Program Analysis based on Weighted Pushdown Model Checking

Li Xin

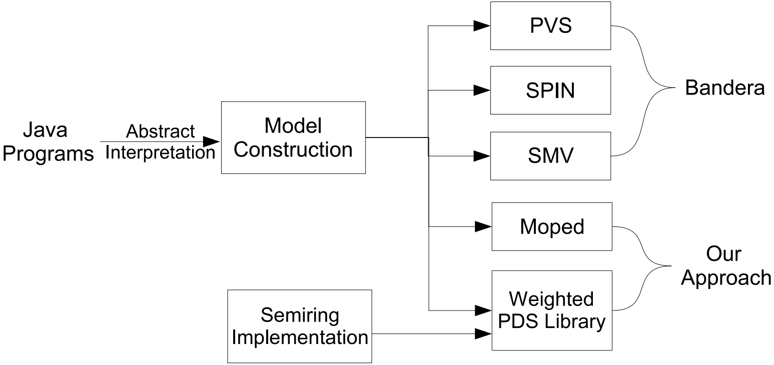
Japan Advanced Institute of Science and Technology

March 5, 2007

Motivations

- Program Analysis = Abstract Interpretation + Model Checking [Steffen91, Schmidt98]
- Advantages of model checking based approach
  - A systematic way by separation of problem abstraction and common implementation efforts
  - Soundness guarantees by A.I. and M.C.
  - “push-button technique” with counterexamples provided as clues for program debugging
- Program analyses based on finite model checking are essentially intraprocedural, i.e. ignore procedure calls
- Pushdown model checking enables interprocedural program analyses for encoding recursions and procedures with the pushdown stack

Our Methodology —
An interprocedural-extension of Bandera-like approach



Java Programs

Abstract Interpretation

Model Construction

PVS

SPIN

SMV

Bandera

Moped

Weighted PDS Library

Our Approach

Semiring Implementation

Finite Model Checking ⇒ Infinite Model Checking

Intraprocedural analysis ⇒ Interprocedural analysis

Our Aims and Work Outline (1)

- **Whether an essential and tough program analysis can be solved by model checking** ?
  - We propose context-sensitive points-to analysis algorithms for Java based on weighted pushdown model checking;
  - Points-to analysis is the basis of interprocedural program analyses for Java, our example is an interprocedural irrelevant code elimination.

Our Aims and Work Outline (2)

- **Whether the primary design choices traditionally concerned can be solved as well** ?
  - Primary design choices are explored in our analysis design
  - The relatively unexploded problem of parametrization is also exploded

	Exploded Supergraph	Interprocedural CFG
On-the-fly	$\surd$	$\surd$
Ahead-of-time	$\surd$	$\surd$
Parameterized flow-sensitivity	Model Reduction	Weight Design Simplification

Pushdown Model Checking

- A pushdown system is a finite transition system with an unbounded stack $P = (Q, \Gamma, \Delta)$, where
  - Q: control locations
  - Γ: stack alphabet
  - Δ: pushdown transitions
- The intersection of context-free language and regular language is closed (context-free)
- The automata-theoretic approach works

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

- Efficient algorithms are developed due to the fact that

Regular sets of configurations are closed under forward and backward reachability.

Weighted Pushdown Model Checking

- A weighted pushdown system $W = (P, S, f)$, where
  - $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring
  - $f : \Delta \rightarrow D$ assigns a value from D to each pushdown transition of P.
- A bounded idempotent semiring S is a semiring $(D, \oplus, \otimes, 0, 1)$ satisfying that
  - $\oplus$ is idempotent, i.e. $a \oplus a = a$
  - $\forall a, b \in D, a \sqsubseteq b$ iff $a \oplus b = a$
- Weighted pushdown model checking is an iterative procedure to find the greatest solution on the weight space
- The analysis is decidable if there exists no infinite descending chains on the weight space

Points-to Analysis for Java

- What is points-to analysis ?

approximate the set of dynamically allocated heap objects pointed to by reference variables at runtime

  - An object that is allocated on the heap memory is either a class instance or an array
  - A reference is a pointer to these objects
    `null` reference refers to no object

- Why points-to analysis?
  - The basis of interprocedural program analysis in Java
  - Quite equivalent to the call graph generation
  - Not a simple matter

