Context-, Flow-, and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems

by Johannes Späth, Karim Ali, Eric Bodden

POPL 2019

Static Analysis Terminology

- *static analysis* computes an approximation of a program's behavior without executing the program
  - e.g., if a variable can hold a specific value
  - e.g., if data flow is possible between two locations

- *flow-sensitive* static analysis
  - distinguish control flow paths

- *context-sensitive* static analysis
  - distinguish approximations by calling contexts

- *field-sensitive* static analysis
  - distinguish different fields of the same object

Why is this difficult?

```
1  public V put(K key, V value) {
2    TreeMap.Entry<K,V> parent = //complex computation done earlier
3    TreeMap.Entry<K,V> e = new TreeMap.Entry<>(key, value, parent);
4    fixAfterInsertion(e);
5  }
6  private void fixAfterInsertion(Entry<K,V> x) {
7    while (x != null && x != root && x.parent.color == RED) {
8      //removed many branches here...
9      x = parentOf(x);
10     rotateLeft(parentOf(parentOf(x)));
11   }
12 }
13 private void rotateLeft(TreeMap.Entry<K,V> p) {
14   if (p != null) {
15     TreeMap.Entry<K,V> r = p.right;
16     p.right = r.left;
17     if (l.right != null) l.right.parent = p;
18     //removed 8 lines with similar field accesses
19     r.left = p;
20     p.parent = r;
21   }
22 }
```

Listing 1. Excerpt code example of TreeMap which is difficult to analyze statically.

Solutions

- limit the sequence of fields accesses (k-limiting)
  - causes analysis imprecision, “over-tainting”

- access graphs - model the language of field accesses using an automaton
  - state explosion problem

- this paper: Synchronized Pushdown Systems (SPDS)
  - builds on the call-PDS (Reps 2003)
  - call-PDS: context-sensitive and flow-sensitive, but field-insensitive
  - this paper develops the notion of a field-PDS
  - SPDS synchronizes the call-PDS and the field-PDS

Pushdown Systems

DEFINITION 1. *A* pushdown system *is a triple* $\mathcal{P} = (P, \Gamma, \Delta)$, *where* P *and* Γ *are finite sets called the* control locations *and the* stack alphabet, *respectively. A* configuration *is a pair* $\langle\langle p, w \rangle\rangle$, *such that* $p \in P$ *and* $w \in \Gamma^*$, *i.e., a control location with a sequence of stack elements. The finite set* Δ *is composed of* rules. *A rule has the form* $\langle\langle p, \gamma \rangle\rangle \rightarrow \langle\langle p', w \rangle\rangle$, *where* $p, p' \in P$, $\gamma \in \Gamma$, *and* $w \in \Gamma^*$.

Call-PDS:

- P (control locations) : program variables
- Γ (stack alphabet): program statements
- Δ (rules) : flow-functions (data-flow effects of statements)

Constructing the Call-PDS

Table 1. The flow function *assignFlow* for normal rules of assignment statements for $\mathcal{P}_{\mathbb{S}}$.

In		Out
Variable ($\mathbb{V}$)	Statement ($\mathbb{S}$)	($\wp(\mathbb{V})$)
x	$x \leftarrow *$	$\varnothing$
y	$x \leftarrow y$	$\{x, y\}$
y	$x.f \leftarrow y$	$\{x, y\}$
y	$x \leftarrow y.f$	$\{x, y\}$

