Pruning, Pushdown Exception-Flow Analysis

Shuying Liang
University of Utah
liangsy@cs.utah.edu

Weibin Sun
University of Utah
wbsun@cs.utah.edu

Matthew Might
University of Utah
might@cs.utah.edu

Andy Keep
University of Utah
andy.keep@gmail.com

David Van Horn
University of Maryland
dvanhorn@cs.umd.edu

Abstract—Statically reasoning in the presence of exceptions and about the effects of exceptions is challenging: exception-flows are mutually determined by traditional control-flow and points-to analyses. We tackle the challenge of analyzing exception-flows from two angles. First, from the angle of pruning control-flows (both normal and exceptional), we derive a pushdown framework for an object-oriented language with full-featured exceptions. Unlike traditional analyses, it allows precise matching of throwers to catchers. Second, from the angle of pruning points-to information, we generalize *abstract* garbage collection to object-oriented programs and enhance it with liveness analysis. We then seamlessly weave the techniques into enhanced reachability computation, yielding highly precise exception-flow analysis, without becoming intractable, even for large applications. We evaluate our pruned, pushdown exception-flow analysis, comparing it with an established analysis on large scale standard Java benchmarks. The results show that our analysis *significantly* improves analysis precision over traditional analysis within a reasonable analysis time.

I. INTRODUCTION

Exceptions are not exceptional enough. They pervade the control-flow structure of modern object-oriented programs. An exception indicates an error occurred during program execution. Exceptions are resolved by locating code specified by the programmer for handling the exception (an exception handler) and executing this code.

This language feature is designed to ensure software robustness and reliability. Ironically, Android malware is exploiting it to leak private sensitive information to the Internet through exception handlers [22]. Analyzing the behavior of programs in the presence of exceptions is important to detect such vulnerabilities. However, exception-flow analysis is challenging, because it depends upon control-flow analysis and points-to analysis, which are themselves mutually dependent, as illustrated in Figure 1.

In Figure 1, edge A denotes the mutual dependence between exception-flow analysis and traditional control-flow analysis (CFA). CFA traditionally analyzes which methods can be invoked at each call-site. Exception-flow analysis refers to the control-flow that is introduced when throwing exceptions [6]. Intuitively, throwing an exception behaves like a global `goto` statement, in that it introduces additional, complex, inter-procedural control flow into the program. This makes it difficult to reason about feasible run-time paths using traditional CFA. Similarly, infeasible call and return flows can cause spurious paths between throw statements and catch blocks. The following simple example demonstrates this:

```
try {
   maybeThrow();  // Call 1
} catch (Exception e) {  // Handler 1
   System.err.println("Got an exception");
}
maybeThrow();  // Call 2
```

Under a monovariant abstraction like 0-CFA [29], where the distinction between different invocations of the same procedure are lost, it will seem as though exceptions thrown from `Call 2` can be caught by `Handler 1`.
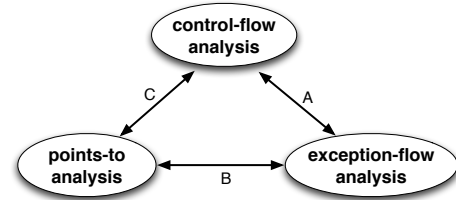

Fig. 1: Relationship among exception-flow analysis, control-flow analysis and points-to analysis.

Edge B in Figure 1 denotes the relationship between exception-flow analysis and points-to analysis. Points-to analysis computes which abstract objects (with respect to allocation sites, calling contexts, etc.) a program variable or register can point to. Points-to analysis affects exception-flow analysis, because the type of the exception at a throw site determines which catch block will be executed. That is to say, exception-flow analysis requires precise points-to analysis. Similarly, exceptional flows affect points-to analysis, since the path taken by the exceptional flow can enable or disable object assignments and bindings.

The mutually recursive relationship of CFA and points-to analysis, denoted by edge C, is obvious: abstract objects (points-to analysis) determine which methods can be resolved in dynamic dispatch (CFA), while control-flow paths affect object assignments and bindings for points-to analysis. In fact, exception-flow analysis is an example of this relationship, which exacerbates the edge C relationship further!

A. Existing approaches

Existing compilers or analysis frameworks provide a conservative model for exception handling. One approach assigns all exceptions thrown in a program to a single global variable. This variable is then read at an exception catch site. This approach is imprecise since it has no knowledge of which exception propagates to a catch site [13], [20].

The second approach analyzes exceptional control flow only intra-procedurally, computing only local catch clauses for a try block, with no dynamic propagation of exceptions inter-procedurally.

The third approach is co-analysis using both control-flow analysis and points-to analysis (a.k.a. on-the-fly control-flow construction) to handle exceptions, which yields reasonable precision, compared to the aforementioned two approaches, as documented in a past precision study [6]. Unfortunately, even for the best co-analysis, where boosting context-sensitivity improves the analysis of exceptions, it does not improve as much as it does for points-to analysis. It is too easy for exceptions to cross context boundaries and merge. For the previous simple example, we could increase to 1-call-site sensitivity. However, context-sensitivity costs more and is easily confused when calls are wrapped, as in:

```
try {
   callsMaybeThrow();  // Call 1
```

```
} catch (Exception e) {   // Handler 1
    System.err.println("Got an exception");
}
callsMaybeThrow();  // Call 2
// ...
void callsMaybeThrow() {
  maybeThrow();
}
```

Similarly, values can easily merge with finitized object-sensitivity in points-to analysis. For example, if object-sensitivity uses k levels of object allocation sites (or a mix with receiver objects) to distinguish contexts, objects are merged when the level exceeds k. Even worse, the limited k-sensitivity does not distinguish live heap objects from dead (garbage) heap objects, the existence of which harms both the precision and performance of the analysis. More detailed related work is described in Section IX.

B. Our approach

Due to the intrinsic relationships illustrated in Figure 1, we propose a hybrid joint analysis of pushdown exception-flow analysis with abstract garbage collection enhanced with liveness analysis. Specifically, a pushdown system derived from the concrete semantics of a core calculus for an object-oriented language extended with exceptions is used to tackle exceptional control-flow matching between catches and throws, in addition to call and return matches. Abstract garbage collection is adapted to an object-oriented program setting, and it is enhanced with liveness analysis to tackle the points-to aspect of exception-flow analysis. We evaluate an implementation for Dalvik bytecode of the joint analysis technique on a standard set of Java benchmarks. The results show that the pruned, pushdown exception-flow analysis yields higher precision than traditional exception-flow analysis by up to 11 times within a reasonable amount of analysis time.

C. Organization

The rest of the paper is organized as follows: Section II presents the core calculus of an object-oriented language extended with exceptions. Section III formulates the concrete semantics for the language with the intent of refactoring and abstracting it into a static analyzer. Section IV derives the abstract semantics from the concrete semantics by reformulating the structure of continuations into a list of frames and forms an implicit pushdown system. Section V-A introduces the adaptation of abstract garbage collection in object-oriented languages. Section V-B enhances the adapted abstract garbage collection with liveness analysis for better precision. The reachability algorithm is described in Section VI. Section VII describes the details of our implementation. The evaluation and benchmarks are reported in Section VIII. Section IX reports related work, and Section X concludes.

II. A FEATHERWEIGHT JAVA WITH EXCEPTIONS

For presentation purpose, we start with a variant of Featherweight Java [14] in "A-Normal" form [11] with exceptions. A-Normal Featherweight Java (ANFJ) is identical to ordinary Featherweight Java, except that arguments to a function call must be atomically evaluable, as they are in A-Normal Form λ-calculus. For example, the body `return f.foo(b.bar());` becomes the sequence of statements

```
B b1 = b.bar();
F f1 = f.foo(b1);
return f1;
```

$$\varsigma \in \Sigma = \mathsf{Stmt} \times FramePointer \times Store \times Kont \times Time$$
$$\sigma \in Store = Addr \rightharpoonup D$$
$$d \in D = Val$$
$$val \in Val = Obj$$
$$o \in Obj = \mathsf{ClassName} \times ObjectPointer$$
$$\kappa \in Kont ::= \mathbf{fun}(v, s, fp, \kappa)$$
$$\qquad | \quad \mathbf{handle}(C, v, \vec{s}, fp, \kappa)$$
$$\qquad | \quad \mathbf{halt}$$
$$fp \in FramePointer \text{ is a set of frame pointers}$$
$$op \in ObjectPointer \text{ is a set of object pointers}$$
$$ptr \in Ptr = FramePointer + ObjectPointer$$
$$a \in Addr = (\mathsf{Var} + \mathsf{Method}) \times Ptr$$
$$t \in Time \text{ is a set of time-stamps.}$$

Fig. 2: Concrete state-space for A-Normal Featherweight Java.

$$\mathcal{C} : \mathsf{ClassName} \rightarrow (\mathsf{FieldName}^* \times Ructor)$$

$$\mathcal{K} \in Ructor = \overbrace{Addr^*}^{\text{fields}} \times \overbrace{D^*}^{\text{arguments}} \rightarrow (\ \overbrace{Store}^{\text{field values}} \times \overbrace{ObjectPointer}^{\text{record}}\ )$$
$$\mathcal{M} : D \times \mathsf{MethodCall} \rightharpoonup \mathsf{Method}$$

Fig. 3: Helper functions for the concrete semantics.

