t, we can (i) first compute $\text{post}^*(C_s)$, and then (ii) read out and combine the value of all paths between C_s and C_t. Let $H = 2^{|\texttt{Obj}|}$ be the length of the longest descending chain of the weighted domain, and T the time to perform either $\otimes$ or $\oplus$. Assume the equivalence relation $\sim$ taken in the analysis has a finite index k. The time required to perform step (ii) can be ignored, and the worst case time complexity of performing step (i) is $\mathcal{O}(k\,|\widehat{\mathcal{S}}|^2\,|\texttt{Ref}|^2\,|\texttt{Stmt}|\,|\texttt{Obj}|\,|\mathcal{C}.\Psi|\,H\,T)$, where $\widehat{\mathcal{S}}$ is states of the product of all condition automata of W_c.

Provided with the newly-detected points-to information, the call graph is potentially updated with new call edges. Briefly, for each statement s in the method $C.\psi$ that dispatches a method invocation,

$$\mathbf{E} = \mathbf{E} \cup \begin{cases} \{(C.\psi, l, L, C'.\psi')\}, & \text{if } s \text{ contains } r_0.f(r_1, ..., r_n) \text{ and the method } C'.\psi' \text{ will be called according to the JVM semantics,when } o \in \mathbf{R}(r_0, L) \text{ for } r_0 \in \texttt{Ref}. \\ \{(C.\psi, l, \_, C'.\psi')\}, & \text{if } s \text{ contains } C'.f(r_1, ..., r_n) \text{ for } C' \in \mathcal{C}. \end{cases}$$

where ψ' is the method signature of f.

5.2 Examples

We illustrate how to conduct the lifted points-to analysis, using the Java code snippet in Example 2.

DEFINITION 17. *Given an alphabet Γ and a fixed k, and let $\mathtt{kind}(\omega)$ denote the number of distinguished symbols appearing in $\omega \in \Gamma^*$. For a word $\omega = \gamma_0\gamma_1...\gamma_n \in \Gamma^*$, we denote by $\omega(k)$ the prefix $\omega' = \gamma_0\gamma_1...\gamma_m$ of ω with $0 \leq m \leq n$, such that $\mathtt{kind}(\omega') \leq k$ and $\mathtt{kind}(\omega'\gamma_{m+1}) > k$ when $m < n$. We define an equivalence relation $\sim$ on Γ^* such that $x \sim y$ for $x, y \in \Gamma^*$ iff $x(k)$ and $y(k)$ are the same.*

One key issue of conducting program analysis by conditional weighted pushdown systems is choosing an appropriate equivalence relation $\sim$ that finitely partitions calling contexts. In practice, it would induce a tradeoff between precision and scalability. Def. 17 provides an example of choosing $\sim$, with taking Kleene star into account. It is easily seen that, the insights behind is how deep the calling context would be looked into, and we will fix $k = 3$ when illustrating the example.

Fig. 7 gives the core parts of the encoding (organized method-wise) of the program in Fig. 2 as a CWPDS. Because program lines are globally numbered, we simply write l for a return point (m, l), and write $\lceil l, \mathtt{C}, L\rceil$ for an abstract heap object $(l, m, \mathtt{C}, L)$. In this example, the set $\mathcal{C}$ of regular conditions is $\{\{l_3\}, \{l_4\}, \{l_5\}, \{l_7\}, \{l_8\}, \{l_9\}, \{l_{13}l_4\}, \{l_{13}l_8\}, \ \mathbb{C}\}$. Applying a context-sensitive heap abstraction, abstract heap objects allocated at line 9 are distinguished as $\lceil l_{19}, \text{Object}[], l_3\rceil$ and $\lceil l_{19}, \text{Object}[], l_7\rceil$, respectively. The lifted analysis can precisely answer both questions posed in Example 2.

6. Related Work and Discussions

6.1 Points-to Analysis

Points-to analysis has been an active field over the past two decades. At present, it remains a challenging task to design a scalable yet precise analysis under the current popular computing resources. Existing practiced points-to analysis are cloning-based to achieve context-sensitivity, as briefly discussed below.