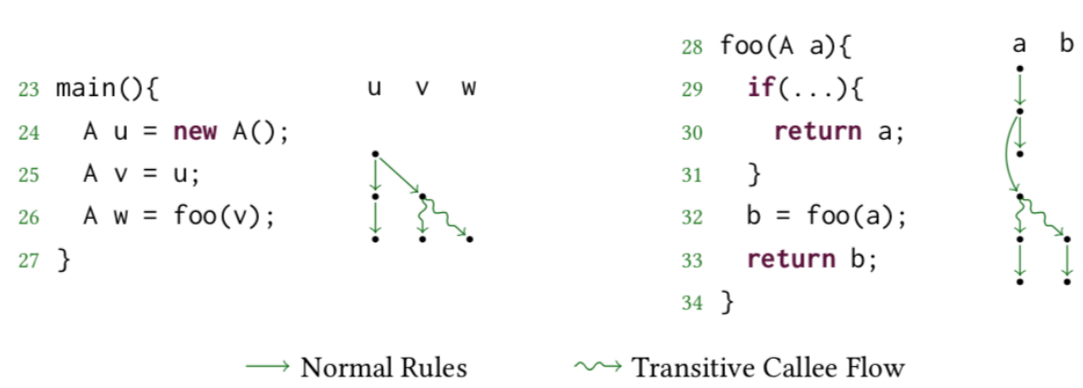


Fig. 1. Data-flow example for a simple recursive program.

- rules capture data flow for each statement
- note that this is a *field-insensitive* analysis

Rules for constructing a Call-PDS

```
23 main(){
24   A u = new A();
25   A v = u;
26   A w = foo(v);
27 }
```


```
28 foo(A a){
29   if(...){
30     return a;
31   }
32   b = foo(a);
33   return b;
34 }
```

⟶ Normal Rules ⇝ Transitive Callee Flow

Fig. 1. Data-flow example for a simple recursive program.

Table 2. Rules of the $\mathcal{P}_{\mathbb{S}}$ for the example in Figure 1.

Normal Rules (main)
$\langle\langle u, 24\rangle\rangle \to \langle\langle u, 25\rangle\rangle$
$\langle\langle u, 24\rangle\rangle \to \langle\langle v, 25\rangle\rangle$
$\langle\langle u, 25\rangle\rangle \to \langle\langle u, 26\rangle\rangle$

Normal Rules (foo)
$\langle\langle a, 28\rangle\rangle \to \langle\langle a, 29\rangle\rangle$
$\langle\langle a, 29\rangle\rangle \to \langle\langle a, 30\rangle\rangle$
$\langle\langle a, 29\rangle\rangle \to \langle\langle a, 31\rangle\rangle$
$\langle\langle a, 32\rangle\rangle \to \langle\langle a, 33\rangle\rangle$
$\langle\langle b, 32\rangle\rangle \to \langle\langle b, 33\rangle\rangle$

Push Rules
$\langle\langle v, 25\rangle\rangle \to \langle\langle a, 28 \cdot 26\rangle\rangle$
$\langle\langle a, 31\rangle\rangle \to \langle\langle a, 28 \cdot 32\rangle\rangle$

Pop Rules
$\langle\langle a, 30\rangle\rangle \to \langle\langle a, \epsilon\rangle\rangle$
$\langle\langle a, 33\rangle\rangle \to \langle\langle a, \epsilon\rangle\rangle$
$\langle\langle a, 33\rangle\rangle \to \langle\langle v, \epsilon\rangle\rangle$
$\langle\langle b, 33\rangle\rangle \to \langle\langle w, \epsilon\rangle\rangle$
$\langle\langle b, 33\rangle\rangle \to \langle\langle b, \epsilon\rangle\rangle$

- normal rules: intraprocedural data-flow
- push rules: data-flow for call to other function
- pop rules: data-flow for return from function call

Creating an Automaton

DEFINITION 2. *Given a pushdown system* $\mathcal{P} = (P, \Gamma, \Delta)$, *a* $\mathcal{P}$-automaton *is a finite non-deterministic automaton* $\mathcal{A} = (Q, \Gamma, \delta, P, F)$ *where* $Q \supseteq P$ *is a finite set of* states, $\delta \subseteq Q \times \Gamma \times Q$ *is the set of transitions and* $F \subseteq Q$ *are the* final states. *The* initial states *are all control locations* P *of the pushdown system* $\mathcal{P}$. *A configuration* $\langle\langle p, w \rangle\rangle$ *is* accepted *by* $\mathcal{A}$, *if the automaton contains a path from state* p *to some final state* $q \in Q$ *such that the word along the path is equal to* w. *We write* $\langle\langle p, w \rangle\rangle \in \mathcal{A}$ *for an accepted configuration.*