This does not change the expressive power of the language or the nature of the analysis to come, but it does simplify the semantics while preserving the essence of the language.

The following grammar describes A-Normal Featherweight Java extended with exceptions; like regular Java, ANFJ has statement forms:

$$\mathsf{Class} ::= \mathtt{class}\ C\ \mathtt{extends}\ C'\ \{\overrightarrow{C'' f}\ ;\ K\ \overrightarrow{M}\}$$
$$K \in \mathsf{Konst} ::= C\ (\overrightarrow{C f})\ \{\mathtt{super}(\overrightarrow{f'})\ ;\ \overrightarrow{\mathtt{this}.f'' = f'''\ ;}\}$$
$$M \in \mathsf{Method} ::= C\ m\ (\overrightarrow{C v})\ \{\ \overrightarrow{C v}\ ;\ \vec{s}\ \}$$
$$s \in \mathsf{Stmt} ::= v = e\ ;^{\ell}$$
$$\qquad | \quad \mathtt{return}\ v\ ;^{\ell}$$
$$\qquad | \quad \mathtt{try}\ \{\vec{s}\}\ \mathtt{catch}\ (C\ v)\ \{\vec{s}'\}^{\ell}$$
$$\qquad | \quad \mathtt{throw}\ v\ ;^{\ell}$$
$$e \in \mathsf{Exp} ::= v \mid v.f \mid v.m(\overrightarrow{v}) \mid \mathtt{new}\ C\ (\overrightarrow{v}) \mid (C)\,v$$

$$f \in \mathsf{FieldName} = \mathsf{Var}$$
$$C \in \mathsf{ClassName} \text{ is a set of class names}$$
$$m \in \mathsf{MethodCall} \text{ is a set of method invocation sites}$$
$$\ell \in \mathsf{Lab} \text{ is a set of labels}$$
$$v \in \mathsf{Var} \text{ is a set of variables}$$

The set Var contains both variable and field names. Every statement has a label. The function $succ : \mathsf{Lab} \rightharpoonup \mathsf{Stmt}$ yields the (semantically) subsequent statement for a statement's label.

III. MACHINE SEMANTICS FOR FEATHERWEIGHT JAVA

In preparation for synthesizing an abstract interpreter, we first construct a small-step abstract machine-based semantics for Featherweight Java. Figure 2 contains the concrete state-space for the small-step Featherweight Java machine. Each machine state has five components: a statement, a frame pointer, a store, a continuation and a timestamp. The encoding of objects abstracts over a low-level implementation: an object is a class plus a base pointer, and field addresses are "offsets" from this base pointer. Given an object (C, op), the address of field f would be (f, op). In the semantics, object allocation creates a single new base object pointer op'.

The concrete semantics use the helper functions described in Figure 3. The constructor-lookup function $\mathcal{C}$ yields the field names and the constructor associated with a class name. A constructor $\mathcal{K}$ takes a newly allocated address to use for fields and a vector of arguments; it returns the change to the store plus the record component of the object that results from running the constructor. The method-lookup function $\mathcal{M}$ takes a method invocation point and an object to determine which method is actually being called at that point. The concrete semantics are encoded as a small-step transition relation, $(\Rightarrow) \subseteq \Sigma \times \Sigma$. Each statement and expression type has a transition rule below.

Variable reference: Variable reference computes the address relative to the current frame pointer and retrieves the result from the store:

$$([\![v = v' \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow (succ(\ell), fp, \sigma', \kappa, t'),$$
$$\text{where } t' = tick(\ell, t), \ \sigma' = \sigma[(v, fp) \mapsto \sigma(v', fp)].$$

Return to call: Returning from a function checks if the top-most frame pointer is a function continuation (as apposed to an exception-handler continuation). If it is, then the machine binds the result and restores the context of the continuation; if not, then the machine skips to the next continuation. If $\kappa = \mathbf{fun}(v', s, fp', \kappa')$:

$$([\![\texttt{return } v \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow (s, fp', \sigma', \kappa', t'), \text{ where}$$
$$t' = tick(\ell, t), \ \sigma' = \sigma[(v', fp) \mapsto d], \ d = \sigma(v, fp).$$

Return over handler: If the topmost continuation is a handler, then the machine pops the handler off the stack. So, if $\kappa = \mathbf{handle}(C, v, \vec{s}, fp', \kappa')$:

$$([\![\texttt{return } v \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow ([\![\texttt{return } v \,;\, ^\ell]\!], fp, \sigma, \kappa', t')$$
$$\text{where } t' = tick(\ell, t).$$

Field reference: Field reference is similar to variable reference, except that it must find the base object pointer with which to compute the appropriate offset:

$$([\![v = v' . f \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow (succ(\ell), fp, \sigma', \kappa, t'),$$
$$\text{where } t' = tick(\ell, t),$$
$$(C, op') = \sigma(v', fp), \ \sigma' = \sigma[(v, fp) \mapsto \sigma(f, op')].$$

Method invocation: Method invocation is a multi-step process: it looks up the object, determines the class of the object and then identifies the appropriate method. When transitioning to the body of the resolved method, a new function continuation is instantiated, which records the caller's execution context. Finally, the store is updated with the bindings of formal parameters to evaluated values of passed arguments.

$$([\![v = v_0 . m \, (\overrightarrow{v'}) \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow (s_0, fp', \sigma', \kappa', t'),$$
$$\text{where } M = [\![C \ m \ (\overrightarrow{C \ v''}) \ \{\overrightarrow{C' \ v'''} \,;\, \vec{s}\}]\!] = \mathcal{M}(d_0, m)$$
$$d_0 = \sigma(v_0, fp), d_i = \sigma(v'_i, fp), t' = tick(\ell, t), fp' = alloc(\ell, t'),$$
$$\kappa' = \mathbf{fun}(v, succ(\ell), fp, \kappa), a'_i = (v''_i, fp'), \sigma' = \sigma[a'_i \mapsto d_i].$$

Object allocation: Object allocation creates a new base object pointer; it also invokes the constructor helper to initialize the object(The $(+)$ operation represents right-biased functional union in that wherever vector $\vec{x}$ is in scope, its components are implicitly in scope: $\vec{x} = \langle x_0, \ldots, x_{length(\vec{x})} \rangle$):

$$([\![v = \texttt{new } C \ (\overrightarrow{v'}) \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow (succ(\ell), fp, \sigma', \kappa, t'),$$
$$\text{where } t' = tick(\ell, t), \ d_i = \sigma(v'_i, fp), \ (\vec{f}, \mathcal{K}) = \mathcal{C}(C),$$
$$fp' = alloc(\ell, t'), \ a_i = (f_i, fp')(\Delta\sigma, op') = \mathcal{K}(\vec{a}, \vec{d}),$$
$$d' = (C, op'), \ \sigma' = \sigma + \Delta\sigma + [(v, fp) \mapsto d'].$$

Casting: A cast references a variable, replacing the class of the object:

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Stmt} \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \hat{D}$$
$$\hat{d} \in \hat{D} = \mathcal{P}\left(\widehat{Val}\right)$$
$$\widehat{val} \in \widehat{Val} = \widehat{Obj}$$
$$\hat{o} \in \widehat{Obj} = \mathsf{ClassName} \times \widehat{ObjectPointer}$$
$$\hat{\kappa} \in \widehat{Kont} = \widehat{Frame}^*$$
$$\hat{\phi} \in \widehat{Frame} = \widehat{CallFrame} + \widehat{HandlerFrame}$$
$$\hat{\chi} \in \widehat{CallFrame} ::= \mathbf{fun}(v, s, \hat{fp})$$
$$\hat{\eta} \in \widehat{HandlerFrame} ::= \mathbf{handle}(C, v, \vec{s}, \hat{fp})$$
$$\hat{fp} \in \widehat{FramePointer} \text{ is a set of frame pointers}$$
$$\widehat{op} \in \widehat{ObjectPointer} \text{ is a set of object pointers}$$
$$\widehat{ptr} \in \widehat{Ptr} = \widehat{FramePointer} + \widehat{ObjectPointer}$$
$$\hat{a} \in \widehat{Addr} = (\mathsf{Var} + \mathsf{Method}) \times \widehat{Ptr}$$
$$\hat{t} \in \widehat{Time} \text{ is a set of time-stamps.}$$