One of the pioneer work is Andersen's points-to analysis for C programs [2]. It is a subset-based, flow-insensitive analysis that is encoded as constraint solving problems. More specifically, pointer assignments are described by subset constraints, e.g. $x = y$ induces $pta(y) \subseteq pta(x)$. This analysis is the basis of many points-to analysis algorithms afterwards. The scalability of Andersen's analysis has been greatly improved by efficient constraint solvers. Andersen's analysis was leveraged to Java by annotated constraints [14].

The first scalable cloning-based context-sensitive Java points-to analysis is presented in [18], in which programs and analysis problems are encoded as rules in logic program Datalog. To handle recursive procedures, calling contexts are cloned after merging loops as equivalent classes. The BDD (Binary Decision Diagram) based implementation, as well as the approximation on recursions, enable the analysis to scale. As pointed out in [8], there are usually many loops within the call graph, each of which typically consists of a large number of methods. Though this analysis scales very well to large-scale Java applications, a loss of precision is caused by approximating recursions.

Reps, et al. present a general framework for program analysis based on CFL-reachability [12], in which a points-to analysis for C programs is discussed by formulating pointer assignments as productions of context-free grammars. Inspired by this work, Sridharan, et al. formulated Andersen's analysis for Java [17] as balanced-parentheses problems with respect to operations of field read and write. A novel refinement-based analysis is later proposed in [16]. It firstly performs a context-insensitive analysis, and later recovers the precision on-demand by removing imprecise propagation of points-to sets as violating a grammar for balanced parentheses, with respect to both heap access and method calls. The aforementioned refinement, as well as demand-driven manner, are essential to the success of this analysis regarding good performance on precision and scalability (measured by downcast safety analysis).

SPARK[7] is a widely-used test-bed for experimenting with various Java points-to analysis algorithms. It supports both equality and subset-based points-to analysis, and provides various algorithms for call graph construction, such as class hierarchy analysis (CHA), rapid type analysis (RTA), on-the-fly algorithms such as the aforementioned cloning-based context-sensitive points-to analysis for Java [18]. It also provides variations regarding field-sensitivity. The BDD-based implementation of the subset-based algorithms further improves the efficiency.

Another stream of research examines calling contexts in terms of sequences of objects on which methods are invoked, called object-sensitivity [11]. Similar to call-site-strings based approach, the sequence of receiver objects can be unbounded and demands proper approximations. A recent empirical study compares the precision of subset-based points-to analysis with various abstractions on context-sensitivity [8]. It shows that, context-sensitivity is crucial to the analysis precision in practice, and points-to analysis with object-sensitivity excels at precision and is more likely to scale.

For the sake of practical scalability, a context-insensitive abstraction on heap is commonly adopted in most of points-to analysis algorithms. Recently, some efforts are made on designing points-to analysis with context-sensitive heap abstraction, e.g., the analysis with full heap cloning within acyclic call paths in [6], and the analysis with bounded heap cloning by merging equivalent calling contexts in [19].

6.2 Stacking-based Program Analysis

Though WPDSs are well understood as a generalized framework for analyzing programs with procedures on certain properties, many problems remain to be investigated like,

- whether stacking-based analysis can provide the same precision as cloning-based analysis on objected-oriented features, and
- whether stacking-based analysis can scale well to large practical applications. Retaining context-sensitivity within recursive procedure calls is the major bottleneck to scalability.

To tackle with the second problem, we propose and develop a scalable stacking-based points-to analysis for Java [9], with no restriction on recursive procedure calls. The first step to scalability is that, instead of passing global variables explicitly as parameters along procedure calls and returns [12], we model the heap memory as a global data structure, which provides global references with synchronized cached data flows. To further accelerate the analysis, we propose a two-staged iterative procedure, to diet most of local iteration cycles for a substantial speedup.

In this paper, we investigate the precision of the stacking-based analysis and propose CWPDSs. Our solution is inspired by the extension of LTL model checking on pushdown systems from simple

valuation to regular valuation [3]. The authors propose techniques for solving model checking problems with regular valuation with an eye on efficiency. They also present algorithms for checking LTL properties on pushdown systems with check points, for program analysis with stack inspection.

Our reduction algorithm by extending the stack alphabet resembles one of their proposals with the following difference. In our case, configurations that cannot be induced by $\Rightarrow_c$ have to be excluded from the $\mathcal{P}$-automaton construction, otherwise the invalid value would be propagated. To this end, instead of modifying model checking algorithms on WPDSs, a simple check is performed during the translation (Fig. 4). Besides, for the sake of efficiency, we place a condition on stack contents excluding the topmost stack symbol. Such a concern has a slight effect on acceptance conditions of condition automata, such that the initial state can be accepted.

