

Title	Parameterized Points-to Analysis for Java based on Weighted Pushdown Model Checking
Author(s)	Li, Xin; Ogawa, Mizuhito
Citation	
Issue Date	2006-11-27
Type	Presentation
Text version	publisher
URL	<a href="http://hdl.handle.net/10119/8311">http://hdl.handle.net/10119/8311</a>
Rights	
Description	3rd VERITE : JAIST/TRUST-AIST/CVS joint workshop on VERification TEchnologyでの発表資料, 開催 : 2006年11月27日 ~ 28日, 開催場所 : JAIST 知識科学研究科講義棟・中講義室

# Parameterized Points-to Analysis for Java based on Weighted Pushdown Model Checking

Li Xin, Ogawa Mizuhito

Japan Advanced Institute of Science and Technology

November 27, 2006

# Points-to Analysis for Java

- Purpose
  - Approximate the set of heap objects pointed to by reference variables at runtime
- Why points-to analysis?
  - Essential to many other program analyses and compiler optimizations
  - Headachy issue in program verifications
- Precision and scalability is dominated by
  - Context-sensitivity** calling contexts are distinguished
  - Flow-sensitivity** execution orders are concerned
  - Field-sensitivity** how instance fields are abstracted

# A Running Example

---

```
1:   A x = new A(); ...o1
2:   B y = new B(); ...o2
3:   y.f = new Object(); ...o3
4:   x = y;
   if(...){
5:   z = x.m(y);
   }else{
6:   x.f = new Object(); ...o4
7:   v = y.m(x);
   }
```

*class A*

*m(B a): { return a; }*

*class B inherits class A*

*m(B b): { return b.f; }*

- Declared type strategy
- Virtual method invocation (dynamic binding) at line 5 and 7
- Call-by-value
- Abstract heap objects are associated with codes in blue

---

**Figure:** An Example of Java Code Fragment

# A Running Example

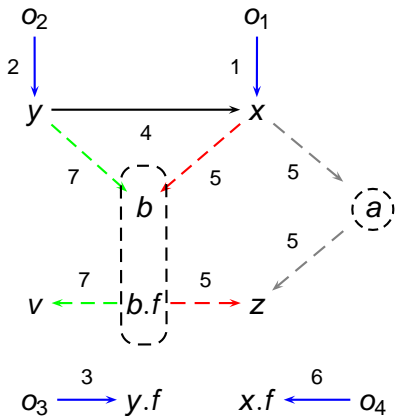
```
1:  A x = new A(); ...o1
2:  B y = new B(); ...o2
3:  y.f = new Object(); ...o3
4:  x = y;
   if(...){
5:  z = x.m(y);
   }else{
6:  x.f = new Object(); ...o4
7:  v = y.m(x);
   }
```

*class A*

*m(B a): { return a; }*

*class B inherits class A*

*m(B b): { return b.f; }*



**Figure:** (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:   A x = new A(); ...O1
2:   B y = new B(); ...O2
3:   y.f = new Object(); ...O3
4:   x = y;
     if(...){
5:   z = x.m(y);
     }else{
6:   x.f = new Object(); ...O4
7:   v = y.m(x);
     }
```

```
class A
```

```
m(B a): { return a; }
```

```
class B inherits class A
```

```
m(B b): { return b.f; }
```



Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:   A x = new A(); ...o1
2:   B y = new B(); ...o2
3:   y.f = new Object(); ...o3
4:   x = y;
    if(...){
5:   z = x.m(y);
    }else{
6:   x.f = new Object(); ...o4
7:   v = y.m(x);
    }
```



```
class A
m(B a): { return a; }
class B inherits class A
m(B b): { return b.f; }
```

Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:  A x = new A(); ...o1
2:  B y = new B(); ...o2
3:  y.f = new Object(); ...o3
4:  x = y;
   if(...){
5:  z = x.m(y);
   }else{
6:  x.f = new Object(); ...o4
7:  v = y.m(x);
   }
```



```
class A
```

```
m(B a): { return a; }
```

```
class B inherits class A
```

```
m(B b): { return b.f; }
```

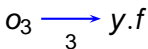


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)



# A Running Example

```
1:   A x = new A(); ...o1
2:   B y = new B(); ...o2
3:   y.f = new Object(); ...o3
4:   x = y;
    if(...){
5:   z = x.m(y);
    }else{
6:   x.f = new Object(); ...o4
7:   v = y.m(x);
    }
```



```
class A
m(B a): { return a; }
class B inherits class A
m(B b): { return b.f; }
```

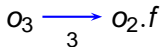


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:  A x = new A(); ...O1
2:  B y = new B(); ...O2
3:  y.f = new Object(); ...O3
4:  x = y;
   if(...){
5:  z = x.m(y);
   }else{
6:  x.f = new Object(); ...O4
7:  v = y.m(x);
   }
```

```
class A
```

```
m(B a): { return a; }
```

```
class B inherits class A
```

```
m(B b): { return b.f; }
```

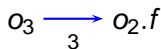
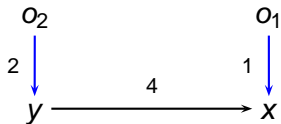