Fig. 4: Abstract state-space for pushdown analysis of A-Normal Featherweight Java.

$$([\![v = (C') \ v']\!], fp, \sigma, \kappa, t) \Rightarrow (succ(\ell), fp, \sigma', \kappa, t'),$$
$$\text{where } t' = tick(\ell, t), \ \sigma' = \sigma[(v, fp) \mapsto \sigma(v', fp)].$$

Try: A $\texttt{try}$ statement creates a new handler continuation and then proceeds to the body of the $\texttt{try}$ statement.

$$([\![\texttt{try } \{\vec{s}\} \texttt{ catch } (C \ v) \ \{\vec{s'}\}^\ell]\!], fp, \sigma, \kappa, t)$$
$$\Rightarrow (succ(\ell), fp, \sigma, \kappa', t')$$
$$\text{where } t' = tick(\ell, t), \ \kappa' = \mathbf{handle}(C, v', s'_1, fp, \kappa).$$

Throw to matching handler: When the machine encounters a throw statement, it must check if the topmost continuation is both a handler and a matching handler; if so, then it returns to the context within the continuation: If $\kappa = \mathbf{handle}(C', v', s, fp'', \kappa')$ and $\sigma(v, fp) = (C, op')$ and C is a C':

$$([\![\texttt{throw } v \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow (s, fp'', \sigma', \kappa', t')$$
$$\text{where } t' = tick(\ell, t), \ \sigma' = \sigma[(v', fp'') \mapsto (C, op')].$$

Throw past non-matching handler: When throwing, if the topmost handler is not a match, the machine looks deeper in the stack for a matching handler. If $\kappa = \mathbf{handle}(C', v', s, fp'', \kappa')$ and $\sigma(v, fp) = (C, op')$ but C is not a C':

$$([\![\texttt{throw } v \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow ([\![\texttt{throw } v \,;\, ^\ell]\!], fp, \sigma, \kappa', t')$$
$$\text{where } t' = tick(\ell, t).$$

Throw past return point: If throwing an exception and the topmost handler is a function return point, then it jumps over this continuation. If $\kappa = \mathbf{fun}(v', s, fp', \kappa')$:

$$([\![\texttt{throw } v \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow ([\![\texttt{throw } v \,;\, ^\ell]\!], fp, \sigma, \kappa', t')$$
$$\text{where } t' = tick(\ell, t).$$

Popping handlers: When control passes out of a $\texttt{try}$ block, the topmost handler must be popped from the stack. To handle this, the "successor" of the last statement in a $\texttt{try}$ block is actually a special $\texttt{pophandler}$ statement, and the "successor" of that statement is the statement directly following the $\texttt{try}$ block.

$$([\![\texttt{pophandler} \,;\, ^\ell]\!], fp, \sigma, \kappa, t) \Rightarrow (succ(\ell), fp, \sigma, \kappa', t')$$
$$\text{where } t' = tick(\ell, t), \ \kappa = \mathbf{handle}(\ldots, \kappa').$$

IV. A PUSHDOWN SEMANTICS OF EXCEPTIONS

With the concrete semantics for A-Normal Featherweight Java with exceptions in place, we are ready to derive the abstract semantics for

static analysis. "Abstracting abstract machines" (AAM) has proposed a systematic approach to derive such kind of abstraction, which is equivalent to most of the conservative static analyses [32]. The idea is to make the analysis finite and terminate by finitize every component in the state, so that there is no source of infinity. However, when we apply this technique, the precision is not satisfiable in the client security analysis [21], because the over-approximation of the continuation component causes spurious control-flow and return-flows.

Therefore, in this work, we choose to abstract *less* than what AAM approach does: we leave the stack (represented as continuation) unbounded in height. In fact, the central idea behind this abstraction is the generalization of two kinds of frames on stack: the function frames and the exception-handler frames. In this way, we form the abstract pushdown semantics. Then, the pushdown abstract semantics will further be computed as control-state reachability in pushdown systems, which is evolved from the work of [26], [9], [10]. However, unlike them, we improve the algorithm to handle new behaviors introduced by exceptions. The algorithm is detailed in Section VI. The rest of the section focuses how we formulate the pushdown semantics.

Abstract semantics are defined on an abstract state-space. To formulate the pushdown abstract state-space, we first reformulate continuations as a list of frames in the concrete semantics:

$$Kont \cong Frame^*$$
$$Frame = CallFrame + HandlerFrame$$
$$CallFrame ::= \mathbf{fun}(v, s, fp)$$
$$HandlerFrame ::= \mathbf{handle}(C, v, \vec{s}, fp).$$

We have two kinds of frames: function frames as well as handler frames. As with continuations, they may grow without bound (The enhanced reachability algorithm handles this in Section VI).

Figure 4 contains the abstract state-space for the pushdown version of the small-step Featherweight Java machine. At this point, we can extract the high-level structure of the pushdown system from the state-space. A configuration in a pushdown system is a control state (from a finite set) paired with a stack (with a finite number of frames that are defined in Figure 4). This can be observed as follows:

$$\hat{\Sigma} = \mathsf{Stmt} \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Kont} \times \widehat{Time}$$
$$\cong \mathsf{Stmt} \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Time} \times \widehat{Kont}$$
$$= \left(\mathsf{Stmt} \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Time}\right) \times \widehat{Kont}$$
$$= \underbrace{\left(\mathsf{Stmt} \times \widehat{FramePointer} \times \widehat{Store} \times \widehat{Time}\right)}_{\text{control states}} \times \underbrace{\widehat{Frame}^*}_{\text{stack}}$$

Now let us show the detailed abstract transition relations. Thanks to the way we do the abstraction so far (That is, structural abstraction of concrete states except for the stack component), the abstract transition relations resemble a lot as their concrete counterparts. The biggest difference in abstract semantics is that it does weak updates using the operator $\sqcup$. For example, for variable reference (weak updates are underlined.):

$$(\llbracket v = v' ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (succ(\ell), \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}'),$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}) \text{ and } \underline{\hat{\sigma}' = \hat{\sigma} \sqcup [(v, \hat{fp}) \mapsto \sigma(v', \hat{fp})]}$$

The other difference is, whenever evaluating expressions, the results are abstract entities that represents one or more concrete entities. For example, field reference:

$$(\llbracket v = v'. f ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (succ(\ell), \hat{fp}, \hat{\sigma}', \hat{\kappa}, \hat{t}')$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \underline{(C, \widehat{op'}) \in \hat{\sigma}(v', \hat{fp})}, \hat{\sigma}' = \hat{\sigma} \sqcup [(v, \hat{fp}) \mapsto \hat{\sigma}(f, \widehat{op'})]$$

The underlined operation shows that there could be more than one abstract objects are evaluated. The two differences apply to all the

[Try]:
$$(\llbracket \mathtt{try}\ \{\vec{s}\}\ \mathtt{catch}\ (C\ v)\ \{\vec{s}'\}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t})$$
$$\rightsquigarrow (succ(\ell), \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}'),$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}) \text{ and } \hat{\kappa}' = \mathbf{handle}(C, v', s_1', \hat{fp}) :: \hat{\kappa}.$$

[Throw to matching handler]:
$$(\llbracket \mathtt{throw}\ v\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (s, \hat{fp}'', \hat{\sigma}', \hat{\kappa}', \hat{t}'),$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\sigma}' = \hat{\sigma} \sqcup [(v', \hat{fp}'') \mapsto (C, \widehat{op'})],$$
$$\hat{\kappa} = \mathbf{handle}(C', v', s, \hat{fp}'') :: \hat{\kappa}', (C, \widehat{op'}) \in \hat{\sigma}(v, \hat{fp}), C \text{ is a } C'.$$

[Throw past non-matching handler]:
$$(\llbracket \mathtt{throw}\ v\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (\llbracket \mathtt{throw}\ v\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}'),$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \hat{\kappa} = \mathbf{handle}(C', v', s, \hat{fp}'') :: \hat{\kappa}',$$
$$(C, \widehat{op'}) \in \hat{\sigma}(v, \hat{fp}) \text{ and } C \text{ is not a } C'.$$

[Throw past return point]:
$$(\llbracket \mathtt{throw}\ v\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (\llbracket \mathtt{throw}\ v\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}'),$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}), \text{ and } \hat{\kappa} = \mathsf{fun}(v', s, \hat{fp}') :: \hat{\kappa}'.$$

[Return over handler]:
$$(\llbracket \mathtt{return}\ v\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (\llbracket \mathtt{return}\ v\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}'),$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}) \text{ and } \hat{\kappa} = \mathsf{handle}(C, v, \vec{s}, \hat{fp}') :: \hat{\kappa}'.$$

[Popping handlers]:
$$(\llbracket \mathtt{pophandler}\ ; {}^\ell \rrbracket, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) \rightsquigarrow (succ(\ell), \hat{fp}, \hat{\sigma}, \hat{\kappa}', \hat{t}'),$$
$$\text{where } \hat{t}' = \widehat{tick}(\ell, \hat{t}) \text{ and } \hat{\kappa} = \mathbf{handle}(\ldots) :: \hat{\kappa}'.$$

Fig. 5: Abstract transition relations (exception)

other rules. To save space, we demonstrate the abstract rules that involve exceptions. Fig 5 shows how we handle the exception-flow and its mix with normal control-flow. The idea is the "multi-pop" behavior introduced when a function call returns or an exception throws (as the concrete semantics). The effect of this approach substantially simplifies the control-reachability algorithm during summarization, as we shall show in Section VI.

