Field-Sensitive Unreachability and Non-Cyclicity Analysis

Enrico Scapin and Fausto Spoto

Dipartimento di Informatica - University of Verona (Italy)

BYTECODE/ETAPS 2013

Static Analysis

Definition

Static analysis consists in building compile-time techniques in order to prove properties of programs before actually running them.

Shape Analyses try to understand how the program execution manipulates the heap.

e.g.,

- *sharing analysis* determines if two variables might be bound to overlapping data structures.
- *reachability analysis* determines if exists a path in memory that links two variables.
- *cyclicity analysis* determines if a variable is bound to a cyclical data structure.

State of the Art

Reachability and Cyclicity, state of the art:

- Stefano Rossignoli and Fausto Spoto, *"Detecting non-cyclicity by abstract compilation into boolean functions".* In: VMCAI'06
- Samir Genaim and Damiano Zanardini, *"Reachability-based Acyclicity Analysis by Abstract Interpretation".* In: CoRR'12
- Đurica Nikolić and Fausto Spoto, *"Reachability Analysis of Program Varibles".* In: IJCAR'12

`x.next=y;` This assignment makes x *cyclical* if and only if y reaches x.

State of the Art

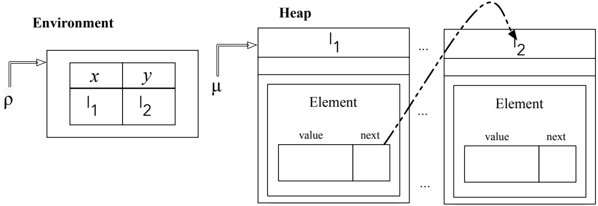
Reachability and Cyclicity, state of the art:

- Stefano Rossignoli and Fausto Spoto, *"Detecting non-cyclicity by abstract compilation into boolean functions"*. In: VMCAI'06
- Samir Genaim and Damiano Zanardini, *"Reachability-based Acyclicity Analysis by Abstract Interpretation"*. In: CoRR'12
- Đurica Nikolić and Fausto Spoto, *"Reachability Analysis of Program Varibles"*. In: IJCAR'12

`x.next=y;` This assignment makes x *cyclical* if and only if y reaches x.

We defined a state as $\sigma = \langle \rho, \mu \rangle$, where:

- ρ maps variables to locations;
- μ binds locations to objects.



State of the Art

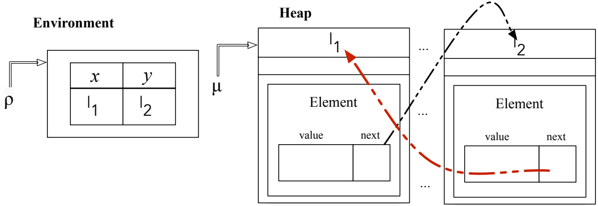
Reachability and Cyclicity, state of the art:

- Stefano Rossignoli and Fausto Spoto, *"Detecting non-cyclicity by abstract compilation into boolean functions".* In: VMCAI'06
- Samir Genaim and Damiano Zanardini, *"Reachability-based Acyclicity Analysis by Abstract Interpretation".* In: CoRR'12
- Đurica Nikolić and Fausto Spoto, *"Reachability Analysis of Program Varibles".* In: IJCAR'12

`x.next=y;` This assignment makes x *cyclical* if and only if y reaches x.

We defined a state as $\sigma = \langle \rho, \mu \rangle$, where:

- ρ maps variables to locations;
- μ binds locations to objects.



Scenario

Given the following Java instructions,

```
while(x!=null)
        x=x.next;
```

Does the loop halt?

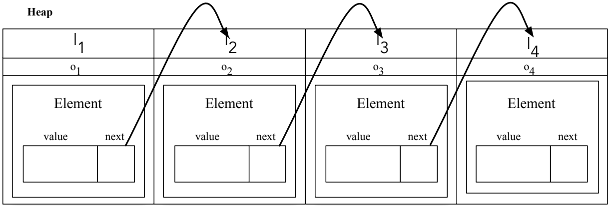
Scenario

Given the following Java instructions,

```
while(x!=null)
        x=x.next;
```

Does the loop halt?

- Assuming $\rho(x) = l_1$ before starting the loop.



The loop terminates in 3 iterations!

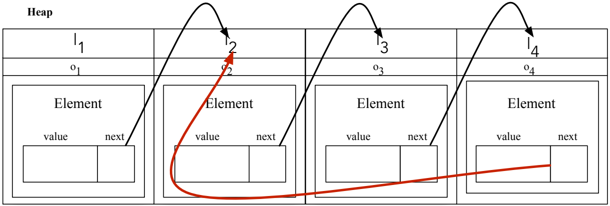
Scenario

Given the following Java instructions,

```
while(x!=null)
          x=x.next;
```

Does the loop halt?

- Assuming $\rho(x) = l_1$ before starting the loop.



The loop does not terminate!

It depends on the cyclicity of variable x.

Can we refine them?

Yes, by developing a field-sensitive analysis!

```
while(x!=null)                x.next=y;
        x=x.next;
```

Goal

For each program point, maintain a set of static fields F such that a program property holds.

Can we refine them?

Yes, by developing a field-sensitive analysis!

```
while(x!=null)                    x.next=y;
        x=x.next;
```

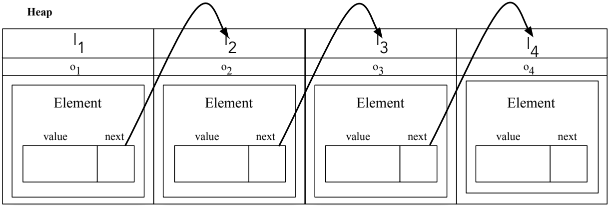
Goal

For each program point, maintain a set of static fields F such that a program property holds.