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:   A x = new A(); ...o1
2:   B y = new B(); ...o2
3:   y.f = new Object(); ...o3
4:   x = y;
    if(...){
5:     z = x.m(y);
        }else{
6:     x.f = new Object(); ...o4
7:     v = y.m(x);
        }
```

```
class A
```

```
m(B a): { return a; }
```

```
class B inherits class A
```

```
m(B b): { return b.f; }
```

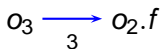
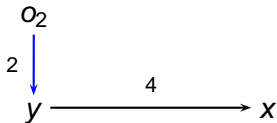


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:   A x = new A(); ...o1
2:   B y = new B(); ...o2
3:   y.f = new Object(); ...o3
4:   x = y;
    if(...){
5:   z = x.m(y);
    }else{
6:   x.f = new Object(); ...o4
7:   v = y.m(x);
    }
```

class A

m(B a): { return a; }

class B inherits class A

m(B b): { return b.f; }

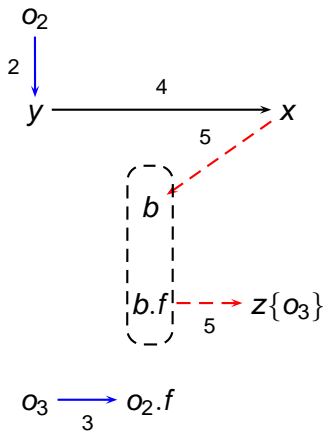


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:  A x = new A(); ...o1
2:  B y = new B(); ...o2
3:  y.f = new Object(); ...o3
4:  x = y;
   if(...){
5:  z = x.m(y);
   }else{
6:  x.f = new Object(); ...o4
7:  v = y.m(x);
   }
```

class A

m(B a): { return a; }

class B inherits class A

m(B b): { return b.f; }

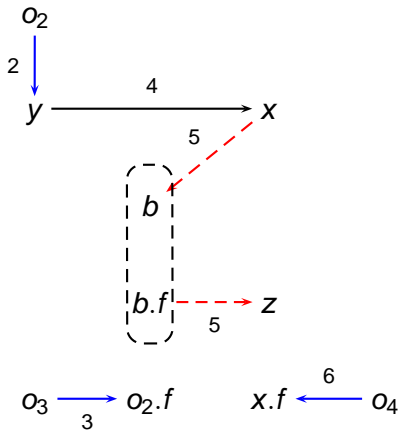


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:  A x = new A(); ...o1
2:  B y = new B(); ...o2
3:  y.f = new Object(); ...o3
4:  x = y;
   if(...){
5:  z = x.m(y);
   }else{
6:  x.f = new Object(); ...o4
7:  v = y.m(x);
   }
```

class A

m(B a): { return a; }

class B inherits class A

m(B b): { return b.f; }

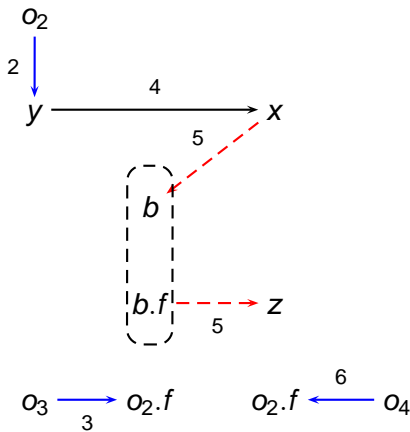


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:   A x = new A(); ...O1
2:   B y = new B(); ...O2
3:   y.f = new Object(); ...O3
4:   x = y;
    if(...){
5:   z = x.m(y);
    }else{
6:   x.f = new Object(); ...O4
7:   v = y.m(x);
    }
```

```
class A
m(B a): { return a; }
class B inherits class A
m(B b): { return b.f; }
```

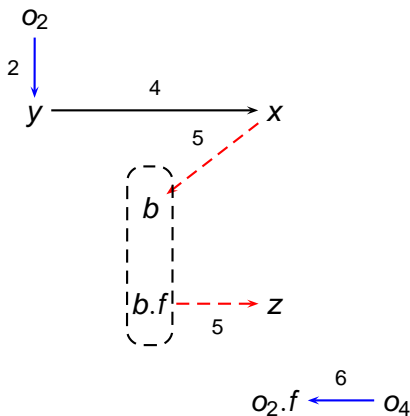


Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)

# A Running Example

```
1:   A x = new A(); ...O1
2:   B y = new B(); ...O2
3:   y.f = new Object(); ...O3
4:   x = y;
   if(...){
5:   z = x.m(y);
   }else{
6:   x.f = new Object(); ...O4
7:   v = y.m(x);
   }
```