In contrast to on-the-fly points-to analysis, an ahead-of-time points-to analysis is proposed as one run of weighted pushdown model checking [9]. The notion of valid paths are enriched such that invalid control flows that violate Java semantics on dynamic dispatch are detected as those carrying conflicted dataflow. The analysis enjoys context-sensitivities regarding both call graph construction and valid paths.

Tremendous improvements have been proposed on weighted pushdown systems. Extended weighted pushdown systems (EWPDSs) are proposed, to provide a convenient abstraction mechanism for local variables [4]. EWPDSs exclude the call to return edge in the supergraph abstraction. Instead, a merge function is proposed to restore the local variables of the caller when the callee returns. Besides, a graph-theoretic algorithm is used to improve the running time for model checking on (extended) weighted pushdown systems [5].

Alur et al. proposed an interesting class of language, so-called visibly pushdown languages (VPLs), by driving the stack operations (correspondingly, the stack height) with inputs [1]. VPL is in between balanced languages and deterministic context-free languages, yet enjoys good closure properties and decidability results as regular languages. It would be interesting to see the possibility of lifting regular conditions of CWPDS to VPLs.

7. Conclusions and Future Work

We extend weighted pushdown systems to conditional weighted pushdown systems, by further associating each transition rule with a regular language that specifies conditions under which the transition rule can be applied. We show that, the model checking problem on conditional weighted pushdown systems can be reduced to the existing model checking problems on weighted pushdown systems. The increase of time complexity for model checking problems on weighted pushdown systems after extending to conditional weighted pushdown systems, is polynomial (including a linear case) in the states of the product automata that recognizes the union of regular conditions.

There are wider applications of conditional weighted pushdown systems than those that weighted pushdown systems are directly applicable. We lift a stacking-based points-to analysis algorithm for Java in the framework of conditional weighted pushdown systems. In addition to the fundamental context-sensitivity in terms of valid paths, the lifted analysis further enjoys context-sensitivity, regarding heap abstraction, call graph construction, and heap access. These context-sensitive properties are shown to be crucial to the analysis precision.

It would be interesting to evaluate the lifted points-to analysis in practice. A practical tradeoff between precision and practical scalability is expected, by a proper choice of the equivalence relation $\sim$ over calling contexts. For simplicity, the choice of $\sim$ is uniform on the entire program in the current presentation. The choice of uniform partitions provides a way of encoding demand-driven style of analysis regarding precision.

Acknowledgments

This research is supported by STARC (Semiconductor Technology Academic Research Center). The authors would like to thank anonymous reviewers for their valuable comments.

References

[1] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[2] L. Andersen. Program analysis and specialization for the C programming language. *PhD thesis*, 1994.

[3] J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.

[4] A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV'05: Proceedings of the 17th International Conference on Computer Aided Verification*, pages 434–448, 2005.

[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] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *SIGPLAN Not.*, 42(6):278–289, 2007.

[7] O. Lhoták and L. Hendren. Scaling Java points-to analysis using spark. In *CC'03: Proceedings of the 12th International Conference on Compiler Construction*.

[8] 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.

[9] X. Li and M. Ogawa. An ahead-of-time yet context-sensitive points-to analysis for Java. In *Proceedings of BYTECODE'09 (to be available in ENTCS)*, York, Mar. 2009.

[10] X. Li and M. Ogawa. Stacking-based context-sensitive points-to analysis for Java. In *HVC'09: Hardware and Software: Verification and Testing, 5th International Haifa Verification Conference, to be available in LNCS*. Springer, 2009.

[11] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[12] T. Reps. Program analysis via graph reachability. In *ILPS '97: Proceedings of the 1997 international symposium on Logic programming*, pages 5–19, Cambridge, MA, USA, 1997. MIT Press.

[13] 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.

[14] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. *SIGPLAN Not.*, 36(11):43–55, 2001.

[15] S. Schwoon. *Model-Checking Pushdown Systems*. PhD thesis, 2002.

[16] 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.

[17] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. *SIGPLAN Not.*, 40(10):59–76, 2005.

[18] 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.

[19] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 225–236, New York, NY, USA, 2008. ACM.