- standard algorithm “post*” [Bouajjani et al. 1997, Esparza et al. 2000, Reps et al. 2007]
- starts from an initial configuration c, and applies a “saturation process” to the automaton
  - add edges in accordance with rules until no further edges are added

Creating an Automaton

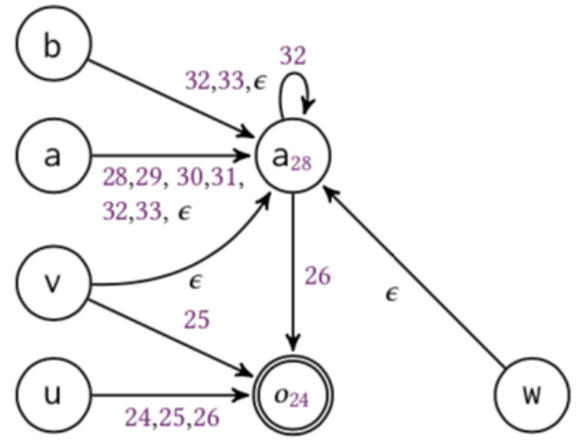
```
23 main(){
24   A u = new A();
25   A v = u;
26   A w = foo(v);
27 }
```


```
28 foo(A a){
29   if(...){
30     return a;
31   }
32   b = foo(a);
33   return b;
34 }
```

⟶ Normal Rules ⇝ Transitive Callee Flow

Fig. 1. Data-flow example for a simple recursive program.

(a) Before post* saturation



(b) After post* saturation

Fig. 2. The automaton $\mathcal{A}_{\mathbb{S}}$ before and after post* saturation with the $\mathcal{P}_{\mathbb{S}}$

Example 2.2. We saturate the $\mathcal{A}_{\mathbb{S}}$ in Figure 2a with the $\mathcal{P}_{\mathbb{S}}$ of Table 2. $\mathcal{A}_{\mathbb{S}}$ initially accepts the configuration $\langle\!\langle \mathsf{u}, 24 \rangle\!\rangle$, which means $\mathcal{A}_{\mathbb{S}}$ tracks the variables that the object allocated in 24 flows to. Therefore, the accepting state of the automaton is labeled by o_{24}, which refers to the object created at this allocation site.

Figure 2b depicts the final post* saturated $\mathcal{A}_{\mathbb{S}}$. The semantics of this automaton is as follows: any variable of any configuration that is accepted within the automaton points-to the object allocated in 24. To keep the figure concise, we visualize transitions between the same states (but with different labels) as a single transition and separate their labels by commas.

Field PDS

DEFINITION 3. *The* pushdown system of fields *is the pushdown system* $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F}, \Delta_{\mathbb{F}})$. *A control location of this system is a pair of a variable and a statement. We use* $x@s$ *for an element* $(x, s) \in \mathbb{V} \times \mathbb{S}$. *The notation emphasizes the fact that* x *holds* at *statement* s. *The pushdown system pushes and pops elements of* $\mathbb{F}$ *to and from the stack. The stack may also be empty, which is represented by the* ϵ *field.*

- P_F pushes and pops fields to and from its stack. Unlike P_S, here the statements that push and pop the fields are not call and return statements but store and load statements. All other statements maintain the field stack unchanged and constitute normal rules of P_F.

.

Constructing the Field-PDS

Table 3. The function *normalFieldFlow* for $\mathcal{P}_{\mathbb{F}}$. Within the comment column t refers to the input statement.

Variable ($\mathbb{V}$)	**Statement** ($\mathbb{S}$)	**Out** ($\wp(\mathbb{V})$)	**Type**	**Comment**
x	$x \leftarrow *$	$\varnothing$	intra	kill info on x at t
y	$x \leftarrow y$	$\{x, y\}$		y copied to x at t
y	$x.f \leftarrow y$	$\{y\}$		info on y is retained at t
y	$x \leftarrow y.f$	$\{y\}$		info on y is retained at t
p_i	$m(p_1, p_2, \ldots, p_n)$	$\{q_i\}$	inter	p_i copied to formal q_i
q_i	**return**	$\{p_i\}$		q_i copied to actual p_i
x	**return** x	$\{y\}$		x copied to assigned value y