Difficulties in Points-to Analysis I

Call graph generation and points-to analysis are mutually dependent

```
1:   A x = new A(); ...$o_1$
2:   B y = new B(); ...$o_2$
3:   y.f = new Object(); ...$o_3$
4:   x = y;
     if(...){
5:   z = x.m(y);
      }else{
6:   x.f = new Object(); ...$o_4$
7:   v = y.m(x);
     }

    class A
    m(B a):  {  return a;  }

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

- Class B inherits Class A with redefining the method m
- o_1, o_2, o_3, o_4 are abstract heap objects respectively associated with allocation sites
- There are method invocations applied on class instances at line 5 and 7
- Reference variables are polymorphic such as line 4

Difficulties in Points-to Analysis I

Call graph generation and points-to analysis are mutually dependent

```
1:      A x = new A(); ...$o_1$
2:      B y = new B(); ...$o_2$
3:      y.f = new Object(); ...$o_3$
4:      x = y;
        if(...){
5:      z = x.m(y);
        }else{
6:      x.f = new Object(); ...$o_4$
7:      v = y.m(x);
        }

    class A
    m(B a):  { return a; }

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

- When a method call is applied to an object, e.g. line 5
- The compiler checks the declared type A of the implicit parameter x and all superclass of A to collect all possible candidates for m
- If m cannot be statically decided, e.g. not a static method, the judgement is postponed to run-time.
- Dynamic binding (or dynamic method dispatch)

Difficulties in Points-to Analysis II

- Due to aliasing, fields could be changed implicitly

```
y = o1;
y.f = o2;
x = y;
x.f = o3;
```

$$\{y \mapsto o_1\}$$
$$\{y \mapsto o_1, o_1.f \mapsto o_2\}$$
$$\{y \mapsto o_1, o_1.f \mapsto o_2, x \mapsto o_1\}$$
$$\{y \mapsto o_1, o_1.f \mapsto o_2, x \mapsto o_1, o_1.f \mapsto o_3\}$$

- A precise points-to analysis, namely field-sensitive analysis, needs to cast aliasing.

Difficulties in Points-to Analysis II

- Due to aliasing, fields could be changed implicitly

`y = o1;`	$\{y \mapsto o_1\}$
`y.f = o2;`	$\{y \mapsto o_1, o_1.f \mapsto o_2\}$
`x = y;`	$\{y \mapsto o_1, o_1.f \mapsto o_2, x \mapsto o_1\}$
`x.f = o3;`	$\{y \mapsto o_1, o_1.f \mapsto o_2, x \mapsto o_1, o_1.f \mapsto o_3\}$

- A precise points-to analysis, namely field-sensitive analysis, needs to cast aliasing.

Difficulties in Points-to Analysis III

- Various infinities exist, such as
  - The number of nestings of array structures, of method invocations, and of field access can be unbounded.
    e.g. $x.f_1.f_2.\cdots.f_n$ is syntactically allowed
  - The number of heap objects allocated can be unbounded.
    e.g. a heap allocation is within a looping structure
- Recursions and nested method invocations are nicely modeled based on pushdown systems.

A Brief Look on Jimple

- Jimple is a typed three-address intermediate representation of Java.
- It is transformed from stack-based Bytecode by hiding stack information and syntactically much simpler.
- Jimple has < 20 operations; Bytecode has > 200.
- From Bytecode to Jimple

iload 0	load variable x_1
iload 1	load variable x_2
iadd	pop and add x_1, x_2 load the sum
istore 1	pop the stack and assign to x_3

$\Rightarrow$

$\$s_1 = x_1;$
$\$s_2 = x_2;$
$\$s_1 = \$s_1 + \$s_2;$
$x_3 = \$s_1;$

Each position in the stack has a corresponding variable, such as $\$s_1$, $\$s_2$.