```
class A
m(B a): { return a; }
class B inherits class A
m(B b): { return b.f; }
```

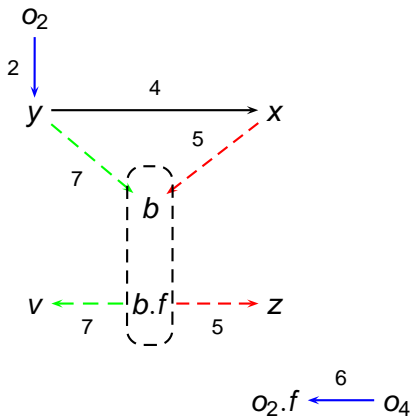


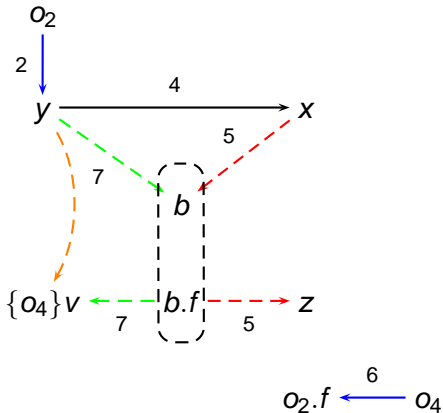
Figure: (a) Example Code Fragment (b) Pointer Assignment Graph of (a)



# A Running Example

```
1:   A x = new A(); ...o1
2:   B y = new B(); ...o2
3:   y.f = new Object(); ...o3
4:   x = y;
   if(...) {
5:     z = x.m(y);
   } else {
6:     x.f = new Object(); ...o4
7:     v = y.m(x);
   }
```