Assume $\mathcal{P}_{\mathbb{F}}$ accepts a configuration $\langle\!\langle x@s, g_0 \cdot \ldots \cdot g_n \rangle\!\rangle$ and let t be an intraprocedural control-flow successor of s. Assume further $y \in normalFieldFlow(x, t)$, then $\mathcal{P}_{\mathbb{F}}$ has a rule:

$$\langle\!\langle x@s, g_0 \rangle\!\rangle \rightarrow \langle\!\langle y@\tilde{t}, g_0 \rangle\!\rangle \in \Delta_{\mathbb{F}}^{normal}.$$

- normal rules: defined in terms of auxiliary function normalFieldFlow
- for assignments like x.f = y and x = y.f, both normal rules and push rules apply

Field-PDS: push-rules and pop-rules

3.2 Push Rules ($\Delta_{\mathbb{F}}^{push}$)

When a configuration $\langle\!\langle y@s, g_0 \cdot \ldots \cdot g_n \rangle\!\rangle$ is accepted by $\mathcal{A}_{\mathbb{F}}$ and some successor t of s is a field-store statement, i.e., $t\colon x.f \leftarrow y$, $\mathcal{P}_{\mathbb{F}}$ lists the normal rule $\langle\!\langle y@s, g_0 \rangle\!\rangle \rightarrow \langle\!\langle y@t, g_0 \rangle\!\rangle$. In addition to this normal rule, $\mathcal{P}_{\mathbb{F}}$ lists the following push rule for a field-store statement which pushes a field element onto the stack: $\langle\!\langle y@s, g_0 \rangle\!\rangle \rightarrow \langle\!\langle x@t, f \cdot g_0 \rangle\!\rangle \in \Delta_{\mathbb{F}}^{push}$. The push rule has the following semantic meaning: Whenever the configuration $\langle\!\langle y@s, g_0 \cdot g_1 \cdot \ldots \cdot g_n \rangle\!\rangle$ turns accepting during the computation of post*, the configuration $\langle\!\langle x@t, f \cdot g_0 \cdot g_1 \cdot \ldots \cdot g_n \rangle\!\rangle$ is marked as accepting as well. Speaking in terms of an access path: If an access path $y.g_0 \cdot g_1 \cdot \ldots \cdot g_n$ holds before statement t, the access path $x.f \cdot g_0 \cdot g_1 \cdot \ldots \cdot g_n$ holds after statement t, i.e., the field f is prepended to the access path.

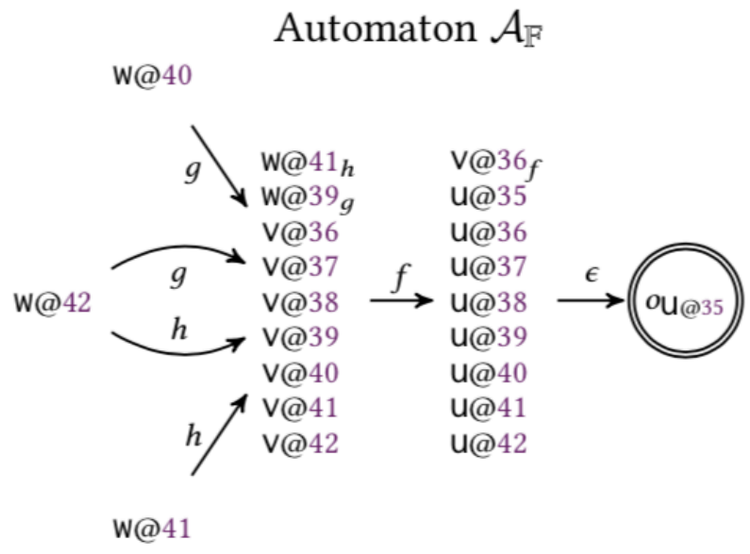
3.3 Pop Rules ($\Delta_{\mathbb{F}}^{pop}$)

The pop rules correspond to the runtime semantics of a field-load statement. For an accepting configuration $\langle\!\langle y@s, f \cdot g_0 \cdot \ldots \cdot g_n \rangle\!\rangle$ where the successor statement t of s is a load statement $x \leftarrow y.f$, $\mathcal{P}_{\mathbb{F}}$ removes a stack element from the field stack, spoken in the form of its rules, the system has a pop rule: $\langle\!\langle y@s, f \rangle\!\rangle \rightarrow \langle\!\langle x@t, \epsilon \rangle\!\rangle \in \Delta_{\mathbb{F}}^{pop}$.