Heap Objects and References

$\mathcal{O}$	heap objects
$\diamond \in \mathcal{O}$	null reference
$\top \in \mathcal{O}$	a "generic" heap object, i.e. anything
$\mathcal{V}_l$	local variables
$\mathcal{V}_s$	static fields
RefVar	$\mathcal{V}_l \cup \mathcal{V}_s \cup \{\texttt{arg}_k, \texttt{this}, \texttt{ret} \mid k \in \mathbb{N}\}$
$\mathcal{F}$	field names
$\mathcal{V}_{ref}$	$\mathcal{O} \times \mathcal{F}^+ \cup \text{RefVar} \times \mathcal{F}^*$
$\mathcal{V}_{diref}$	$\text{RefVar} \cup \mathcal{O} \times \mathcal{F}$

Note:

- $\mathcal{V}_{ref}$ denotes the references in Java allowing field nesting syntactically; whereas
- $\mathcal{V}_{diref}$ denotes the references specific to Jimple. It is syntactically simpler with direct field access, i.e. $x.f_1.f_2$ is not allowed

Heap Environment

- The data domain of interest is heap environment, i.e. mappings from references to heap objects
- The set of heap environments is defined as

$$\mathsf{Henv} = \{\mathsf{henv} \mid \mathsf{henv} : \mathcal{V}_{diref} \to \mathcal{O}\}$$

- The points-to information is formulated as an heap environment $\mathsf{h}_{\mathbb{V}} \in \mathsf{Henv}$ such that

$$[v_1 \mapsto \mathsf{h}_{\mathbb{V}}(v_1), ..., v_n \mapsto \mathsf{h}_{\mathbb{V}}(v_n)]$$

where $\mathbb{V} = \{v_i \mid 1 \leq i \leq n\} \subseteq \mathcal{V}_{diref}$

Syntax of Heap Environment Transformers

Let $v \in \texttt{RefVar}$, $f \in \mathcal{F}$, $o \in \mathcal{O}$ and $\texttt{henv} \in \text{Henv}$. $\mathbb{F}$ is the set of heap environment transformers, given by ExpFun

---

$$
\begin{array}{lcl}
\text{ExpFun} & ::= & \lambda\texttt{henv}.\ \text{ExpHenv} \\
\text{ExpHenv} & ::= & \texttt{henv} \\
 & | & \text{ExpHenv} \bullet \text{ExpMap} \\
\text{ExpMap} & ::= & [v \mapsto o] \\
 & | & [v_1 \mapsto \text{Expt}, ..., v_n \mapsto \text{Expt}] \\
 & | & [\text{Expf} \mapsto \text{Expt}] \\
\text{Expf} & ::= & \text{Expt}.f \\
\text{Expt} & ::= & o \\
 & | & \texttt{henv}(v) \\
 & | & \texttt{henv}(\text{Expt}.f)
\end{array}
$$

Notes: $\bullet$ can be regarded as the union operation on maps
$\mathbb{F}$ will constitute the weight space

Formalization of Jimple

Jimple Syntax	Formalization
$x = \text{new T}$	$\{\lambda \text{henv}.\text{henv} \bullet [x \mapsto o]\}$ where $o \in \mathcal{O}$ is a fresh abstract heap object
$x = \text{newarray T}[n]$	$\{\lambda \text{henv}.\text{henv} \bullet [x \mapsto o]\}$ where $o \in \mathcal{O}$ is a fresh abstract heap object
$x = \texttt{null}$	$\{\lambda \text{henv}.\text{henv} \bullet [x \mapsto \diamond]\}$
$x = y$	$\{\lambda \text{henv}.\text{henv} \bullet [x \mapsto \text{henv}(y)]\}$
$x = y.f$	$\{\lambda \text{henv}.\text{henv} \bullet [x \mapsto \text{henv}(\text{henv}(y).f)]\}$
$y.f = x$	$\{\lambda \text{henv}.\text{henv} \bullet [\text{henv}(y).f \mapsto \text{henv}(x)]\}$
return x	$\{\lambda \text{henv}.\text{henv} \bullet [\texttt{ret} \mapsto \text{henv}(x)]\}$, when x is a reference variable.