V. ENHANCED ABSTRACT GARBAGE COLLECTION

The previous section formulates a pushdown system to handle complicated control-flows (both normal and exceptional). This section describes how we prune the analysis for exceptions from the angle of points-to analysis with enhanced garbage collection generalized for object-oriented programs.

A. Abstract garbage collection in an object-oriented setting

The idea of abstract garbage collection was first proposed in the work of Might and Shivers [24] for higher-order programs. As an analog to the concrete garbage collection, abstract garbage collection reallocates unreachable abstract resources. Order-of-magnitude improvements in precision have been reported, even as it drops run-times by cutting away false positives. It is natural to think that this technique can benefit exception-flow analysis for object-oriented languages. In fact, in an object-oriented setting, abstract garbage collection can free the analysis from the context-sensitivity and object-sensitivity limitation, since the "garbage" discarded is ignorant of any form of sensitivity! For example, in the following simple code snippet,

```
A a1 = idA(new A());
A a2 = idA(new A()):
B b1 = idB(a1.makeB());
B b2 = idB(a2.makeB());
```

`idA` and `idB` are identity functions. Traditionally, with one level of object-sensitivity and one level of context sensitivity, we are able to distinguish the arguments passed in all of the four lines. However, it

is easy to exceed the k-sensitivity (call site, allocation sites, receiver objects, *etc.*) in modern software constructs. Abstract garbage collection can play a role in the way that it discards conservative values and enables fresh bindings for reused variables (formal parameters). This does not need knowledge about any sensitivity! Thus, it can avoid "merging" of abstract object values (and so indirectly eliminate potentially spurious function calls). For exceptions specifically, abstract garbage collection can help avoid conflating exception objects at various throw sites.

To gain the promised analysis precision and performance, we must conduct a careful and subtle redesign of the abstract garbage collection machinery for object-oriented languages. Specifically, we need to make it work with the abstract semantics defined in Section IV. In addition, the reachability algorithm should also be able to work with abstract garbage collection. Fortunately, the challenge of how to adapt abstract garbage collection into pushdown systems has been resolved in the work of Earl *et al.* [10]. Here we focus on the enhanced machinery for object-oriented languages.

First, we describe how we adapt abstract garbage collection to analyze object-oriented languages. Abstract garbage collection discards unreachable elements from the store, it modifies the transition relation to conduct a "stop-and-copy" garbage collection before each transition. To do so, we define a garbage collection function $\hat{G} : \hat{\Sigma} \to \hat{\Sigma}$ on configurations:

$$\hat{G}(\overbrace{\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}}^{\hat{\varsigma}}) = (\vec{s}, \hat{fp}, \hat{\sigma}|Reachable(\hat{\varsigma}), \hat{\kappa}),$$

where the pipe operation $f|S$ yields the function f, but with inputs not in the set S mapped to bottom—the empty set. The reachability function $Reachable : \hat{\Sigma} \to \mathcal{P}(\widehat{Addr})$ first computes the root set and then the transitive closure of an address-to-address adjacency relation:

$$Reachable(\overbrace{\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}}^{\hat{\varsigma}}) = \left\{ \hat{a} : \hat{a}_0 \in Root(\hat{\varsigma}) \text{ and } \hat{a}_0 \xrightarrow[\hat{\sigma}]{*} \hat{a} \right\},$$

where the function $Root : \hat{\Sigma} \to \mathcal{P}(\widehat{Addr})$ finds the root addresses:

$$Root(\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}, \hat{t}) = \{(v, \hat{fp}) : (v, \hat{fp}) \in dom(\hat{\sigma})\} \cup StackRoot(\hat{\kappa})$$

The $StackRoot : \widehat{Kont} \to \mathcal{P}(\widehat{Addr})$ function finds roots on the stack. However, only $\widehat{CallFrame}$ has the component to construct addresses, so we define a helper function $\hat{\mathcal{F}} : \widehat{Kont} \to \widehat{CallFrame}^*$ to extract only $\widehat{CallFrame}$ out from the stack and skip over all the handle frames. Now $StackRoot$ is defined as

$$StackRoot(\hat{\kappa}) = \{(v, \hat{fp}_i) : (v, \hat{fp}_i) \in dom(\hat{\sigma}) \text{ and } \hat{fp}_i \in \hat{\mathcal{F}}(\hat{\kappa})\},$$

and the relation:$(\to) \subseteq \widehat{Addr} \times \widehat{Store} \times \widehat{Addr}$ connects adjacent addresses: $\hat{a} \to_{\hat{\sigma}} \hat{a}'$ iff there exists $(C, \widehat{op}) \in \hat{\sigma}(\hat{a})$ such that $\hat{a}' \in \{(f, \widehat{op}) : (f, \widehat{op}) \in dom(\hat{\sigma})\}$. The formulated abstract garbage collection semantics constructs the subroutine eagc that is called in Alg. 4, which is the interface to enable abstract garbage collection in the reachability algorithm.

B. Abstract garbage collection enhanced with liveness analysis

Abstract garbage collection can avoid conflating abstract objects for reused variables or formal parameters, but it can not discover "garbage" or "dead" abstract objects in the local scope. The following example illustrates this:

```
bool foo(A a) {
  B b = B.read(a);
  C p = C.doSomething(b);
  return bar(C.not(p));
}
```

Obviously, in the function body foo, b is actually "dead" after the second line. However, näive abstract garbage collection has no knowledge of this. In fact, this is a problem for näive concrete garbage collection [1]. In the realm of static analysis, the garbage value pointed to by b can pollute the exploration of the entire state space.

In addition, in the register-based byte code that our implementation analyzes, there are obvious cases where the same register is reassigned multiple times at different sites within a method. The direct adaptation of abstract garbage collection to an object-oriented setting in Section V-A cannot collect these registers between uses. In other words, for object-oriented programs, we also want to collect "dead" registers, even though they are reachable under description in Section V-A. This can be easily achieve by using liveness analysis. Of course, we could also solve it by transforming the byte code into Static Single Assignment (SSA) form. However, as mentioned above, liveness analysis has additional benefits, so we chose to enhance the abstract garbage collection with live variable analysis (LVA).

LVA computes the set of variables that are *alive* at each statement within a method. The garbage collector can then more precisely collect each frame.

Since LVA is well-defined in the literature [2], we skip the formalization here, but the $Root$ is now modified to collect only *live* variables of the current statement $Lives(s_0)$:

$$Root(\vec{s}, \hat{fp}, \hat{\sigma}, \hat{\kappa}) = \{(v', \hat{fp})\} \cup StackRoot(\hat{\kappa}),$$
$$\text{where } (v', \hat{fp}) \in dom(\hat{\sigma}) \text{ and } v' \in Lives(s_0).$$

The liveness property is embedded in the overall eagc subroutine in Alg. 4.

VI. EXTENDING PUSHDOWN REACHABILITY ANALYSIS FOR EXCEPTIONS

Given the formalisms in the previous sections, it is not immediately clear how to convert these rules into a static analyzer, or more importantly, how to handle the unbounded stack without it always visiting new machine configurations. Thus, we need a way to compute a finite summary of the reachable machine configurations.

In abstract interpretation frameworks, the Dyck State Graph synthesis algorithm [9], which is a purely functional version of the Saturation algorithm [26], provides a method for computing reachable pushdown control states. We build our algorithms on the work of Earl *et al.* [10]. As it turns out, it is not hard to extend the summarization idea to deal with an unbounded stack with exceptions. In the following sections, we present the complete algorithm in a top-down fashion, which aims to easily turn into actual working code. The algorithm code uses previous definitions specified in Section IV.

A. Analysis setup

Algorithm 1: ANALYZE

Input: s: a list of program statements (with an initial entry point s_0).
Output: Dyck State Graph DSG : a triple of a set of control states a set of edges, and a initial state.