```
class A
m(B a): { return a; }
class B inherits class A
m(B b): { return b.f; }
```



# What does the example tell?

- Points-to analysis and call graph construction are mutually dependent
- Call graph construction
  - **On-the-fly**: constructed during points-to analysis
  - **Ahead-of-time**: a pre-computed approximated call graph is explored for points-to analysis
- Two occasions need points-to information:
  - Call graph construction
  - Instance field abstraction

## Definition

Let  $\mathcal{V}$  and  $\mathcal{O}$  be a set of abstract reference variables and a set of abstract heap objects respectively. A transitive and reflexive **points-to relation** is defined as  $\mapsto: \mathcal{V} \times \mathcal{H}$ , where  $\mathcal{H} = \mathcal{V} \cup \mathcal{O}$ . Its inverse is defined as a **flows-to relation**  $\rightsquigarrow$ .

## Definition

A **pointer assignment graph** is defined as  $G_a = (N_a, E_a)$ , where  $N_a = \mathcal{V} \cup \mathcal{O}$  is a set of nodes, and  $E_a = \rightsquigarrow$  is a set of edges.

## Definition

Let  $\mathcal{F}$  be a set of fields and  $\mathcal{L}$  be a set of local variables. A **field sensitive analysis** abstracts an instance field  $l.f (l \in \mathcal{L}, f \in \mathcal{F})$  as pairs of  $\{(o, f) \mid f \mapsto o\}$ .

# Work Summary

- Program Analysis = Abstract Interpretation + Model Checking
- Context-sensitive points-to analysis algorithms based on weighted pushdown model checking
- Parameterized flow-sensitivity so that the abstraction design is easily tuned
- Variations of points-to analysis algorithms based on the following dimensions:
  - On-the-fly vs. Ahead-of-time call graph construction
  - Lightweight semiring operations vs. Smaller pushdown transitions in the abstraction design
- Evaluation within the Soot framework

# Pushdown Model Checking

- Model: Pushdown System (PDS)
- A PDS + (e.g. Simple) Valuation  
     $\cong$  A Pushdown Automaton  
     $\cong$  Context-free Language
- The intersection of context-free language and regular language is closed (context-free)
- The automata-theoretic approach works

$$\mathcal{M} \models \mathcal{S} \Leftrightarrow L(\mathcal{M}) \cap L(\mathcal{S})^c = \emptyset$$

- Efficient algorithms are developed due to the fact that:  
    “Regular sets of configurations are closed under forward and backward reachability”

# Weighted Pushdown Model Checking

- Associate a weight from a bounded idempotent semiring to each pushdown transition rule
- Solve the **Generalized Pushdown Reachability** (GPR) problem: “Compute weights over paths in a pushdown graph leading from a pushdown configuration to a regular set of pushdown configurations”

## Definition

A **bounded idempotent semiring**  $S$  is a semiring  $(D, \oplus, \otimes, 0, 1)$ , s.t.

- $\oplus$  is idempotent, i.e.  $a \oplus a = a$ .
- A partial order  $\sqsubseteq$  is defined:  $\forall a, b \in D, a \sqsubseteq b$  iff  $a \oplus b = a$ .

That is, no infinite descending chain on weight space is required.

# Application of Pushdown Systems to Program Analyses

- Suitable for modeling interprocedural program analyses
  - Calls and returns are correctly paired (**context-sensitivity**)
  - No limitation on recursion steps (vs. K-CFA)
- Pushdown model checking
  - Model program's data domain
  - Demand finite domain abstraction (by automata-theoretic approach )
- Weighted pushdown model checking
  - Model program's flow function space
  - Demand infinite descending chains on the weight space, but infinite domain abstraction is possible
  - Regular pushdown configurations as an abstraction of calling contexts (**context-sensitivity**)