An accepting configuration $\langle\!\langle y@s, f \cdot g_0 \cdot \ldots \cdot g_n \rangle\!\rangle$ induces the configuration $\langle\!\langle x@t, g_0 \cdot \ldots \cdot g_n \rangle\!\rangle$. In other terms, when the access path $y.f \cdot g_0 \cdot \ldots \cdot g_n$ holds before t, the analysis continues to propagate the data flow $x.g_0 \cdot \ldots \cdot g_n$ after statement t.

Example: Field-PDS Rules + Automaton

```
35 foo(u, v, w){
36   v.f = u;
37   x = w.f;
38   if(...){
39     w.g = v;
40   } else {
41     w.h = v;
42   }
43 }
```



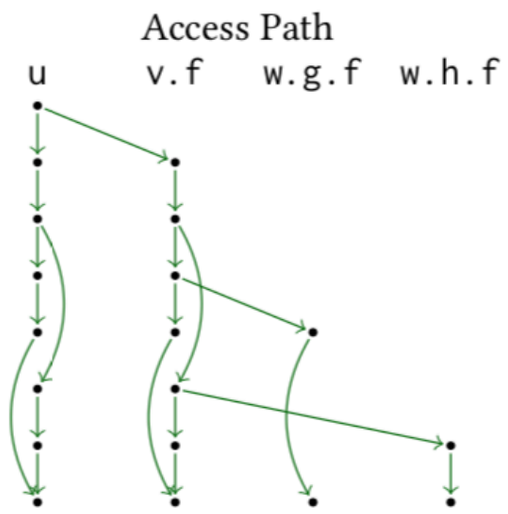


Fig. 3. A post*-saturated $\mathcal{A}_{\mathbb{F}}$ when initialized with the configuration $\langle\!\langle \mathsf{u@35}, \epsilon \rangle\!\rangle$ and saturated with $\mathcal{P}_{\mathbb{F}}$ listed in Table 4. Next to it, the same information represented as standard data-flow graph with an access-path domain.

Table 4. The rule set $\Delta_{\mathbb{F}}$ of $\mathcal{P}_{\mathbb{F}}$ for the code shown in Figure 3.

Normal Rules	
$\langle\!\langle \mathsf{u@35}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@36}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{v@37}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@38}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{u@36}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@37}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{v@38}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@39}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{u@37}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@38}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{v@39}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@42}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{u@38}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@39}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{v@38}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@40}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{u@39}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@42}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{v@40}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@41}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{u@38}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@40}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{v@41}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@42}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{u@40}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@41}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{w@39}, * \rangle\!\rangle \to \langle\!\langle \mathsf{w@42}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{u@41}, * \rangle\!\rangle \to \langle\!\langle \mathsf{u@42}, * \rangle\!\rangle$	$\langle\!\langle \mathsf{w@41}, * \rangle\!\rangle \to \langle\!\langle \mathsf{w@42}, * \rangle\!\rangle$
$\langle\!\langle \mathsf{v@36}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@37}, * \rangle\!\rangle$	

Push Rules
$\langle\!\langle \mathsf{u@35}, * \rangle\!\rangle \to \langle\!\langle \mathsf{v@36}, \mathsf{f} \cdot * \rangle\!\rangle$
$\langle\!\langle \mathsf{v@38}, * \rangle\!\rangle \to \langle\!\langle \mathsf{w@39}, \mathsf{g} \cdot * \rangle\!\rangle$
$\langle\!\langle \mathsf{v@40}, * \rangle\!\rangle \to \langle\!\langle \mathsf{w@41}, \mathsf{h} \cdot * \rangle\!\rangle$

Pop Rules
$\langle\!\langle \mathsf{w@36}, \mathsf{f} \rangle\!\rangle \to \langle\!\langle \mathsf{x@37}, \epsilon \rangle\!\rangle$