1   $\hat{\sigma}_0 \longleftarrow$ empty store
2   $\hat{fp}_0 \longleftarrow$ initial empty stack frame pointer
3   $\hat{t}_0 \longleftarrow$ empty list of contexts
4   $\hat{q}_0 \longleftarrow (s_0, fp_0, \hat{\sigma}_0, \hat{t}_0)$
5   The initial working set $W_0 \longleftarrow \{\hat{q}_0\}$
6   $IECG_0 \longleftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
7   $DSG_0 \longleftarrow (\{\hat{q}_0\}, \emptyset, \hat{q}_0)$
8   $(DSG, IECG, \hat{\sigma}, W) \longleftarrow$ EVAL$(DSG_0, IECG_0, \hat{\sigma}_0, W_0)$

9   **return** DSG

The analysis for a program starts from the ANALYZE function, as shown in Alg. 1. It accepts a program expression (an entry point to a program), and gives out a Dyck State Graph (DSG). Formally speaking, a DSG of a pushdown system is the subset of a pushdown system reachable over legal paths. (A path is *legal* if it never tries

to pop a when a frame other than a is on top of the stack.) Note that the $\widehat{Time}$ component is designed for accommodating traditional analysis, depending on actual implementation. For example, the last k call sites or object-allocation labels, or the mix of them. The analysis produces DSG from the subroutine EVAL, which is the fix-point synthesis algorithm.

In Alg. 1, $IECG$ is a composed data structure used in the ϵ summarization algorithm. It is derived from the idea of an ϵ closure graph (ECG) in the work of Earl *et al.* [9], but supports efficient caching of ϵ closures along with transitive push frames on the stack. Specifically, $IECG = (\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}, \overrightarrow{PSF}, \overleftarrow{PFP}, \overleftarrow{NEP})$. The six components can be considered maps:

- ϵ predecessors $\overleftarrow{G} : \hat{\Sigma} \to \{\hat{\Sigma}\}$, maps a target node to the source node(s) of an ϵ edge(s)
- ϵ successors $\overrightarrow{G} : \hat{\Sigma} \to \{\hat{\Sigma}\}$, maps a source node to the target node(s) of an ϵ edge(s)
- top frames $\overrightarrow{TF} : \hat{\Sigma} \to \{\widehat{Frame}\}$, records the shallow pushed stack frame(s) for a state node.
- possible stack frames $\overrightarrow{PSF} : \hat{\Sigma} \to \{\widehat{Frame}\}$, compute all possible pushed stack frame of a state. It is used for abstract garbage collection.
- predecessors for push action $\overleftarrow{PFP} : (\hat{\Sigma}, \widehat{Frame}) \to \{\hat{\Sigma}\}$, records source state node(s) for a pushed frame and the net-changed state. For example in the legal path: $\hat{q}_0 \xrightarrow{g^+} \hat{q}_1 \longrightarrow ... \xrightarrow{g^-} \hat{q}_2$, the entry $(\hat{q}_1, g^+) \longrightarrow \{\hat{q}_0\}$ is in $\overleftarrow{PFP}$.
- non-ϵ predecessors ($\overleftarrow{NEP} : \hat{\Sigma} \to \{\hat{\Sigma}\}$), maps a state node to non-ϵ predecessors.

These data structures (and $IECG$) have the same definition in the following algorithms.

1) Fix-point algorithm of the pushdown exception framework: Alg. 2 describes the fix-point computation for the reachability algorithm. It iteratively constructs the reachable portion of the pushdown transition relation (Ln. 5-12) by inserting ϵ-summary edges whenever it finds empty-stack (Ln. 13-20) (*e.g.*, push a, push b, pop b, pop a) paths between control states. Ln. 22-25 decides when to terminate the analysis: no new frontier edges and the new store component $\hat{\sigma}''$ is subsumed by the old store $\hat{\sigma}'$. The second condition uses the technique presented by Shivers [29]. Otherwise, it recurs to EVAL.

Now we explain Ln. 5-12 in more detail by examining the subroutines that are called. As is shown in Ln. 7, the *raw* new states and edges are obtained from calling STEP (shown in Alg. 3). The algorithm enables the widening strategy in the pushdown reachability algorithm by instrumenting the $\hat{\sigma}$ component (it is *widened* during iteration in EVAL (Ln. 7 and Ln. 12)).

The other important part of the algorithm is STEPIPDS, Alg. 4 shows the details. STEPIPDS does three things: (1) It incorporates the enhanced abstract garbage collection into the pushdown framework by calling `eagc` (Ln. 3). The actual algorithm can be derived from the semantics presented in the Section V; (2) It calls the pushdown abstract transition relation function of `next` based on the cleaned state after garbage collection. The semantics presented in Section IV reflect the structure of `next`; (3) It summarizes the stack actions from the newly explored nodes, and so to construct possible edges for DSG. This is done in the algorithm DECIDESTACKACTION, which compares the continuation before the transition and the continuation after, then decides which of the three stack actions: epsilon, push and pop to take. Also note that we add only state nodes (e.g. $\hat{q}$) into the working set if they are not appeared in the following sets: state nodes of the current DSG, predecessors of $\hat{q}$ and successors of $\hat{q}$, for the purpose of avoiding non-necessary re-computation.

Algorithm 2: EVAL

Input: $DSG, IECG$(definition referred to Section VI-A), $\hat{\sigma}$, working set W
Output: $DSG', IECG', \hat{\sigma}'', W'$
1   $\Delta S, \Delta E, \hat{\sigma}'', W' \leftarrow \emptyset$
2   $(E, S, \hat{q}_0) \leftarrow DSG$
3   $(\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}, \overrightarrow{PSF}, \_, \_) \leftarrow IECG$
4   $IECG' \leftarrow (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$
5   **for** $s \in W$ **do**
6   **for** $\hat{\kappa} \in \overrightarrow{TF}(s)$ **do**
7   **for** $(g, s_1, \hat{\sigma}') \in$ STEP$(s, \hat{\kappa}, \overrightarrow{PSF}(s), \hat{\sigma})$ **do**
8   **if** $s \notin (\overrightarrow{G}(s) \cup S \cup \overleftarrow{G}(s))$ **then**
9   insert s_1 in ΔS
10   insert $E(s, g, s_1)$ in ΔE
11   insert s_1 in W'
12          $\hat{\sigma}'' = \hat{\sigma}' \sqcup \hat{\sigma}$
13   **for** $E \in \Delta E$ **do**
14   **switch** E **do**
15   **case** (s', ε, s'')
16           $IECG' \leftarrow$ PROPAGATE$(E, IECG)$
17   **case** (s', g^+, s'')
18           $IECG' \leftarrow$ PROCESSPUSH$(E, IECG)$
19   **case** (s', g^-, s'')
20           $IECG' \leftarrow$ PROCESSPOP$(E, IECG)$
21     $DSG' \leftarrow (E \cup \Delta E, S \cup \Delta S)$
22   **if** $\hat{\sigma}'' \sqsubseteq \hat{\sigma} \wedge \Delta E == \emptyset$ **then**
23   **return** $(DSG', IECG', \hat{\sigma}'', W')$
24   **else**
25   EVAL$(DSG', IECG', \hat{\sigma}'', W')$

Algorithm 3: STEP

Input: control state $\hat{q}$, continuation $\hat{\kappa}$, a list of frames $\vec{\phi}$
Output: a set of records (stack action g, $\hat{q}'$, $\hat{\sigma}$)
1   $result \longleftarrow \emptyset$
2   **for** $(g, \hat{q}') \in$ STEPIPDS$(\hat{q}, \hat{\kappa}, \vec{\phi})$ **do**
3   insert $(g, \hat{q}', \hat{\sigma})$ in $result$
4   **return** $result$

Returning to EVAL in Alg. 2, Ln. 13-20 summarizes and propagates the new knowledge of the stack, given ΔE, by calling the Alg. 6, Alg. 8, or Alg 9 based on the stack action. These algorithms are detailed in Section VI-B, along with the mechanism to deal with exceptions.

B. Synthesizing a Dyck State Graph with exceptional flow

For pushdown analysis *without exception handling*, only two kinds of transitions can cause a change to the set of ϵ-predecessors ($\overleftarrow{G}$): an intra-procedural empty-stack transition and a frame-popping procedure return. With the addition of **handle** frames to the stack, there are several new cases to consider for popping frames (and hence adding ϵ-edges).

In the following text, we first highlight how to handle the exceptional flows during DSG synthesis, particularly as it relates to maintaining ϵ-summary edges. Then we present the generalized algorithms for these cases. The figures in this section use a graphical scheme for describing the cases for ϵ-edge insertion. Existing edges are solid lines, while the ϵ-summary edges to be added are dotted lines. The superscripts of $+$ and $-$ on exception handler frame η, call frame χ and general frame ϕ mean push or pop actions of the correspondent frames.

Intraprocedural push/pop of handle frames: The simplest case is entering a `try` block and leaving a `try` block entirely intraprocedurally—without throwing an exception. Figure 6a shows such a case: if there is a handler push followed by a handler pop, the synthesized (dotted) edge must be added.