# Intention Behind the Semiring Design

- **Weight space**  $\Rightarrow$  Flow function space
- A **weight** intends a function to represent how a property is carried at each step of program execution.
- **1**  $\Rightarrow$  Properties keep unchanged by this transition step
- **0**  $\Rightarrow$  The program execution is interrupted by some error
- **$f \otimes g$**   $\Rightarrow$  Function composition of  $g \circ f$
- **$f \oplus g$**   $\Rightarrow$  Conservative approximation over two control flows at their meet
- The optional commutativity of  $\otimes$  facilitates modeling a flow-sensitive analysis



# Abstraction of Heap Memory

## Definition

Let  $\mathcal{O}$  be a set of run-time objects allocated in the heap memory. Functions  $\eta_\tau : \mathcal{O} \rightarrow \mathcal{T}$  and  $\eta_\iota : \mathcal{O} \rightarrow \mathcal{L}$  are defined respectively, where  $\mathcal{T}$  is a set of types (class names) of heap objects, and  $\mathcal{L}$  is a set of memory allocation sites in the program.

## Definition

Let  $\mathcal{O} \subseteq \mathcal{T} \times \mathcal{L} \cup \{\diamond\}$  be a set of abstract heap objects, where  $\diamond$  represents null reference. An **abstraction** on  $\mathcal{O}$  is defined as  $\tilde{\alpha} : \mathcal{O} \rightarrow \mathcal{O}$ , s.t.  
 $\forall o \in \mathcal{O}, \tilde{\alpha}(o) = (\tau, \iota)$ , where  $\tau = \eta_\tau(o) \in \mathcal{T}, \iota = \eta_\iota(o) \in \mathcal{L}$ .

## Remarks:

- $\forall (\tau_i, \iota_i), (\tau_j, \iota_j) \in \mathcal{O}, \iota_i = \iota_j \Rightarrow \tau_i = \tau_j$
- $\forall o_i, o_j \in \mathcal{O}, \tilde{\alpha}(o_i) = \tilde{\alpha}(o_j)$  iff the allocation sites for them are the same.
- An array is approximated with a single element with its base type.

# An Algorithm with Lightweight Semiring Operations

Approaches:

- Reachability analysis on the product of  $G_a$  and  $G_f$ .
- For efficiency, a variation of “exploded supergraph” is explored

## Definition

A **weighted pointer assignment graph** is defined as  $G_l = (N_l, E_l, L_l)$  from  $G_a$ , where  $N_l = \{\Lambda\} \cup \mathcal{V}$  is a set of nodes,  $E_l \subseteq N_l \times L_l \times N_l$  is a set of edges, and  $L_l = \{\lambda x.x\} \cup \{\lambda x.o \mid o \in \mathcal{O}\}$  is a set of labels, such that

- $(v_1, \lambda x.x, v_2) \in E_l$  if  $(v_1, v_2) \in E_a$ ,  $v_1, v_2 \in \mathcal{V}$
- $(\Lambda, \lambda x.o, v) \in E_l$  if  $(o, v) \in E_a$ ,  $o \in \mathcal{O}$ ,  $v \in \mathcal{V}$

Remarks:

- $\Lambda$ : an environment that allocates new heap objects
- Heap objects are labeled on the edges

# The Underlined Model for Model Checking

## Definition

A **weighted flows-to graph**  $G_p = (N_p, E_p, L_p)$  is the product of  $G_l$  and  $G_f$ , where  $N_p = N_l \times N_f$  is a set of nodes,  $E_p \subseteq N_p \times L_p \times N_p$  is a set of edges, and  $L_p = L_l$  is a set of labels.

## Algorithm

Let  $\mathcal{A}[\cdot] : S \rightarrow \mathcal{P}(\rightsquigarrow)$ , and  $N_l = \{\Lambda\} \cup V_g \cup V_l$  ( $V_g \subseteq \mathcal{V}$  represents global variables and  $V_l \subseteq \mathcal{V}$  represents local variables), s.t.  $\forall e_f = (n_1, n_2) \in E_f$

---

$$e_f \in E_i \quad \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in V\} \cup \\ \{((v_1, n_1), \lambda x.x, (v_2, n_2)) \mid (v_1, \lambda x.x, v_2) \in E_l, (v_1, v_2) \in F, v_1 \in \mathcal{V}\} \cup \\ \{((\Lambda, n_1), \lambda x.o, (v, n_2)) \mid (\Lambda, \lambda x.o, v) \in E_l, (c, v) \in F, o \in \mathcal{O}\} \subseteq E_p \\ \text{where } F = \mathcal{A}[\text{StmtOf}(n_2)], V = N_l - \{v \mid (h, v) \in F\}$$

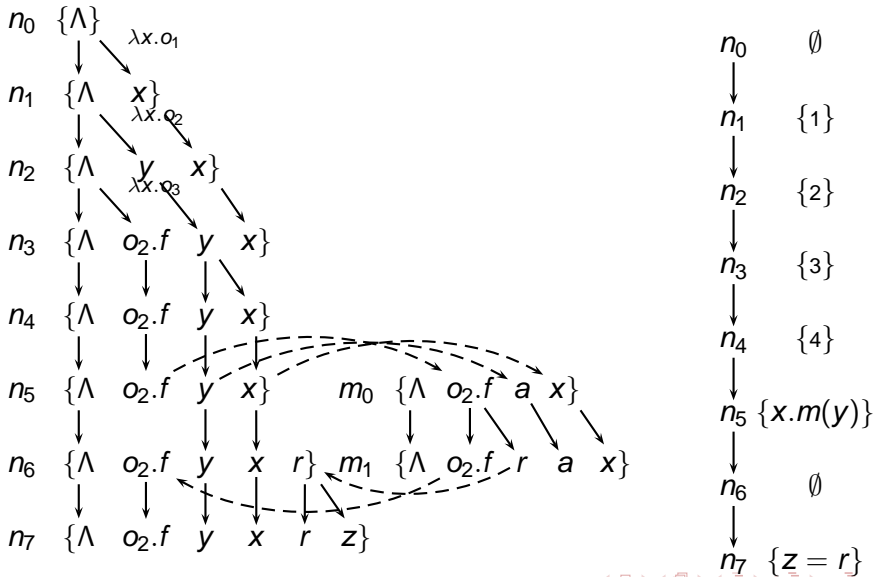