Formalization of Jimple

$x.f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [\mathtt{arg}_1 \mapsto \mathtt{henv}(m_1), ...,$ $\mathtt{arg}_l \mapsto \mathtt{henv}(m_l), \mathtt{this} \mapsto \mathtt{henv}(x)]$ where $m_i (1 \leq i \leq l) \in$ RefVar, $m_j (l \leq j \leq n)$ are variables of primitive type.
$f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [\mathtt{arg}_1 \mapsto \mathtt{henv}(m_1), ...,$ $\mathtt{arg}_l \mapsto \mathtt{henv}(m_l)]\}$, where $m_i (1 \leq i \leq l) \in$ RefVar, $m_j (l \leq j \leq n)$ are variables of primitive type
$z = \mathtt{ret}$	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [z \mapsto \mathtt{henv}(\mathtt{ret})]\}$
$x :=$ @this: T	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [x \mapsto \mathtt{henv}(\mathtt{this})]\}$
$x :=$ @parameter$_k$: T	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [x \mapsto \mathtt{henv}(\mathtt{arg}_k)]\}$

Function Composition and Evaluation

- The composition of heap environment transformers is, for $\mathrm{exph}_1, \mathrm{exph}_2 \in \mathrm{ExpHenv}$,

$$\begin{array}{ll} & (\lambda \mathrm{henv}.\ \mathrm{exph}_2) \circ (\lambda \mathrm{henv}.\ \mathrm{exph}_1) \\ =_\eta & \lambda \mathrm{h}.\ (\lambda \mathrm{henv}.\ \mathrm{exph}_2 (\lambda \mathrm{henv}.\ \mathrm{exph}_1 \mathrm{h})) \\ =_\beta & \lambda \mathrm{h}.\ \mathrm{exph}_2[\mathrm{henv} := \mathrm{exph}_1[\mathrm{henv} := \mathrm{h}]] \end{array}$$

- To get the points-to information amounts to the evaluation of heap environments.

Let $v \in \mathrm{RefVar}$, $o \in \mathcal{O}$, $w \in \mathcal{F}^*$, and $\mathrm{henv} \in \mathrm{Henv}$. The evaluation of $v.w$, $o.w$ by henv is defined as

$$eval(\mathrm{henv}, v.w) = \begin{cases} \mathrm{henv}(v) & \text{if } w = \epsilon \\ (\mathrm{henv}\ \ldots\ (\mathrm{henv}(v).f_1)\ \ldots\ f_n) & \text{if } w = f_1 \cdots f_n \in \mathcal{F}^* \end{cases}$$

$$eval(\mathrm{henv}, o.w) = \begin{cases} o & \text{if } w = \epsilon \\ (\mathrm{henv}\ \ldots\ (\mathrm{henv}(o.f_1))\ \ldots\ f_n) & \text{if } w = f_1 \cdots f_n \in \mathcal{F}^* \end{cases}$$

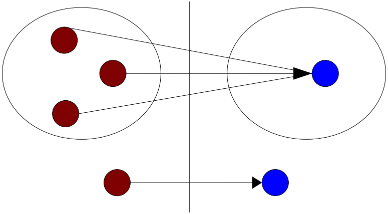
Basic Ideas of the Analysis

- The weight space is a powerset construction: $\mathcal{P}(\mathbb{F})$
  $\otimes$: the reverse of function composition
  $\oplus$: combines the analysis from different pathes by set union.
- The analysis result is a set of heap environment transformers.
- Each transformer corresponds to changes on heap environment along a possible program run.
- The analysis demands $\mathbb{F}$ to be finite! Abstractions are needed to remove the various infinities.

Abstractions

An unique abstract heap object is associated with a heap allocation site.

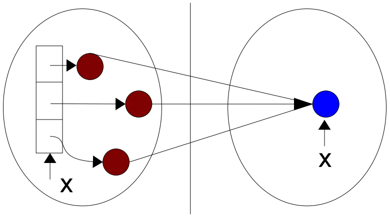
```
i = 3;
while(i>0)
{
    A x = new A();
    x.f();
    i--;
}
    A x = new A();
```



Abstractions

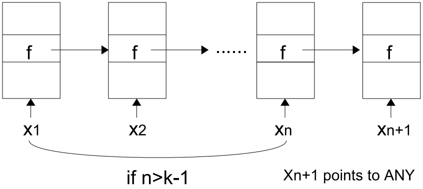
An array is abstracted to a single heap allocation with the type of an array element.

```
i = 3;
A[] x = new A[3];
while(i>0)
{
    A[i] = new A();
    i--;
}
```



Abstractions

A bound k is set on the field nesting if necessary.



