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

Finite Model Checking \(\Rightarrow\) Infinite Model Checking
Intraprocedural analysis \(\Rightarrow\) 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

<table>
<thead>
<tr>
<th></th>
<th>Exploded Supergraph</th>
<th>Interprocedural CFG</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-the-fly</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Ahead-of-time</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Parameterized flow-sensitivity</td>
<td>Model Reduction</td>
<td>Weight Design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simplification</td>
</tr>
</tbody>
</table>
Pushdown Model Checking

- A pushdown system is a finite transition system with an unbounded stack $P = (Q, \Gamma, \Delta)$, where
  - $Q$: control locations
  - $\Gamma$: stack alphabet
  - $\Delta$: pushdown transitions

- The intersection of context-free language and regular language is closed (context-free)

- The automata-theoretic approach works

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

- Efficient algorithms are developed due to the fact that regular sets of configurations are closed under forward and backward reachability.
Weighted Pushdown Model Checking

- 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 \texttt{x} = \texttt{new A(); ...o}_1
2:   B \texttt{y} = \texttt{new B(); ...o}_2
3:   \texttt{y.f} = \texttt{new Object(); ...o}_3
4:   \texttt{x} = \texttt{y};
    \hspace{1em} \texttt{if(...)}\{
5:   \hspace{1em} \texttt{z} = \texttt{x.m(y);}
    \hspace{1em} \} \texttt{else}\{
6:   \hspace{1em} \texttt{x.f} = \texttt{new Object(); ...o}_4
7:   \hspace{1em} \texttt{v} = \texttt{y.m(x);}
    \}\}

\begin{itemize}
  \item Class B inherits Class A with redefining the method m
  \item \(o_1, o_2, o_3, o_4\) are abstract heap objects respectively associated with allocation sites
  \item There are method invocations applied on class instances at line 5 and 7
  \item Reference variables are polymorphic such as line 4
\end{itemize}
Difficulties in Points-to Analysis I

Call graph generation and points-to analysis are mutually dependent

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; }

► 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)
Due to aliasing, fields could be changed implicitly

\[
\begin{align*}
y &= o_1; \\
y.f &= o_2; \\
x &= y; \\
x.f &= o_3;
\end{align*}
\]

\[
\begin{align*}
\{ y &\mapsto o_1 \} \\
\{ y &\mapsto o_1, o_1.f &\mapsto o_2 \} \\
\{ y &\mapsto o_1, o_1.f &\mapsto o_2, x &\mapsto o_1 \} \\
\{ y &\mapsto o_1, o_1.f &\mapsto o_2, x &\mapsto o_1, o_1.f &\mapsto o_3 \}
\end{align*}
\]

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

\[
\begin{align*}
y &= o_1; & \{y \mapsto o_1\} \\
y.f &= o_2; & \{y \mapsto o_1, o_1.f \mapsto o_2\} \\
x &= y; & \{y \mapsto o_1, o_1.f \mapsto o_2, x \mapsto o_1\} \\
x.f &= o_3; & \{y \mapsto o_1, o_1.f \mapsto o_2, x \mapsto o_1, o_1.f \mapsto o_3\}
\end{align*}
\]

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

\[
\begin{align*}
\text{iadd} & \quad \text{pop and add } x_1, x_2 \\
\text{istore} 1 & \quad \text{pop the stack and assign to } x_3
\end{align*}
\]

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

\[
\begin{align*}
&s_1 = x_1; \\
&s_2 = x_2; \\
&s_1 = s_1 + s_2; \\
&x_3 = s_1;
\end{align*}
\]
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 | $\mathcal{V}_l \cup \mathcal{V}_s \cup \{\text{arg}_k, \text{this}, \text{ret} \mid k \in \mathbb{N}\}$ |

| $\mathcal{F}$ | field names |
| $\mathcal{V}_{\text{ref}}$ | $\mathcal{O} \times \mathcal{F}^+ \cup \text{RefVar} \times \mathcal{F}^*$ |
| $\mathcal{V}_{\text{deref}}$ | $\text{RefVar} \cup \mathcal{O} \times \mathcal{F}$ |

Note:

- $\mathcal{V}_{\text{ref}}$ denotes the references in Java allowing field nesting syntactically; whereas

- $\mathcal{V}_{\text{deref}}$ denotes the references specific to Jimple. It is syntactically simpler with direct field access, i.e. $x.f_1.f_2$ is not allowed
The data domain of interest is heap environment, i.e. mappings from references to heap objects.