Algorithm 4: STEPIPDS

Input: a source state $\hat{q}$, continuation $\hat{\kappa}$, list of frames $\vec{\hat{\phi}}$, *Options*: global analysis options
Output: a set of tuples $(\vec{\hat{\phi}}', \hat{q}')$
1   $result \longleftarrow \emptyset$
2   $\hat{q}' \longleftarrow \hat{q}$
3   **if** $Options.doGC$ **then** $\hat{q}' \longleftarrow \text{eagc}(\hat{q}, \vec{\hat{\phi}})$
4   $confs \longleftarrow \text{next}(\hat{q}', \hat{\kappa})$
5   **for** $(\hat{q}'', \hat{\kappa}') \in confs$ **do**
6     $g \longleftarrow \text{DECIDESTACKACTION}(\hat{\kappa}, \hat{\kappa}')$
7   insert $(g, \hat{q}'')$ in $result$
8   **return** $result$

Algorithm 5: DECIDESTACKACTION

Input: continuation before transition $\hat{\kappa}$, new continuation $\hat{\kappa}'$
Output: stack action g
1   **if** $\hat{\kappa} = \hat{\kappa}'$ **then return** ϵ
2   $(g_1 :: \hat{\kappa}_1) \longleftarrow \hat{\kappa}$
3   $(g_2 :: \hat{\kappa}_2') \longleftarrow \hat{\kappa}'$
4   **if** $\hat{\kappa}_1 == \hat{\kappa}'$ **then**
5   **return** g_1^-
6   **else if** $\hat{\kappa} == \hat{\kappa}_2'$ **then**
7   **return** g_2^+

Locally caught exceptions: Figure 6c presents a case where a local handler catches an exception, popping it off the stack and continuing.
**Exception propagation along the stack**: Figure 6b illustrates a case where an exception is not handled locally, and must pop off a call frame to reach the next handler on the stack. In this case, a popping self-edge from control state q' to q' lets the control state q' see frames beneath the top. Using popping self-edges, a single state can pop off as many frames as necessary to reach the handle—one at a time.
Control transfers mixed in try/catch: Figure 6d illustrates the situation where a procedure tries to return while a **handle** frame is on the top of the stack. It uses popping self-edges as well to find the top-most **call** frame.
Uncaught exceptions: The case in Figure 6e shows popping all frames back to the bottom of the stack—indicating an uncaught exception.
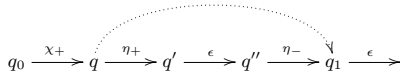
C. The generalized algorithms: PROPAGATE, PROCESSPOP, PROCESSPUSH

Section VI-B graphically illustrates the new cases for handling exceptions (Figure 6). The following text presents the working algorithms to achieve the synthesis process. Alg. 6 handles the cases when
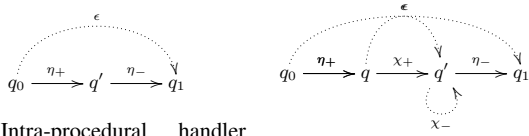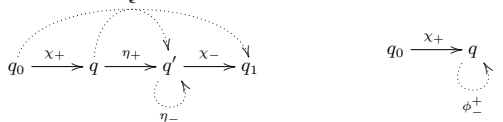

(a) Intra-procedural handler push/pop

(b) Exception propagation

(c) Locally caught exceptions

(d) Control transfers mixed in try/catch

(e) Uncaught exceptions

Fig. 6: Synthesizing a DSG with exceptional flow

an ϵ edge is added. These cases are: intra-procedural empty-stack transition, a frame-popping procedure return, or a frame-popping intra-procedural or inter-procedural exception catch, as presented in Figure 6.

Algorithm 6: PROPAGATE

Input: An edge E, an $IECG$ (refer to Section VI-A)
Output: $IECG'$
1   $(\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}, \overrightarrow{PSF}, \overleftarrow{PFP}, \overleftarrow{NEP}) \leftarrow IECG$
2   $topFramesToAdd \leftarrow \emptyset$
3   $\overleftarrow{G}', \overrightarrow{G}', \overrightarrow{TF}', \overrightarrow{PSF}', \overleftarrow{PFP}', \overleftarrow{NEP} \leftarrow \emptyset$
4   $(s_1, \varepsilon, s_2) \leftarrow E$
5   $preds \leftarrow \overleftarrow{G}(s_1) \cup \{s_1\}$
6   $nexts \leftarrow \overrightarrow{G}(s_2) \cup \{s_2\}$
7   **for** $s \in preds$ **do**
8     $\overrightarrow{G}' \leftarrow \overrightarrow{G} \sqcup [s \mapsto \overrightarrow{G}(s) \cup nexts]$
9   insert $\overrightarrow{TF}(s)$ in $topFramesToAdd$
10   **for** $s \in nexts$ **do**
11     $\overleftarrow{G}' \leftarrow \overleftarrow{G} \sqcup [s \mapsto \overleftarrow{G}(s) \cup preds]$
12     $\overrightarrow{TF}' \leftarrow \overrightarrow{TF} \sqcup [s \mapsto \overrightarrow{TF}(s) \cup topFramesToAdd]$
13   **for** $f \in \overrightarrow{TF}(s_1)$ **do**
14       $\overleftarrow{PFP}' \leftarrow \overleftarrow{PFP} \sqcup [(s, f) \mapsto \overleftarrow{PFP}(s, f)]$
15     $\overrightarrow{PSF}' \leftarrow \text{UPDATEPSF}(s, \overrightarrow{TF}', \overrightarrow{PSF}, \overleftarrow{NEP}, \overleftarrow{G}')$
16   $IECG' \leftarrow (\overleftarrow{G}', \overrightarrow{G}', \overrightarrow{TF}', \overrightarrow{PSF}', \overleftarrow{PFP}', \overleftarrow{NEP})$
17   **return** $IECG'$

The algorithm works as follows: It accepts an ϵ edge E and the current record of $IECG$ (introduced in Section VI-A) and produces a new IECG $IECG'$. It propagates the ϵ successors for each control state in $\overleftarrow{G}(s_1) \cup s_1$ (Ln. 8) and prepares the accumulated top frames for propagation for each successor state node in $\overrightarrow{G}$ (Ln. 12). Similarly, it propagates the ϵ predecessors for each control state in $\overrightarrow{G}(s_2) \cup s_2$. The predecessor nodes of pushed frames for the current target note state s will also be propagated with the new propagated top frames (Ln. 13-14). Finally, it propagates the possible stack frames $\overrightarrow{PSF}$ (for abstract garbage collection) in Ln. 15, for each control state in the original non-ϵ predecessors and new ϵ predecessors $\overleftarrow{G}$, as shown in Alg. 7 Ln. 2-3.

Algorithm 7: UPDATEPSF

Input: $s, \overrightarrow{TF}', \overrightarrow{PSF}, \overleftarrow{NEP}, \overleftarrow{G}'$
Output: $\overrightarrow{PSF}''$
1   $\overrightarrow{PSF}' \leftarrow \overrightarrow{PSF} \sqcup [s \mapsto \overrightarrow{TF}'(s)]$
2   **for** $spred \in \overleftarrow{NEP}(s) \cup \overleftarrow{G}'(s)$ **do**
3     $\overrightarrow{PSF}'' \leftarrow \overrightarrow{PSF}' \sqcup [s \mapsto \overrightarrow{PSF}'(spred)]$
4   **return** $\overrightarrow{PSF}''$

Alg. 8 handles the case of popping frames, including function call return popping and exception handling popping. The algorithm is reduced to Alg. 6 to introduce ϵ edges, for each tuple in $\overleftarrow{PFP}$.

Algorithm 8: PROCESSPOP

Input: $E, IECG$
Output: $IECG'$
1   $IECG' \leftarrow \emptyset$ $(s_1, g^-, s_2) \leftarrow E$
2   **for** $s \in \overleftarrow{PFP}(s_1, g^-)$ **do**
3     $IECG' \leftarrow IECG \sqcup \text{PROPAGATE}((s, \varepsilon, s_2), IECG)$
4   **return** $IECG'$

Alg. 9 is presented for completeness. It handles pushing stack frames in function calls and exception handlers in `try` blocks. Since the pushing action introduces a new top frame, it maintains extensions (propagation) for the data structure top frames $\overrightarrow{TF}$, predecessors for push frames $\overleftarrow{PFP}$, non-ϵ predecessors $\overleftarrow{NEP}$ and possible stack frames $\overrightarrow{PSF}$.