Advantage over Access Paths

```
44 foo(a){
45   while(...){
46     b = new B();
47     b.f = a
48     a = b;
49   }
50 }
```

$\langle\!\langle a@46, * \rangle\!\rangle \rightarrow \langle\!\langle b@47, f \cdot * \rangle\!\rangle$

$\langle\!\langle a@48, * \rangle\!\rangle \rightarrow \langle\!\langle a@45, * \rangle\!\rangle$

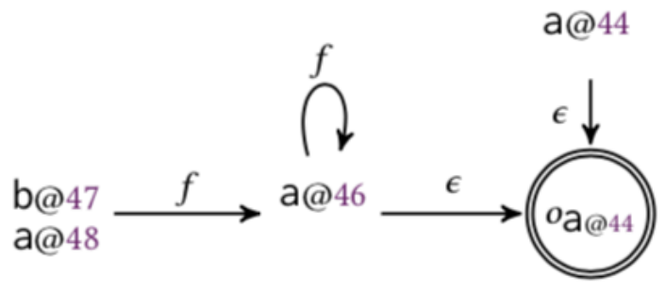


Fig. 4. $\mathcal{P}_{\mathbb{F}}$ and its finite representation of an infinite set of access paths.

- Field-PDS for the TreeMap example in the paper
- the takeaway: *infinitely many access paths are represented in a finite representation!*

Synchronized Pushdown Systems

- call-PDS: *flow-sensitive* and *context-sensitive* analysis
- field-PDF: *flow-sensitive* and *field-sensitive* analysis
- each system has a precision advantage and disadvantage, when compared to the other

- SPDS aim to achieve the *best of both worlds*

Synchronized Pushdown Systems

Definition 4. *For the call-PDS* $\mathcal{P}_{\mathbb{S}} = (\mathbb{V}, \mathbb{S}, \Delta_{\mathbb{S}})$ *and the field-PDS* $\mathcal{P}_{\mathbb{F}} = (\mathbb{V} \times \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}})$, synchronized pushdown systems *are a quintuple* $SPDS = (\mathbb{V}, \mathbb{S}, \mathbb{F} \cup \{\epsilon\}, \Delta_{\mathbb{F}}, \Delta_{\mathbb{S}})$. *A configuration of the SPDS extends from the configuration of each system: A* synchronized configuration *is a triple* $(x, s, f) \in \mathbb{V} \times \mathbb{S}^+ \times \mathbb{F}^*$, *which we denote as* $\langle\!\langle v.f_1 \cdot \ldots \cdot f_m @ s_0^{s_1 \cdot \ldots s_n} \rangle\!\rangle$ *where* $s = s_0 \cdot s_1 \cdot \ldots \cdot s_n$ *and* $f = f_1 \cdot \ldots f_m$. *For synchronized pushdown systems we define the set of all reachable synchronized configurations from a start configuration* $c = \langle\!\langle v.f_1 \cdot \ldots \cdot f_m @ s_0^{s_1 \cdot \ldots s_n} \rangle\!\rangle$ *to be*

$$post_{\mathbb{S}}^{\mathbb{F}}(c) = \{\langle\!\langle w.g @ t_0^{t_1 \cdot \ldots t_n} \rangle\!\rangle \mid \langle\!\langle w @ t_0, g \rangle\!\rangle \in post_{\mathbb{F}}^{*}(\langle\!\langle v @ s_0, f \rangle\!\rangle) \wedge \langle\!\langle w, t \rangle\!\rangle \in post_{\mathbb{S}}^{*}(\langle\!\langle v, s \rangle\!\rangle)\}.$$

Hence, a synchronized configuration c *is accepted if* $\langle\!\langle v, s_0 \cdot \ldots \cdot s_n \rangle\!\rangle \in \mathcal{A}_{\mathbb{S}}$ *and* $\langle\!\langle v @ s_0, f_1 \cdot \ldots \cdot f_m \rangle\!\rangle \in \mathcal{A}_{\mathbb{F}}$ *and* $post_{\mathbb{S}}^{\mathbb{F}}(c)$ *can be represented by the automaton pair* $(\mathcal{A}_{\mathbb{S}}, \mathcal{A}_{\mathbb{F}})$, *which we refer to as* $\mathcal{A}_{\mathbb{S}}^{\mathbb{F}}$.