The set of heap environments is defined as

\[ \text{Henv} = \{ h_{\text{env}} \mid h_{\text{env}} : \mathcal{V}_{\text{dirref}} \to \mathcal{O} \} \]

The points-to information is formulated as an heap environment \( h_{\mathcal{V}} \in \text{Henv} \) such that

\[ [v_1 \mapsto h_{\mathcal{V}}(v_1), \ldots, v_n \mapsto h_{\mathcal{V}}(v_n)] \]

where \( \mathcal{V} = \{ v_i \mid 1 \leq i \leq n \} \subseteq \mathcal{V}_{\text{dirref}} \)
Syntax of Heap Environment Transformers

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

\[
\begin{align*}
\text{ExpFun} & ::= \lambda \text{henv}. \text{ExpHenv} \\
\text{ExpHenv} & ::= \text{henv} \\
 & \mid \text{ExpHenv} \circ \text{ExpMap} \\
\text{ExpMap} & ::= [v \mapsto o] \\
 & \mid [v_1 \mapsto \text{Expt}, \ldots, v_n \mapsto \text{Expt}] \\
 & \mid [\text{Expf} \mapsto \text{Expt}] \\
\text{Expf} & ::= \text{Expt}.f \\
\text{Expt} & ::= o \\
 & \mid \text{henv}(v) \\
 & \mid \text{henv}(\text{Expt}.f)
\end{align*}
\]

Notes: $\circ$ can be regarded as the union operation on maps

$F$ will constitute the weight space
### Formalization of Jimple

<table>
<thead>
<tr>
<th>Jimple Syntax</th>
<th>Formalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x = \text{new } T)</td>
<td>({\lambda \text{henv}.\text{henv} \bullet [x \mapsto o]}) when (o \in \mathcal{O}) is a fresh abstract heap object</td>
</tr>
<tr>
<td>(x = \text{newarray } T[n])</td>
<td>({\lambda \text{henv}.\text{henv} \bullet [x \mapsto o]}) where (o \in \mathcal{O}) is a fresh abstract heap object</td>
</tr>
<tr>
<td>(x = \text{null})</td>
<td>({\lambda \text{henv}.\text{henv} \bullet [x \mapsto \Diamond]})</td>
</tr>
<tr>
<td>(x = y)</td>
<td>({\lambda \text{henv}.\text{henv} \bullet [x \mapsto \text{henv}(y)]})</td>
</tr>
<tr>
<td>(x = y.f)</td>
<td>({\lambda \text{henv}.\text{henv} \bullet [x \mapsto \text{henv}(\text{henv}(y).f)]})</td>
</tr>
<tr>
<td>(y.f = x)</td>
<td>({\lambda \text{henv}.\text{henv} \bullet [\text{henv}(y).f \mapsto \text{henv}(x)]})</td>
</tr>
<tr>
<td>return (x)</td>
<td>({\lambda \text{henv}.\text{henv} \bullet [\text{ret} \mapsto \text{henv}(x)]}), when (x) is a reference variable.</td>
</tr>
</tbody>
</table>
## Formalization of Jimple

<table>
<thead>
<tr>
<th>Expression</th>
<th>Formalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x.f(m_1, ..., m_l, m_{l+1}, ...m_n)$</td>
<td>${ \lambda \text{henv}. \text{henv} \bullet [\text{arg}_1 \mapsto \text{henv}(m_1), ..., \text{arg}_l \mapsto \text{henv}(m_l), \text{this} \mapsto \text{henv}(x)] }$</td>
</tr>
<tr>
<td>$f(m_1, ..., m_l, m_{l+1}, ...m_n)$</td>
<td>${ \lambda \text{henv}. \text{henv} \bullet [\text{arg}_1 \mapsto \text{henv}(m_1), ..., \text{arg}_l \mapsto \text{henv}(m_l)] }$</td>
</tr>
<tr>
<td>$z = \text{ret}$</td>
<td>${ \lambda \text{henv}. \text{henv} \bullet [z \mapsto \text{henv}(\text{ret})] }$</td>
</tr>
<tr>
<td>$x := @\text{this}: T$</td>
<td>${ \lambda \text{henv}. \text{henv} \bullet [x \mapsto \text{henv}(\text{this})] }$</td>
</tr>
<tr>
<td>$x := @\text{parameter}_k : T$</td>
<td>${ \lambda \text{henv}. \text{henv} \bullet [x \mapsto \text{henv}(\text{arg}_k)] }$</td>
</tr>
</tbody>
</table>

Where $m_i (1 \leq i \leq l) \in \text{RefVar}$, $m_j (l \leq j \leq n)$ are variables of primitive type.
Program Analysis based on Weighted Pushdown Model Checking

Li Xin

Motivations
Introductions
Formalizations of Java Semantics on Pointer Operations
Context-sensitive Points-to Analysis
Conclusions

Function Composition and Evaluation

The composition of heap environment transformers is, for \( \text{exph}_1, \text{exph}_2 \in \text{ExpHenv} \),

\[
(\lambda \text{henv}. \text{exph}_2) \circ (\lambda \text{henv}. \text{exph}_1)
=_{\eta} \lambda h. (\lambda \text{henv}. \text{exph}_2(\lambda \text{henv}. \text{exph}_1 h))
=_{\beta} \lambda h. \text{exph}_2[henv := \text{exph}_1[henv := h]]
\]