Algorithm 9: PROCESSPUSH

Input: $E, IECG$
Output: $IECG'$

1   $IECG' \ (\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}, \overrightarrow{PSF}, \overleftarrow{NEP}) \leftarrow IECG$
2   $\overleftarrow{G}', \overrightarrow{G}', \overrightarrow{TF}', \overrightarrow{PSF}', \overrightarrow{PFP}', \overleftarrow{NEP}' \leftarrow \emptyset$
3   $(s_1, g^+, s_2) \leftarrow E$
4   **for** $s \in \overrightarrow{G}(s_2) \cup \{s2\}$ **do**
5      $\overrightarrow{TF}' \leftarrow \overrightarrow{TF} \sqcup [s \mapsto \{f\}]$
6      $\overrightarrow{PFP}' \leftarrow \overrightarrow{PFP} \sqcup [(s,f) \mapsto \{s_1\}]$
7      $\overleftarrow{NEP}' \leftarrow \overleftarrow{NEP} \sqcup [s \mapsto \{s_1\}]$
8      $\overrightarrow{PSF}' \leftarrow \text{UPDATEPSF}(s, \overrightarrow{TF}', \overrightarrow{PSF}, \overleftarrow{NEP}', \overleftarrow{G})$
9   $IECG' \leftarrow (\overleftarrow{G}, \overrightarrow{G}, \overrightarrow{TF}', \overrightarrow{PSF}', \overrightarrow{PFP}', \overleftarrow{NEP}')$
10 **return** $IECG'$

VII. IMPLEMENTATION

We have implemented the analysis framework[1] with pushdown abstraction and enhanced abstract garbage collection to analyze Android applications, which are Java programs. The analyzer works directly on Dalvik bytecode, which is compiled from Java programs into Dalvik Virtual Machine (DVM). Different from Java bytecode, Dalvik bytecode is register-based. What's more important, it closely resembles the high-level Java source code. We choose to work on bytecode in implementation for two reasons: (1) The semantics of Dalvik bytecode is almost identical to that of high-level Java while bringing more advantage in analyzing `finally`. (2) It enables us to analyze off-the-shelf Android applications.

The `finally` blocks: In previous sections, we described the semantics and algorithms for `try/catch`. To analyze full-featured exceptions, we have to deal with the `finally` blocks. It is known to be non-trivial to handle `finally` in static analysis [8]. However, this is not a problem in our analysis. The reason is that the analyzer directly works on object-oriented byte code, where the `finally` is compiled away by compiler in this level. Specifically, the blocks of code for `finally` are copied into try and catch blocks before any possible exit points, which include normal `return` statements or `throw` statements. This eases the static analysis substantially. In addition, `finally` blocks are translated as one kind of `catch` handler, which is the `catchall` handler, with the exception type `java/lang/Exception`. During the pushdown analysis, `catchall` is placed below any other normal `catch` handlers on the stack, it is matched last and executed for any possible `throw` exceptions.

VIII. EVALUATION

To evaluate the effectiveness of our analysis technique, we compare our analysis with one of the well-known finite-state based static analysis frameworks—WALA.[2] In fact, there are two representative traditional static analysis frameworks for object-oriented programs, Doop [7] and WALA. They are both finite-state static analysis but orthogonal work. For this reason, there are no comparison results reported in the literature for the two analysis frameworks. However, we still experimented with Doop [7] virtual image provided by the Doop authors. However, the results were incomplete due to significantly slower running times on several of the DaCapo [4] benchmarks. As a result, we do not feel a fair comparison can be made.

As it turns out, WALA is based on the work of Reps *et al.* [26], which was later formalized into pushdown reachability. In this sense,

WALA is more similar to our approach with respect to pushdown reachability. Therefore, we compare our analysis with WALA. In specific, WALA mainly adopts co-analysis of control-flow and data-flow analyses, performing call-graph construction and pointer analysis together, by propagating pointer information on the constructed CFG. The framework provides several context-sensitivities [31], including 0-CFA, 0-1-CFA (0-CFA with 1-object sensitivity), and analysis with additional disambiguation of container elements 0-container and 0-1-container. [3] In particular, the 0-1-CFA enables several optimizations for string and thrown objects. The 0-1-container policy extends the 0-1-CFA with unlimited object-sensitivity for collection objects, which is the most precise default option. Our evaluation uses the 0-1-container as the baseline.

To make the comparison more compelling, we conduct experiments on the DaCapo [4] benchmarks. It has much larger scale code bases to analyze than ordinary Java applications presented in the Google market. This allows a more realistic workload to stress-test the analysis. Due to some conflicts in Java GUI classes, `eclipse` can not be ported in DVM. Other 10 programs out of 11 Java applications in the DaCapo benchmark suffice for our purpose.

A. Metrics for precision

Our basis for comparison in precision is the average cardinality of a points-to set [12], [6], [15] and exception-catcher links (E-C links) [12].

The average cardinality of a points-to set computes the average number of abstract (exception) objects for pointers that are collected into a single representative in the abstraction. In our evaluation, it has two forms: `VarPointsTo` and `Throws`. `VarPointsTo` refers to the average cardinality of the points-to set for non-exception abstract objects, and `Throws` refers to exception objects specifically. (In Table I, we normalized the two metrics computed in WALA, relative to that in our analysis.). We adopt this metric because it reflects analysis precision by recognizing that the more objects are conflated for a variable, the less precise the analysis. When this metric is a large value, it indicates a negative impact on normal control-flow analysis because it means that virtual method resolution needs to dynamically dispatch to more than one function causing spurious control-flow paths. This same reasoning applies for exception-flow analysis. (The more subtle relationships have been illustrated in Section I). More rationals of using this metric to measure precision for object-oriented programs are illustrated in the literature [12], [6], [15]. Following WALA's heap model, we compute the same metric in our pushdown framework.

The E-C links, proposed by Fu *et al.* [12] is to reflect the precision of handling exceptional flows. It is also used in the work of [6]. We compute the metric in our analysis framework, which is within the range of 1-3 across the DaCapo benchmarks. Because WALA directly computes the catchers intra-procedurally, we do not compute the comparison ratio as we do for `VarPointsTo` and `Throws`.

In addition, we also evaluated the precision of our pruned, pushdown analysis with respect to the client security analysis. We refer readers to the related work [22].

B. Results

Table I shows that the pushdown exception-flow analysis with enhanced abstract garbage `pdxfa+eagc` outperforms finite-state context-sensitive analysis (represented by WALA) with a precision of 4.5-11 times for `Throws` and up to 7 times for general points-to

---

Benchmark	LOC	Opts	Nodes	Edges	Methods	VarPointsTo*	Throws*
antlr	35,000	pdxfa+1obj	4.1x	1.3x	1.2x	1.5x	2.8x
		pdxfa + eagc	3.9x	1x	1x	3x	4.6x
bloat	70,344	pdxfa+1obj	1.9x	1.4x	2.4x	3.3x	2.4x
		pdxfa + eagc	1.2x	1.3x	1.1x	6.3x	6x
chart	217,788	pdxfa+1obj	2.3x	1.3x	1.1x	2x	2.3x
		pdxfa + eagc	2.1x	1.1x	1.2x	6x	4.5x
fop	184,386	pdxfa+1obj	2.1x	1.4x	1.1x	4.2x	5.5x
		pdxfa + eagc	1.9x	1.3x	1.5x	7.3x	11x
hsqldb	155,591	pdxfa+1obj	8.9x	4.4x	3.4x	1x	2.3x
		pdxfa + eagc	5.3x	2.7x	3.3x	3x	4.5x
luindex	38,221	pdxfa+1obj	1.9x	1.9x	1.8x	1x	1.6x
		pdxfa + eagc	3.5x	1.7x	1.2x	1.5x	4x
lusearch	87,000	pdxfa+1obj	1.5x	1.6x	1.6x	1.6x	2.3x
		pdxfa + eagc	1x	1.5x	1.4x	2.5x	4.5x
pmd	55,000	pdxfa+1obj	1.8x	1.3x	1.5x	2.2x	5.2x
		pdxfa + eagc	1.5x	1.1 x	1x	3.7x	7.7x
xalan	159,026	pdxfa+1obj	1.9x	1.3x	1.7x	2.8x	6.2x
		pdxfa + eagc	1.4x	1.2x	1.3x	3.7x	10.3x

TABLE I: Precision comparison. Values in columns `Nodes`, `Edges` and `Methods` are ratios of the number of nodes, edges and methods reached in our analysis, relative to the ones in WALA respectively. Values in columns `VarPointsTo*` and `Throws*` are ratios of average cardinality of general point-to set and exception points-to set in WALA, relative to the ones in our analysis receptively. Note that we did not list the results for the benchmark `jython` because it runs out of memory after one hour. *The table shows that the pushdown exception-flow analysis with enhanced abstract garbage collection `pdxfa+eagc` outperforms finite-state analysis in WALA in precision by 4.5X-11X for `Throws` and up to 7X for general points-to information `VarPointsTo`.*

information `VarPointsTo`. `Nodes` and `Edges` are control-flow graph information. `Methods` denotes the analyzed methods. The values in these columns in Table I are normalized relative to those reported by WALA 0-1-contain analysis. As is shown in Table I, our pruned, pushdown analysis technique (`pdxfa+eagc`) generally explores more edges and nodes, and explores up to 3.4 times more methods.