Algorithm TRANS
Input: A conditional weighted pushdown system $W_c = (P_c, S, f)$,
where $P_c = (Q, \Gamma, \mathcal{C}, \Delta, q_0, \gamma_0)$ with $\mathcal{C} = \{L_1, ..., L_n\}$
Output: A weighted pushdown system $W_t = (P, S, f')$, where $P = (Q, \Gamma', \Delta', q_0, \gamma_0')$

0. let $A = \{A_1, ..., A_n\}$ with $A_i = \phi(L_i^R)$ for $1 \leq i \leq n$, and let $\Pi_{1 \leq i \leq n} A_i = (\widehat{S}, \Gamma, \widehat{\delta}, \dot{\mathbf{s}}, \widehat{F})$
1. $\Gamma' := \Gamma \times \widehat{S}$, and $\gamma_0' := (\gamma_0, \dot{\mathbf{s}})$
2. **for** each $r \in \Delta$
3. **if** $r : \langle p, \gamma \rangle \overset{A_i}{\hookrightarrow} \langle q, \nu \rangle$ with $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$
4. **for** each $\mathbf{s} \in \widehat{S}$
5. **if** $\mathbf{s}_i \in F_i$
6. **then** $r' := \langle p, (\gamma, \mathbf{s}) \rangle \hookrightarrow \langle q, (\nu, \mathbf{s}) \rangle$
7. $\Delta' := \Delta' \cup \{r'\}$ and $f'(r') = f(r)$
8. **if** $r : \langle p, \gamma \rangle \overset{A_i}{\hookrightarrow} \langle q, \epsilon \rangle$ with $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$
9. **for** each $\mathbf{s} \in \widehat{S}$
10. **if** $\mathbf{s}_i \in F_i$
11. **then** $r' := \langle p, (\gamma, \mathbf{s}) \rangle \hookrightarrow \langle q, \epsilon \rangle$
12. $\Delta' := \Delta' \cup \{r'\}$ and $f'(r') = f(r)$
13. **if** $r : \langle p, \gamma \rangle \overset{A_i}{\hookrightarrow} \langle q, \alpha\beta \rangle$ with $A_i = (S_i, \Gamma, \delta_i, \dot{s}_i, F_i)$
14. **for** each $\mathbf{s}, \mathbf{t} \in \widehat{S}$
14. **if** $\mathbf{s}_i \in F_i$ and $\mathbf{t} = \widehat{\delta}(\mathbf{s}, \beta)$
15. **then** $r' := \langle p, (\gamma, \mathbf{s}) \rangle \hookrightarrow \langle q, (\alpha, \mathbf{t})(\beta, \mathbf{s}) \rangle$
16. $\Delta' := \Delta' \cup \{r'\}$ and $f'(r') = f(r)$

Figure 4. An Algorithm Translating CWPDSs to WPDSs

$\mathcal{A}[\![x = \text{new } \mathtt{T}]\!] = \{r : \langle \mathtt{Henv}, C.\psi \rangle \overset{\mathtt{tail}(L)}{\hookrightarrow} \langle x, C.\psi \rangle \mid f(r) = \lambda x.\{(l, C.\psi, \mathtt{T}, \mathtt{tail}(L))\} \text{ for } L \in \text{ACC}(C.\psi)/_\sim\}$

$\mathcal{A}[\![x = y]\!] = \{\langle y, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle x, C.\psi \rangle\}$

$\mathcal{A}[\![x := (\mathtt{T})y]\!] = \{\langle y, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle x, C.\psi \rangle\}$

$\mathcal{A}[\![\text{return } x]\!] = \{\langle x, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \mathtt{ret}, C.\psi \rangle\}$

$\mathcal{A}[\![x := \text{@this} : \mathtt{T}]\!] = \{\langle \mathtt{this}, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle x, C.\psi \rangle\} \cup A_e$

where $A_e = \begin{cases} \{r : \langle \mathtt{Henv}, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \mathtt{this}, C.\psi \rangle \mid f(r) = \lambda x.\{(\_, C.\psi, \mathtt{T}, \mathbb{C})\}\}, & \text{if } C.\psi \in M_0; \\ \emptyset, & \text{otherwise.} \end{cases}$