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

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

\[
\text{eval}(\text{henv}, v.w) = \begin{cases} 
\text{henv}(v) & \text{if } w = \epsilon \\
(\text{henv} \ ... \ (\text{henv}(v).f_1) \ ... \ f_n) & \text{if } w = f_1 \cdot \cdot \cdot f_n \in \mathcal{F}^*
\end{cases}
\]

\[
\text{eval}(\text{henv}, o.w) = \begin{cases} 
o & \text{if } w = \epsilon \\
(\text{henv} \ ... \ (\text{henv}(o.f_1)) \ ... \ f_n) & \text{if } w = f_1 \cdot \cdot \cdot f_n \in \mathcal{F}^*
\end{cases}
\]
Basic Ideas of the Analysis

- The weight space is a powerset construction: $\mathcal{P}(F)$
  - $\otimes$: the reverse of function composition
  - $\oplus$: combines the analysis from different paths 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 $F$ to be finite! Abstractions are needed to remove the various infinities.
An unique abstract heap object is associated with a heap allocation site.

```java
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.

```java
i = 3;
A[] x = new A[3];
while (i>0)
{
    A[i] = new A();
i--;
}
```
A bound \( k \) is set on the field nesting if necessary.

\[
\text{henv} \in \text{Henv}, \, v \in \text{RefVar}, \, o \in \mathcal{O}, \, w \in \mathcal{F}^*, \quad \text{for } n > k - 1
\]

\[
\text{henv}(\text{eval}(\text{henv}, v \cdot w).f) = \begin{cases} 
\text{eval}(\text{henv}, v \cdot w') & \text{if } |w| + 1 \leq k \\
\top & \text{and } w' = w \cdot f 
\end{cases}
\]

otherwise
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 $\text{PathCons} \subseteq \mathcal{V}_{\text{ref}} \times \mathcal{T}$.

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

- An invalid path is excluded from the analysis result when the current data flow conflicts with the type constraints.
Let $\mathcal{T}$ be a finite set of types (i.e. class names) of abstract heap objects, and $\text{type} : \mathcal{O} \rightarrow \mathcal{T}$ be the function that gets the type of an abstract object.

\[
\text{any} = \text{type}(\top) \\
\text{none} = \text{type}(\diamond)
\]

A relation on types is defined as

For $t, t' \in \mathcal{T} \setminus \{\text{any}, \text{none}\}$, $t'$ 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 any, for each $t$ in $\mathcal{T}$;
- 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

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Formalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x = y.f]</td>
<td>[{\lambda henv. henv \bullet [x \mapsto henv(henv(y).f)]}]</td>
</tr>
<tr>
<td>[y.f = x]</td>
<td>[{\lambda henv. henv \bullet [henv(y).f \mapsto henv(x)]}]</td>
</tr>
<tr>
<td>[x.f(m_1, \ldots, m_l, m_{l+1}, \ldots, m_n)]</td>
<td>[{\lambda henv. henv \bullet [\text{arg}_1 \mapsto henv(m_1), \ldots, \text{arg}_l \mapsto henv(m_l), \text{this} \mapsto henv(x)]}]</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Formalization</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y.f )</td>
<td>{λ\text{henv}.\text{henv} \cdot [x \mapsto \text{henv}(\text{henv}(y).f)], \emptyset }</td>
</tr>
<tr>
<td>( y.f = x )</td>
<td>{λ\text{henv}.\text{henv} \cdot [\text{henv}(y).f \mapsto \text{henv}(x)], \emptyset }</td>
</tr>
<tr>
<td>( x.f(m_1, ..., m_l, m_{l+1}, ..., m_n) )</td>
<td>{λ\text{henv}.\text{henv} \cdot [\arg_1 \mapsto \text{henv}(m_1), ..., \arg_l \mapsto \text{henv}(m_l), \text{this} \mapsto \text{henv}(x)], {(x, \epsilon, t)} }</td>
</tr>
</tbody>
</table>

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

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

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

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

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

\[ e_1 \otimes e_2 = \{ \lambda henv. henv, \{y, \epsilon, A\}\} \]
A Prototype Implementation (CFG)

- **Phase 1**
  - **Java**
  - **Jimple**
  - **Call Graph**
  - **Points-to Analysis**

- **Phase 2**
  - **Interprocedural CFG**
  - **Pushdown Transitions**
  - **Control States (Singleton Set)**
  - **Abstraction on Statements**

- **Phase 3**
  - **Bounded Idempotent Semiring**
  - **Weighted PDS**
  - **Weights**

- **Translator**
  - **SOOT**
A Prototype Implementation (Exploded Supergraph)
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