$$e_f \in E_t \quad \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in V_l\} \subseteq E_p$$

$$e_f \in E_c \quad \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in V_g \cup \{\Lambda\}\} \cup \\ \{((h, n_1), \lambda x.x, (v, n_2)) \mid (h, v) \in F\} \subseteq E_p \\ \text{where } F = \mathcal{A}[\text{StmtOf}(n_1)]$$

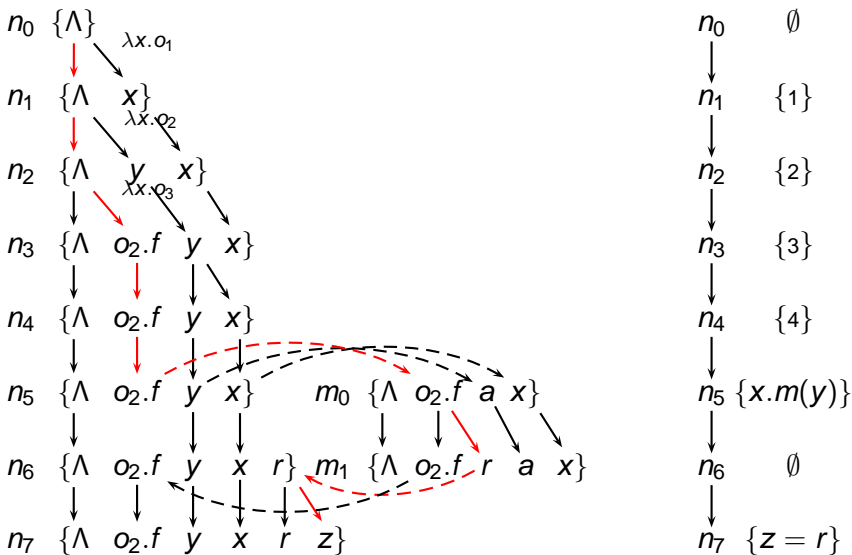
$$e_f \in E_r \quad \{((v, n_1), \lambda x.x, (v, n_2)) \mid v \in V_g \cup \{\Lambda\}\} \subseteq E_p$$

---

# Part of $G_p$ for the Running Example



## Part of $G_p$ for the Running Example



# A Semiring Design

Let  $\mathcal{S} = \mathcal{P}(\mathcal{O})$ ,  $D_1 = \{\lambda x.s \mid s \in \mathcal{S}\}$  and  $D_2 = \{\lambda x.x \cup s \mid s \in \mathcal{S}\}$

## Definition

A bounded idempotent semiring  $S = (D, \oplus, \otimes, \mathbf{0}, \mathbf{1})$  is defined as

- The weight space  $D = D_1 \cup D_2$
- $\mathbf{1}$  is defined as  $\lambda x.x$  and  $\mathbf{0}$  is defined as  $\lambda x.\emptyset$
- The  $\otimes$  operator is defined as

$$\forall d_i, d_j \in D \setminus \{\mathbf{0}, \mathbf{1}\}, d_i \otimes d_j = d_j$$

- The  $\oplus$  operator equals set union  $\cup$ , defined as

$$\forall d_i = \lambda x.s_i, d_j = \lambda x.s_j \in \tilde{D}, d_i \oplus d_j = d_j \oplus d_i = \lambda x.s_i \cup s_j$$

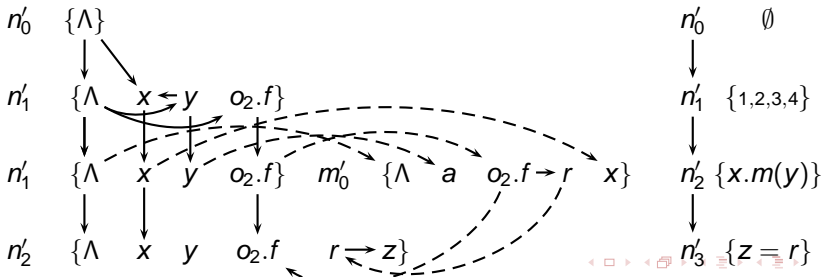
$$\forall d_i = \lambda x.s_i \in \tilde{D}, d_j = \lambda x.x \cup s_j \in \bar{D}, d_i \oplus d_j = d_j \oplus d_i = \lambda x.x \cup s_i \cup s_j$$

$$\forall d_i = \lambda x.x \cup s_i, d_j = \lambda x.x \cup s_j \in \bar{D}, d_i \oplus d_j = d_j \oplus d_i = \lambda x.x \cup s_i \cup s_j$$

Distributivity of  $\otimes$  over  $\oplus$  is easily checked.

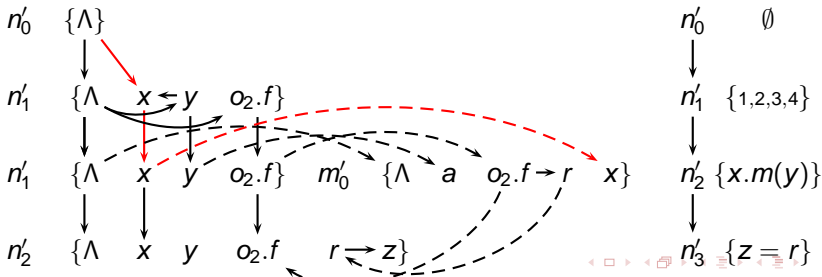
# Parameterized Flow-sensitivity

- Problems:  $G_p$  will explode for large-scale programs
- Solutions:  $G_f$  is firstly shrunk by grouping nodes into blocks
  - One node possibly associated with a set of program statements
  - Each node has an unique entry after shrinking
- Parameterized flow-sensitivity by shrinking
  - Shrinking is NOT arbitrary to keep soundness (loops, branches)
  - An extreme shrinking collapses each method into a single node (**flow-insensitive**)



# Parameterized Flow-sensitivity

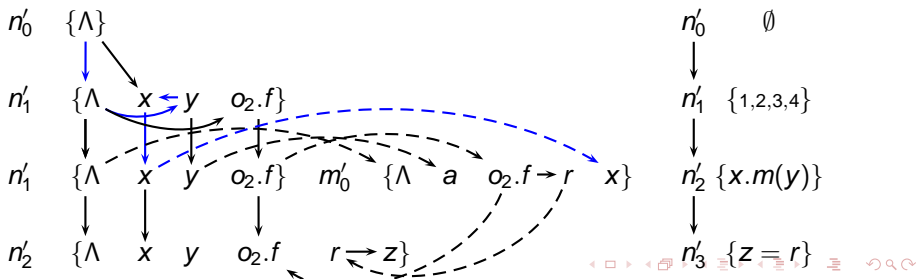
- Problems:  $G_p$  will explode for large-scale programs
- Solutions:  $G_f$  is firstly shrunk by grouping nodes into blocks
  - One node possibly associated with a set of program statements
  - Each node has an unique entry after shrinking
- Parameterized flow-sensitivity by shrinking
  - Shrinking is NOT arbitrary to keep soundness (loops, branches)
  - An extreme shrinking collapses each method into a single node (**flow-insensitive**)





# Parameterized Flow-sensitivity

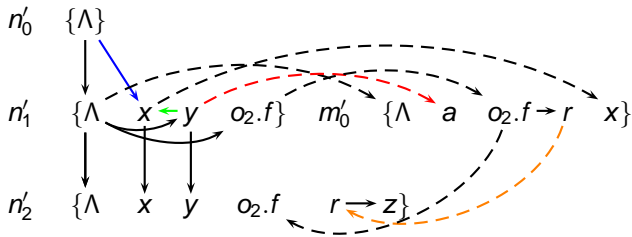
- Problems:  $G_p$  will explode for large-scale programs
- Solutions:  $G_f$  is firstly shrunk by grouping nodes into blocks
  - One node possibly associated with a set of program statements
  - Each node has an unique entry after shrinking
- Parameterized flow-sensitivity by shrinking
  - Shrinking is NOT arbitrary to keep soundness (loops, branches)
  - An extreme shrinking collapses each method into a single node (**flow-insensitive**)



# Encoding to Weighted PDS

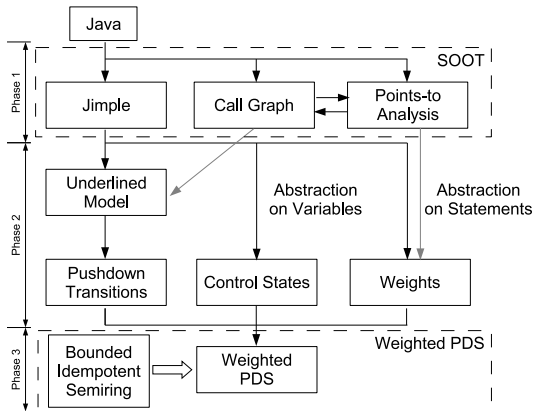
Given a weighted flows-to graph  $G_p = (N_p, E_p, L_p)$ , with  $N_p = \{\Lambda\} \cup \mathcal{V} \times N_f$

- $\{\Lambda\} \cup \mathcal{V} \Rightarrow$  control states
- $N_f \Rightarrow$  stack alphabets
- $E_p \Rightarrow$  pushdown transition rules



$\langle \Lambda, n'_0 \rangle \hookrightarrow \langle x, n'_1 \rangle$   