$\mathcal{A}[\![x := \text{@parameter}_k : \mathtt{T}]\!] = \{\langle \mathtt{arg}_k, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle x, C.\psi \rangle\} \cup A_p$

where $A_p = \begin{cases} \{r : \langle \mathtt{Henv}, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \mathtt{arg}_k, C.\psi \rangle \mid f(r) = \lambda x.\{(\_, C.\psi, \mathtt{T}, \mathbb{C})\}\}, & \text{if } C.\psi \in M_0; \\ \emptyset, & \text{otherwise.} \end{cases}$

$\mathcal{A}[\![x = y[i]]\!] = \{\langle [\![o]\!], C.\psi \rangle \overset{\mathtt{tail}(L)}{\hookrightarrow} \langle x, C.\psi \rangle \mid o \in \mathbf{R}(y, L) \text{ for } L \in \text{ACC}(C.\psi)/_\sim\} \cup A_g$

$\mathcal{A}[\![y[i] = x]\!] = \{\langle x, C.\psi \rangle \overset{\mathtt{tail}(L)}{\hookrightarrow} \langle [\![o]\!], C.\psi \rangle \mid o \in \mathbf{R}(y, L) \text{ for } L \in \text{ACC}(C.\psi)/_\sim\} \cup A_g$

where $A_g = \{r : \langle \mathtt{Henv}, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle [\![o]\!], C.\psi \rangle \mid f(r) = \lambda x.s \text{ for } s = \mathbf{R}([\![o]\!], \mathbb{C}), o \in \mathbf{R}(y, \mathbb{C})\}$

$\mathcal{A}[\![x = y.f]\!] = \{\langle o.f, C.\psi \rangle \overset{\mathtt{tail}(L)}{\hookrightarrow} \langle x, C.\psi \rangle \mid o \in \mathbf{R}(y, L) \text{ for } L \in \text{ACC}(C.\psi)/_\sim\} \cup A_f$

$\mathcal{A}[\![y.f = x]\!] = \{\langle x, C.\psi \rangle \overset{\mathtt{tail}(L)}{\hookrightarrow} \langle o.f, C.\psi \rangle \mid o \in \mathbf{R}(y, L) \text{ for } L \in \text{ACC}(C.\psi)/_\sim\} \cup A_f$

where $A_f = \{r : \langle \mathtt{Henv}, C.\psi \rangle \overset{\mathbb{C}}{\hookrightarrow} \langle o.f, C.\psi \rangle \mid f(r) = \lambda x.s \text{ for } s = \mathbf{R}(o.f, \mathbb{C}), o \in \mathbf{R}(y, \mathbb{C})\}$

Figure 5. $\mathcal{A}[\![\_]\!] : \mathtt{Stmt} \to \mathcal{P}(\hookrightarrow)$

main:

$\langle \texttt{Henv}, \texttt{sp}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{Henv}, \text{main}\rangle$ $\quad$ id

$\langle \texttt{Henv}, \text{main}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle a, \text{main}\rangle$ $\quad \lambda x.\{\lceil l_3, \texttt{A}, \varepsilon\rceil\}$

$\langle a, \text{main}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle x, \text{foo}\ l_4\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{foo}}, l_4\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle c, \text{main}\rangle$ $\quad$ id

$\langle a, \text{main}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle t, \text{foo}\ l_5\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{get}}, l_5\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle i, \text{main}\rangle$ $\quad$ id

$\langle \texttt{Henv}, \text{main}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle b, \text{main}\rangle$ $\quad \lambda x.\{\lceil l_7, \texttt{B}, \varepsilon\rceil\}$

$\langle b, \text{main}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle x, \text{foo}\ l_8\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{foo}}, l_8\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle d, \text{main}\rangle$ $\quad$ id

$\langle b, \text{main}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle t, \text{foo}\ l_9\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{get}}, l_9\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle s, \text{main}\rangle$ $\quad$ id

A:

$\langle \texttt{Henv}, \text{main}\rangle \overset{\{l_3\}}{\hookrightarrow} \langle \lceil l_3, \texttt{A}, \varepsilon\rceil.\text{arr}, \text{A}\rangle$ $\quad \lambda x.\{\lceil l_{19}, \text{Object}[], l_3\rceil\}$