For henv $\in$ Henv, $v \in$ RefVar, $o \in \mathcal{O}$, $w \in \mathcal{F}^*$,

$$\text{henv}(\mathit{eval}(\text{henv}, v.w).f) = \begin{cases} \mathit{eval}(\text{henv}, v.w') & \text{if } |w| + 1 \le k \\ & \text{and } w' = w \cdot f \\ \top & \text{otherwise} \end{cases}$$

An Ahead-of-time Analysis (Interprocedural CFG)

- The analysis starts with an imprecise call graph; invalid path removal is designers' responsibility.
- Propose an ahead-of-time analysis with automatic invalid path removal.
- The weight space is extended by pairing a set of path constraints $\texttt{PathCons} \subseteq \mathcal{V}_{ref} \times \mathcal{T}$

By a path constraint $(v.w, t) \in \texttt{PathCons}$, we mean a call edge demands the actual type of $eval(\texttt{henv}, v.w)$ to satisfy with t.

- An invalid path is excluded from the analysis result when the current data flow conflicts with the type constraints.

A Type Relation to Formalize Dynamic Binding

- Let $\mathcal{T}$ be a finite set of types (i.e. class names) of abstract heap objects, and $\mathtt{type} : \mathcal{O} \to \mathcal{T}$ be the function that gets the type of an abstract object.

$$\mathtt{any} = \mathtt{type}(\top)$$

$$\mathtt{none} = \mathtt{type}(\diamond)$$

- A relation on types is defined as

For $t, t' \in \mathcal{T} \setminus \{\mathtt{any}, \mathtt{none}\}$, t' conflicts with t wrt some method iff

- $t' \neq t$, and
- either t' does not inherit from t, or t' inherits from t but t' redefines the method.

Otherwise, we say t' satisfies with t. Furthermore,

- t satisfies with $\mathtt{any}$, for each t in $\mathcal{T}$;
- $\mathtt{none}$ conflicts with t, for each t in $\mathcal{T}$.

Abstraction for the Ahead-of-time Analysis

The essential part of the modified abstraction relates to fields and dynamic method dispatch is shown as follows

Syntax	Formalization
$x = y.f$	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [x \mapsto \mathtt{henv}(\mathtt{henv}(y).f)]\}$
$y.f = x$	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [\mathtt{henv}(y).f \mapsto \mathtt{henv}(x)]\}$
$x.f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$\{\lambda \mathtt{henv}.\mathtt{henv} \bullet [\mathtt{arg}_1 \mapsto \mathtt{henv}(m_1), ..., \mathtt{arg}_l \mapsto \mathtt{henv}(m_l), \mathtt{this} \mapsto \mathtt{henv}(x)]\}$ where $m_i(1 \leq i \leq l)$ are reference variables, $m_j(l \leq j \leq n)$ are variables of primitive type.

Abstraction for the Ahead-of-time Analysis

The essential part of the modified abstraction relates to fields and dynamic method dispatch is shown as follows

Syntax	Formalization
$x = y.f$	$\{\lambda \text{henv}.\text{henv} \bullet [x \mapsto \text{henv}(\text{henv}(y).f)], \emptyset\}$
$y.f = x$	$\{\lambda \text{henv}.\text{henv} \bullet [\text{henv}(y).f \mapsto \text{henv}(x)], \emptyset\}$
$x.f(m_1, ..., m_l, m_{l+1}, ...m_n)$	$\{\lambda \text{henv}.\text{henv} \bullet [\texttt{arg}_1 \mapsto \text{henv}(m_1), ...,$ $\texttt{arg}_l \mapsto \text{henv}(m_l), \texttt{this} \mapsto \text{henv}(x)]$ $, \{(x, \epsilon, t)\}\}$ where $m_i (1 \leq i \leq l)$ are reference variables, $m_j (l \leq j \leq n)$ are variables of primitive type. A call edge is generated according to the imprecise initial call graph.