 $\langle y, n'_1 \rangle \hookrightarrow \langle x, n'_1 \rangle$   
 $\langle y, n'_1 \rangle \hookrightarrow \langle a, m'_0 n'_2 \rangle$   
 $\langle r, n'_1 \rangle \hookrightarrow \langle r, \varepsilon \rangle$

## Evaluation within the SOOT Framework (On-the-fly + Lightweight Semiring Operation)



- Obstacle: restriction from the interaction of soot and weighted PDS library
- Bottleneck: weighted PDS constructed from scratch for each model checking request
- An incremental model construction is promising when possible

software	LOC	MCR	time (s)	TMCR(s)	TMC(s)
jetty	≈3000	1173	632.23	619.57 (97.99%)	468 (75.5%)

# Points-to Analysis with Ahead-of-time Call Graph Construction

- Target: reduce frequent model checking demands
- Approaches
  - A pre-computed approximated call graph is explored
  - Invalid pathes are “removed” during model checking
  - Extra relations to model instance field accesses
- A semiring design with **smaller pushdown transitions**
  - $\hat{\mathcal{S}} \subseteq \mathcal{P}(\mathcal{V} \times \mathcal{H}) = \mathcal{P}(\mapsto)$ , s.t.  $\forall s \in \hat{\mathcal{S}}$

$$\forall (v_1, h_1), (v_2, h_2) \in s, h_1 = h_2 \text{ if } v_1 = v_2$$

- “ $x = y; y = z$ ”  $\Rightarrow$   $\{x \mapsto y, y \mapsto z\}$  instead of  $\{x \mapsto z, y \mapsto z\}$   
 $v_1 \mapsto v_2 \implies v_1 \mapsto v_2'$  (flow-sensitive)
- e.g.  $\{x \mapsto y, y \mapsto o, z \mapsto x\} \implies \{x \mapsto y', y \mapsto o, z \mapsto x'\}$  (i.e. A transitive closure on  $s \in \hat{\mathcal{S}}$  does not make sense )

# A Semiring Design with Smaller Pushdown Transitions

## Definition

A bounded idempotent semiring  $S = (D, \oplus, \otimes, 0, 1)$  is defined as

- The weight space  $D = \mathcal{P}(\mathcal{D})$ , where  $\mathcal{D} = \hat{S} \cup \{\text{ID}\} \setminus \emptyset$
- $0 = \emptyset$  and  $1 = \{\text{ID}\}$
- $\forall w_1, w_2 \in D, w_1 \otimes w_2 = \{d_1 \odot d_2 \mid d_1 \in w_1, d_2 \in w_2\}$ , where

$$d_1 \odot d_2 = \begin{cases} d_1 \text{ (resp. } d_2) & \text{if } d_2 = \text{ID (resp. } d_1 = \text{ID)} \\ f_0(d_1, d_2) \cup f_1(d_1, d_2) & \text{o.w.} \end{cases}$$

$$f_0(d_1, d_2) = d_1 \setminus \{ (v, h_1) \in d_1 \mid \exists h_2 \text{ s.t. } (v, h_2) \in d_2 \}$$

$$f_1(d_1, d_2) = \{ (v_2, h'_2) \mid \forall (v_2, h_2) \in d_2, h'_2 = \begin{cases} h_1 & \text{if } \exists h_1 \text{ s.t. } (v_1, h_1) \in d_1, v_1 = v_2 \\ h_2 & \text{o.w.} \end{cases} \}$$

- $\forall w_1, w_2 \in D, w_1 \oplus w_2 = w_1 \cup w_2$

**Remarks on  $w_1 \otimes w_2$ :** ①  $f_0$ : Relations in  $w_1$  are changed by subsequent operations in  $w_2$  (**flow-sensitive**); ②  $f_1$ : The second components of relations in  $w_2$  are substituted w.r.t  $w_1$ .

# Path Elimination

- $\mathcal{C} \subseteq \mathcal{P}(\mathcal{V} \times \mathcal{T})$ : represent expected types of method receivers
- **type** :  $\mathcal{O} \rightarrow \mathcal{T}$ : get types of abstract heap objects  
**loc** :  $\mathcal{O} \rightarrow \mathcal{L}$ : get allocation sites of abstract heap objects
- $\alpha: \mathcal{C} \times \mathcal{D} \rightarrow \{\mathbf{TRUE}, \mathbf{FALSE}\}$  is introduced as an judgement relation. That is,  $\forall d \in \mathcal{D}, c \in \mathcal{C}, c \alpha d$  iff  $\exists (v, t) \in c$ , and  $(v, o) \in d$ , such that  $t' \times t$ , where  $t' = \mathbf{type}(o)$ .
- $\times: \mathcal{T} \times \mathcal{T} \rightarrow \{\mathbf{TRUE}, \mathbf{FALSE}\}$  defines a relation among classes.  
 $\forall t, t' \in \mathcal{T}, t' \times t$  iff
  - r1.*  $t' \neq t$
  - r2.*
    - a)*  $t'$  does not inherit from  $t$ ; or