- the definition of an SPDS is a 5-tuple including a call-PDS and a field-PDS
- reachable and accepted configurations of an SPDS are defined in terms of the reachable and accepted configurations of the embedded call-PDS and field-PDS

SPDS Example

```
51 bar(u, v){
52   v.h = u;
53   w = foo(v);
54   x = w.g;
55   y = x.f;
56 }
```

```
57 foo(p){
58   q.g = p;
59   return q;
60 }
```

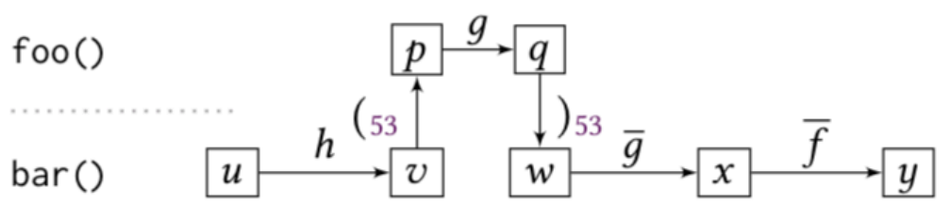


Fig. 5. A code snippet and a labeled graph representation of the code.

- nodes: program variables
- horizontal arrows: field accesses
  - $\overline{\text{f}}$: field f is loaded (pop rule)
  - f: field f is stores (push rule)
- vertical arrows: calls and returns

SPDS Example

```
51 bar(u, v){
52   v.h = u;
53   w = foo(v);
54   x = w.g;
55   y = x.f;
56 }
```

```
57 foo(p){
58   q.g = p;
59   return q;
60 }
```

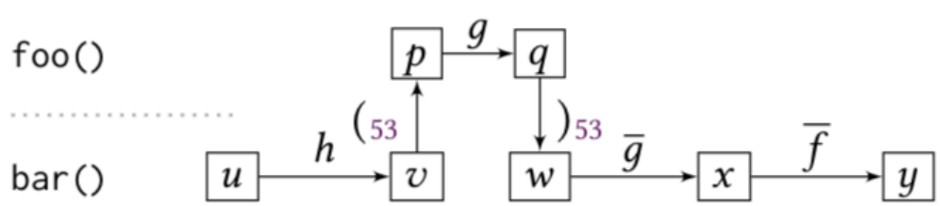


Fig. 5. A code snippet and a labeled graph representation of the code.

Assume a context-, flow- and field-sensitive data-flow analysis to track the object pointed to by u@51. We refer to this abstract object by $o_{\text{u@51}}$. Additionally, assume we want to infer whether $o_{\text{u@51}}$ is accessible by y@55. The actual data-flow is best understood within the graph representation which contains a path from u to y. The labels along this path concatenate to form the sequence (or word) $h \cdot ({}_{53} \cdot g \cdot)_{53} \cdot \overline{g} \cdot \overline{f}$. The parentheses ${}_{53}$ and $)_{53}$ are properly matched. Therefore, the path is realizable in terms of context-sensitivity, i.e., a valid execution path. However, the path is not feasible in terms of field accesses. The field store g is properly matched against the load $\overline{g}$, but the field store h does not match the load of $\overline{f}$. In other words, there is no data-flow connection between u@51 and y@55, which means the latter does not point to $o_{\text{u@51}}$. Synchronized pushdown systems prove the same as $\langle\!\langle y.\epsilon@55^{\epsilon} \rangle\!\rangle \notin post^{\mathbb{F}}_{\mathbb{S}}(\langle\!\langle u.\epsilon@51^{\epsilon} \rangle\!\rangle)$.

What else is in the paper?

- combined automaton? → undecidability
- worst-case complexity analysis
- applications
  - points-to analysis (a non-distributive problem, so other approaches such as IFDS/IDE are not a good fit)
  - typestate analysis
  - backward and demand-driven analyses
- evaluation