The Basic Ideas of the Design

- A known satisfied constraint will not contribute to the analysis result

1. $x = o$; $e_1 : \{\lambda \text{henv}.\text{henv} \bullet [x \mapsto o], \emptyset\}$
2. $x.m()$; $e_2 : \{\lambda \text{henv}.\text{henv}, \{(x, \epsilon, \text{A})\}\}$

$$eval(e_1(\text{henv}), x) = o$$

if `type`(o) satisfies with A or is any, $e_1 \otimes e_2 = \{\lambda \text{henv}.\text{henv}, \emptyset\}$

if `type`(o) conflicts with A, $e_1 \otimes e_2 = 0$

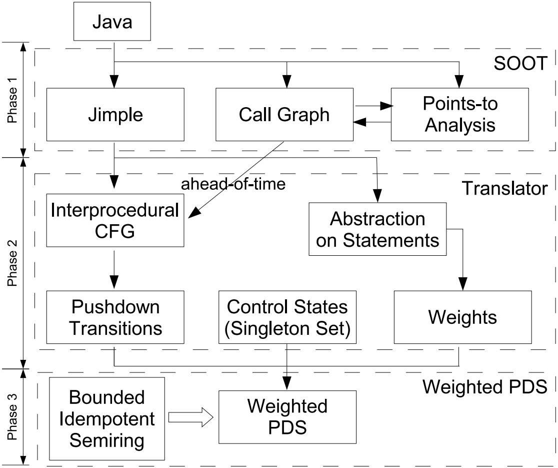
- The judgement on path constraints can be pending due to aliasing, and the aliasing will be traced backward.

1. $x = y$; $e_1 : \{\lambda \text{henv}.\text{henv} \bullet [x \mapsto \text{henv}(y)], \emptyset\}$
2. $x.m()$; $e_2 : \{\lambda \text{henv}.\text{henv}, \{(x, \epsilon, \text{A})\}\}$

$$eval(e_1(\text{henv}), x) = \text{henv}(y)$$

$$e_1 \otimes e_2 = \{\lambda \text{henv}.\text{henv}, \{y, \epsilon, \text{A}\}\}$$

A Prototype Implementation (CFG)



Java

Phase 1

SOOT

Jimple

Call Graph

Points-to Analysis

Phase 2

ahead-of-time

Translator

Interprocedural CFG

Abstraction on Statements

Pushdown Transitions

Control States (Singleton Set)

Weights

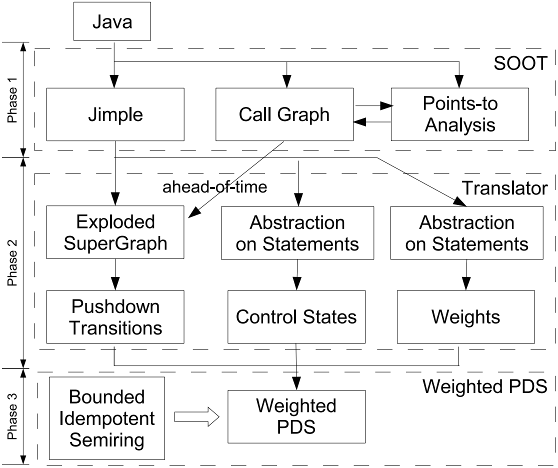
Phase 3

Weighted PDS

Bounded Idempotent Semiring

Weighted PDS

A Prototype Implementation (Exploded Supergraph)



Java

SOOT

Phase 1

Jimple

Call Graph

Points-to Analysis

Translator

Phase 2

ahead-of-time

Exploded SuperGraph

Abstraction on Statements

Abstraction on Statements

Pushdown Transitions

Control States

Weights

Weighted PDS

Phase 3

Bounded Idempotent Semiring

Weighted PDS

Related Works

- Constant propagation [SCP05] and critical path findings [ESOP06] by weighted pushdown model checking
- Traditionally, points-to analysis is mostly based on constraints/equations solving, e.g. $x = y \Rightarrow pt(y) \subseteq pt(x)$.
- The first scalable context-sensitive points-to analysis for Java [PLDI04] copies each method contexts and collapses loops; The analysis scales by using BDD as the underlined data structure.
- A recent context-sensitive points-to analysis for Java explores CFL-reachability for context-sensitivity [PLDI06]; The scalability is obtained by the demand-driven manner.

Conclusions

- Propose points-to analysis algorithms based on weighted pushdown model checking; primary dimensions traditionally concerned are explored
- Our formulation also clearly explains how each factor plays a role in the points-to analysis
- More implementation efforts can be handed over to model checkers as the fixpoint calculator, e.g. the ahead-of-time analysis with automatic path removal can be done in one run model checking.
- Since pushdown model checking with more than two stacks is undecidable, abstraction cannot be avoided. To cover concurrent behaviors will be our future work.

Thanks!

li-xin@jaist.ac.jp