    - b)*  $t'$  inherits from  $t$ , but  $t'$  redefines the method to be invoked.
- $\times$  is defined as the reverse of  $\times$ . That is,

$$\forall t, t' \in \mathcal{T}, t' \times t \text{ iff } t' \times t = \mathbf{FALSE}$$

# A Semiring Design with Path Elimination

## Definition

The previous semiring  $S$  is extended to be  $S_e = (D_e, \oplus_e, \otimes_e, 0_e, 1_e)$ , where

- $D_e = \mathcal{P}(\mathbb{D})$ , where  $\mathbb{D} = \{(d, c) \mid d \in \mathcal{D}, c \in \mathcal{C}\}$
- $1_e = \{(\text{ID}, \emptyset)\}$  and  $0_e = \emptyset$
- $\forall w_1, w_2 \in D_e, w_1 \otimes_e w_2 = \{\mathfrak{d}_1 \odot_e \mathfrak{d}_2 \mid \mathfrak{d}_1 \in w_1, \mathfrak{d}_2 \in w_2\}$ , such that  $\forall \mathfrak{d}_1 = (d_1, c_1), \mathfrak{d}_2 = (d_2, c_2) \in \mathbb{D}$ ,

$$\mathfrak{d}_1 \odot_e \mathfrak{d}_2 = \begin{cases} 0_e & \text{if } c_2 \propto d_1 \\ (d_1 \odot d_2, c_1 \uplus c_2) & \text{o.w.} \end{cases}$$

where  $c_1 \uplus c_2 = c_1 \cup f_8(c_2 \setminus c, d_1)$ , and  $c = f_7(c_2, d_1)$ .  $\forall c \in \mathcal{C}, d \in \mathcal{D}$ ,

$$f_7(c, d) = \{(v, t) \in c \mid \exists o \in \mathcal{C}, \text{ s.t. } (v, o) \in d, t' = \text{type}(o), t' \times t\}$$

$$f_8(c, d) = \{(\tilde{v}, t) \mid \forall (v, t) \in c, \tilde{v} = \begin{cases} v' & \text{if } \exists (v, v') \in d, v' \in \mathcal{V} \\ v & \text{o.w.} \end{cases}\}$$

- $\forall w_1, w_2 \in D_e, w_1 \oplus_e w_2 = w_1 \cup w_2$

## Remarks on Path Elimination

- $(v, t) \in c \implies (v', t)$ , where  $c \in \mathcal{C}$
- $c_1 \uplus c_2$ 
  - $f_7$ : remove constraints of  $c_2$  satisfied by  $d_1$
  - $f_8$ : substitute variables of relations in  $c_2$  w.r.t  $d_1$
- Examples
  - $\{(x, o), \emptyset\} \odot_e \{ID, (x, A)\} = 0_e$  if  $(x, A) \propto (x, o)$
  - $\{(x, o)(y, x), \emptyset\} \odot_e \{ID, (x, A)\} = \{(x, o)(y, x), \emptyset\}$  if  $\mathbf{type}(o) \times A$
  - $\{(y, x), \emptyset\} \odot_e \{ID, (y, A)\} = \{(y, x), (x, A)\}$
- Associativity of  $\otimes_e(\odot_e)$  is not obvious but proved



# Model Field Accesses

## Definition

Let  $\mathcal{L}$  be a set of local variables of reference type, and  $\mathcal{F}$  be a set of field names. let  $\hat{\mathcal{H}} = \mathcal{L} \cup \mathcal{O}$ . A **field read relation** is defined as  $\mathbb{R} : \hat{\mathcal{H}} \times \mathcal{F} \times \hat{\mathcal{H}}$ . A **field write relation** is defined as  $\mathbb{W} : \hat{\mathcal{H}} \times \mathcal{F} \times \hat{\mathcal{H}}$ . The points-to relation is redefined as  $\mathbb{P} : \mathcal{L} \times \hat{\mathcal{H}}$ .

## Remarks:

- $(h_1, f, h_2) \in \mathbb{R}$  models the field read access “ $h_2 = h_1.f$ ” ( $h_2 \rightsquigarrow h_1.f$ )
- $(h_1, f, h_2) \in \mathbb{W}$  models the field write access “ $h_1.f = h_2$ ” ( $h_1.f \rightsquigarrow h_2$ )
- $(h_1, f, h_2) \in \mathbb{R} \implies (h'_1, f, h_2)$
- $(h_1, f, h_2) \in \mathbb{W} \implies (h'_1, f, h'_2)$
- A flow-sensitive analysis concerning field accesses seems intractable in this setting
  - $\{h_2 \rightsquigarrow h_1.f\} \otimes \{h_3.f \rightsquigarrow h_2\} \Rightarrow \{h_3.f \rightsquigarrow h_1.f\}$
  - $\{h_2 \rightsquigarrow h_1.f\} \otimes \{h_3 \rightsquigarrow h_2.f\} ?$

# Conclusions

- Weighted pushdown model checking enables a fast design of interprocedural context-sensitive program analyses
- Pushdown systems provides us with handy context-sensitivity for program analyses
- Promising for developing a scalable analysis when the implementation allows
- Some future work
  - Evaluation on the ahead-of-time construction
  - Efficient data structures (like BDD) or other decision procedures could be explored

Thanks!  
li-xin@jaist.ac.jp