To evaluate the contribution of each aspect (pushdown exception-flow analysis and enhanced abstract garbage collection) to precision improvement, when comparing with WALA, we also conduct an additional experiment with only the pushdown exception-flow analysis with 1-object sensitivity (as WALA 0-1-container does), denoted as the option `pdxfa+1obj`. The result shows that the `pdxfa` improves the precision more than enhanced abstract garbage collection does.

C. Analysis time

For completeness, we also report an analysis time comparison. Table II is the ratio of our analysis time to that of WALA.

WALA reports less analysis time than our analysis. This is not surprising. First, our analysis is derived from the polynomial complexity algorithm in [26], [10]. Even with enhanced garbage collection, it only reduces the complexity by a constant factor. Second, WALA has been significantly optimized by the IBM research lab, particularly with underlying Java (collection) libraries rewritten specifically for its framework. Our implementation is based on Scala's default data structures and our specialized Gödel hashing data structure [23]. Last but not least, the analysis time is reasonably acceptable, given the high precision that our analysis technique can provide. For example, for the largest benchmark `chart`, the unoptimized analyzer takes roughly 13 minutes.

IX. RELATED WORK

Exception Analysis The bulk of the earlier literature for analyzing Java programs has generally focused on finite-state abstractions, *i.e.*, k-CFA and its variants. Specifically, for the work that acknowledges exceptional flows, the analysis is based on either context-insensitivity or a limited form of context-sensitivity. Analyzers that use only syntactic, type-based analysis of exceptional flow are extremely imprecise [19], [28]. Propagating exceptions via the imprecise call graphs cause the analysis result in: (1) inclusion of many spurious paths between exception throw sites and handlers that are not truly realizable at run time; (2) unable to tell and differentiate where an exception comes from. Fu *et al.* [12] approached the problem by employing points-to information to refine control-flow reachability. Later, Bravenboer and Smaragdakis exploited this mutual recursion by co-analyzing data- and exception-flow [6]. It reports precision improvement in both pointer-to analysis and exception analysis.

Points-to Analysis Precise and scalable context-sensitive points-to analysis has been an open problem for decades. We describe a portion of the representative work in the literature. Much work in pointer analysis exploits methods to improve performance by strategically reducing precision. Lattner *et al.* show that an analysis with a context-sensitive heap abstraction can be efficient by sacrificing precision under unification constraints [18]. In full-context-sensitive pointer analysis, Milanova *et al.* found that an object-sensitive analysis [25] is an effective context abstraction for object-oriented programs. BDDs have been used to compactly represent the large amount of redundant data in context-sensitive pointer analysis efficiently [3], [34], [35]. Such advancements could be applied to our pushdown framework, as they are orthogonal to its central thesis. Recently, Khedker *et al.* [16] exploits liveness analyses to improve points-to analysis. Our work also uses liveness analyses but extends it to work with abstract garbage collection. In fact, to the best of our knowledge, we are the first work that explores abstract garbage collection in analyzing object-oriented programs and enhances it with liveness analysis to explicitly prune points-to precision.

Pushdown analysis for the λ-calculus Vardoulakis and Shivers's CFA2 [33] is the precursor to the pushdown control-flow analysis [9]. Our work directly draws on the work of pushdown analysis for higher-order programs [9] and introspective pushdown system (IPDS) for higher-order programs [10]. We extend the earlier work in three dimensions: (1) We generalize the framework to an object-oriented language; (2) We adapt the Dyck state graph synthesis algorithm to handle the new stack change behavior introduced by exceptions; (3) We reveal necessary details to design and implement a static analyzer even in the exceptions.

CFL- and pushdown-reachability techniques Earl *et al.* [10] develop a pushdown reachability algorithm suitable for the push-

Benchmark	antlr	bloat	chart	fop	hsqldb	luindex	lusearch	pmd	xalan
Ratio	8.5x	5.6x	9.7x	7.9x	5.2x	3.1x	8.7x	9x	4.7x

TABLE II: Analysis time

down systems that we generate. It essentially draws on CFL- and pushdown-reachability analysis [5], [17], [26], [27]. For instance, epsilon closure graphs, or equivalent variants thereof, appear in many context-free-language and pushdown reachability algorithms. Dyck state graph synthesis is an attractive perspective on pushdown reachability because it allows targeted modifications to the algorithm.

Pushdown exception-flow analysis There is few work on pushdown analysis for object-oriented languages as a whole. Sridharan and Bodik proposed demand-driven analysis for Java that matches reads with writes to object fields selectively, by using refinement [30]. They employ a refinement-based CFL-reachability technique that refines calls and returns to valid matching pairs, but approximates for recursive calls. They do not consider specific applications of CFL-reachability to exception-flow.

X. Conclusion

Exception-flows are mutually determined by traditional control-flow analysis and points-to analysis. In order to model exceptional control-flow precisely, we abandoned traditional finite-state approaches (e.g. k-CFA and its variants). In its place, we generalized pushdown control-flow analysis from the λ-calculus [10] to object-oriented programs, and made it capable of handling exceptions in the process. Pushdown control-flow analysis models the program stack (precisely) with the pushdown stack, for the purpose of pruning control-flows. To prune the precision with respect to points-to information, we adapted abstract garbage collection to object-oriented program analysis and enhanced it with live variable analysis. Computing the reachable control states of the pushdown system (the enhanced Dyck state graph) yields combined data-flow analysis and control-flow analysis of a program. Comparing this approach to the established traditional analysis framework shows *substantially* improved precision, within a reasonable analysis time.

XI. Acknowledgments

References

[1] O. Agesen, D. Detlefs, and J. E. Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. In *PLDI'98*. ACM, 1998.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI'03*. ACM, 2003.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06*. ACM, 2006.

[5] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: application to model-checking. In *Proceedings of the 8th International Conference on Concurrency Theory*, pages 135–150. Springer-Verlag, 1997.

[6] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA'09, pages 1–12. ACM, 2009.

[7] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09*, pages 243–262. ACM, 2009.

[8] R. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of points-to analysis of java in the presence of exceptions. *IEEE Trans. Softw. Eng.*, 2001.

[9] C. Earl, M. Might, and D. Van Horn. Pushdown control-flow analysis of higher-order programs. In *Proceedings of the 2010 Workshop on Scheme and Functional Programming*, Aug. 2010.

[10] C. Earl, I. Sergey, M. Might, and D. Van Horn. Introspective pushdown analysis of higher-order programs. In *ICFP'12*. ACM, 2012.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI '93*. ACM, 1993.

[12] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of Java server applications. *IEEE Trans. Softw. Eng.*, 31(4):292–311, Apr. 2005.

[13] L. Hendren. Scaling Java points-to analysis using Spark. In *12th International Conference on Compiler Construction*. Springer, 2003.

[14] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3), May 2001.

[15] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI '13*. ACM, 2013.

[16] U. P. Khedker, A. Mycroft, and P. S. Rawat. Liveness-based pointer analysis. In *SAS'12*, pages 265–282. Springer-Verlag, 2012.

[17] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *PLDI'04*. ACM, 2004.

[18] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI'07*. ACM, 2007.

[19] X. Leroy and F. Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, Mar. 2000.

[20] O. Lhoták. *Program analysis using binary decision diagrams*. PhD thesis, McGill University, 1987.

[21] S. Liang, M. Might, and D. Van Horn. Anadroid: Malware analysis of android with user-supplied predicates. *CoRR*, abs/1311.4198, 2013.

[22] S. Liang, M. Might, D. Van Horn, S. Lyde, T. Gilray, and P. Aldous. Sound and precise malware analysis for android via pushdown reachability and entry-point saturation. In *SPSM*. ACM, 2013.

[23] S. Liang, W. Sun, and M. Might. Fast flow analysis with gödel hashes. In *SCAM*, 2014.

[24] M. Might and O. Shivers. Improving flow analyses via Gamma-CFA: Abstract garbage collection and counting. In *ICFP '06*, pages 13–25. ACM, 2006.

[25] A. Milanova and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol*, 2005.

[26] T. Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11-12):701–726, Dec. 1998.

[27] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Science of Computer Programming*, 58(1-2):206–263, Oct. 2005.

[28] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, Apr. 2003.

[29] O. G. Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

[30] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI'06*, pages 387–400. ACM, 2006.

[31] M. Sridharan, S. Chandra, J. Dolby, S. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850, pages 196–232. Springer, 2013.

[32] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP'10*, pages 51–62. ACM, 2010.

[33] D. Vardoulakis. *CFA2: Pushdown flow analysis for higher-order languages*. PhD thesis, Northeastern University, 2012.

[34] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI'04*. ACM, 2004.

[35] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA'08*, pages 225–236. ACM, 2008.