$\langle \texttt{Henv}, \text{main}\rangle \overset{\{l_7\}}{\hookrightarrow} \langle \lceil l_7, \texttt{A}, \varepsilon\rceil.\text{arr}, \text{A}\rangle$ $\quad \lambda x.\{\lceil l_{19}, \text{Object}[], l_7\rceil\}$

foo:

$\langle x, \text{foo}\rangle \overset{\{l_4\}}{\hookrightarrow} \langle \texttt{this}^{\text{A.set}}, \text{A.set}\ l_{13}\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{A.set}}, l_{13}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{ret}^{\text{foo}}, \text{foo}\rangle$ $\quad$ id

$\langle x, \text{foo}\rangle \overset{\{l_8\}}{\hookrightarrow} \langle \texttt{this}^{\text{B.set}}, \text{B.set}\ l_{13}\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{B.set}}, l_{13}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{ret}^{\text{foo}}, \text{foo}\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{foo}}, \text{foo}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{ret}^{\text{A.set}}, \epsilon\rangle$ $\quad$ id

A.set:

$\langle \texttt{Henv}, \text{A.set}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \lceil l_3, \texttt{A}, \varepsilon\rceil.\text{arr}, \text{A.set}\rangle$ $\quad \lambda x.\{\lceil l_{19}, \text{Object}[], l_3\rceil\}$

$\langle \texttt{Henv}, \text{A.set}\rangle \overset{\{l_{13}l_4\}}{\hookrightarrow} \langle \lceil l_3, \texttt{A}, \varepsilon\rceil.f, \text{A.set}\rangle$ $\quad \lambda x.\{\lceil l_{21}, \texttt{Integer}, l_{13}l_4\rceil\}$

$\langle \lceil l_3, \texttt{A}, \varepsilon\rceil.f, \text{A.set}\rangle \overset{\{l_{13}l_4\}}{\hookrightarrow} \langle [\![\lceil l_{19}, \text{Object}[], l_3\rceil]\!], \text{A.set}\rangle$ $\quad$ id

$\langle \lceil l_3, \texttt{A}, \varepsilon\rceil.f, \text{A.set}\rangle \overset{\{l_{13}l_4\}}{\hookrightarrow} \langle \texttt{ret}^{\text{A.set}}, \text{A.set}\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{A.set}}, \text{A.set}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{ret}^{\text{A.set}}, \epsilon\rangle$ $\quad$ id

A.get:

$\langle \texttt{Henv}, \text{A.get}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \lceil l_3, \texttt{A}, \varepsilon\rceil.\text{arr}, \text{A.get}\rangle$ $\quad \lambda x.\{\lceil l_{19}, \text{Object}[], l_3\rceil\}$

$\langle \texttt{Henv}, \text{A.get}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \lceil l_7, \texttt{A}, \varepsilon\rceil.\text{arr}, \text{A.get}\rangle$ $\quad \lambda x.\{\lceil l_{19}, \text{Object}[], l_7\rceil\}$

$\langle \texttt{Henv}, \text{A.get}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle [\![\lceil l_{19}, \text{Object}[], l_3\rceil]\!], \text{A.get}\rangle$ $\quad \lambda x.\{\lceil l_{21}, \texttt{Integer}, l_{13}l_4\rceil\}$

$\langle \texttt{Henv}, \text{A.get}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle [\![\lceil l_{19}, \text{Object}[], l_7\rceil]\!], \text{A.get}\rangle$ $\quad \lambda x.\{\lceil l_{31}, \texttt{String}, l_{13}l_8\rceil\}$

$\langle [\![\lceil l_{19}, \text{Object}[], l_3\rceil]\!], \text{A.get}\rangle \overset{\{l_5\}}{\hookrightarrow} \langle \texttt{ret}^{\text{A.get}}, \text{A.get}\rangle$ $\quad$ id

$\langle [\![\lceil l_{19}, \text{Object}[], l_7\rceil]\!], \text{A.get}\rangle \overset{\{l_9\}}{\hookrightarrow} \langle \texttt{ret}^{\text{A.get}}, \text{A.get}\rangle$ $\quad$ id

$\langle \texttt{ret}^{\text{A.get}}, \text{A.get}\rangle \overset{\mathbb{C}}{\hookrightarrow} \langle \texttt{ret}^{\text{A.get}}, \epsilon\rangle$ $\quad$ id

Figure 7. Core Parts of the Encoding of the Program in Fig. 2 as Conditional Weighted Pushdown Systems