We introduce the concept of path $\mathcal{P}$ as a tuple of fields linking two locations inside the heap μ.

e.g., $\ell_1 \rightsquigarrow^{\mathcal{P}}_{\mu} \ell_4$

with $\mathcal{P} = \langle El.next, El.next, El.next \rangle$



Field-sensitive properties

Let

- $\mathcal{F}$: set of all fields;
- $\mathsf{L}_\sigma(x)$: set of all locations reachable from x.

Unreachability for each path from x to y in state σ, the fields in F are not part of that path.

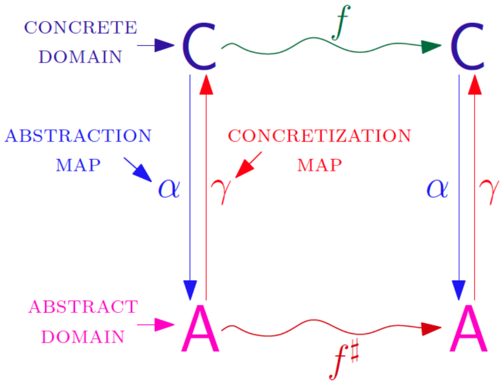
$$\forall \mathcal{P} \subseteq \mathcal{F} \left(x \rightsquigarrow_\sigma^{\mathcal{P}} y \Longrightarrow \mathcal{P} \cap F = \emptyset \right) \equiv x \not\rightsquigarrow_\sigma^{F} y$$

Non-cyclicity for each cycle reachable from x in state σ, the fields in F are not part of the cycle.

$$\forall \ell \in \mathsf{L}_\sigma(x), \forall \mathcal{P} \subseteq \mathcal{F} \left(\ell \rightsquigarrow_\mu^{\mathcal{P}} \ell \Rightarrow \mathcal{P} \cap F = \emptyset \right) \equiv x \rightsquigarrow_\sigma^{\not\circlearrowleft F}$$

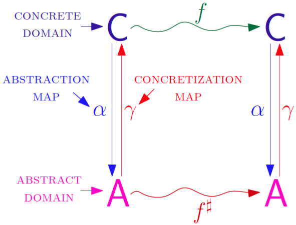
Abstract Interpretation

In order to make our analysis computable, we use the general framework of Abstract Interpretation.



Concrete and Abstract Domains

- Σ - set of all states
- V - set of all variables
- $\mathcal{F}$ - set of all program fields



- Concrete domain: $\mathsf{C} = \wp(\Sigma)$
- Abstract domain: $\mathsf{A} = \wp(V \times V \times \wp(\mathcal{F})) \cup \wp(V \times \wp(\mathcal{F}))$
- Concretization map $\gamma \colon \mathsf{A} \to \mathsf{C}$

$$\gamma(I \in A) = \left\{ \sigma \in \Sigma \;\middle|\; \begin{array}{l} \left(\forall a \not\leftrightarrow^{F} b \in I, \exists F' \subseteq \mathcal{F}.\, a \not\leftrightarrow^{F'}_{\sigma} b \wedge F \subseteq F'\right) \wedge \\ \left(\forall c \rightsquigarrow\!\not\circlearrowleft^{F} \in I, \exists F' \subseteq \mathcal{F}.\, c \rightsquigarrow\!\not\circlearrowleft^{F'}_{\sigma} \wedge F \subseteq F'\right) \end{array} \right\}$$

Our properties are under-approximated by the information in I.

Methodology

1. Program Under Analysis

```
class Element{
 private Object value;
 private Element prec, next;

 public Element(Object value){
  this.value=value;
 }
 public Element(Object value, Element prec){
  this.value=value;
  this.prec=prec;
  prec.next=this;
 }
}
public class MWexample{
 public static void main(String[] args){
  Element top = new Element(new Integer(0));
  for(int i=1;i<=3;i++)
   top = new Element(new Integer(i),top);
 }
}
```

Methodology

1 Program Under Analysis

```
class Element{
 private Object value;
 private Element prec, next;

 public Element(Object value){
  this.value=value;
 }
 public Element(Object value, Element prec){
  this.value=value;
  this.prec=prec;
  prec.next=this;
 }
}
public class MWexample{
 public static void main(String[] args){
  Element top = new Element(new Integer(0));
  for(int i=1;i<=3;i++)
   top = new Element(new Integer(i),top);
 }
}
```

2 Java Bytecode

```
invokespecial #1 <Object/<init>()V>
aload_0
aload_1
putfield #2 Element.value: Object
aload_0
aload_2
putfield #3 Element.prec: Element
aload_2
aload_0
putfield #4 Element.next: Element
return
```

Methodology

1 Program Under Analysis

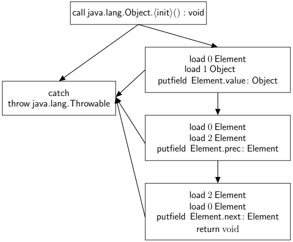
```
class Element{
 private Object value;
 private Element prec, next;

 public Element(Object value){
  this.value=value;
 }
 public Element(Object value, Element prec){
  this.value=value;
  this.prec=prec;
  prec.next=this;
 }
}
public class MWexample{
 public static void main(String[] args){
  Element top = new Element(new Integer(0));
  for(int i=1;i<=3;i++)
   top = new Element(new Integer(i),top);
 }
}
```

2 Java Bytecode

```
invokespecial #1 <Object/<init>()V>
aload_0
aload_1
putfield #2 Element.value: Object
aload_0
aload_2
putfield #3 Element.prec: Element
aload_2
aload_0
putfield #4 Element.next: Element
return
```

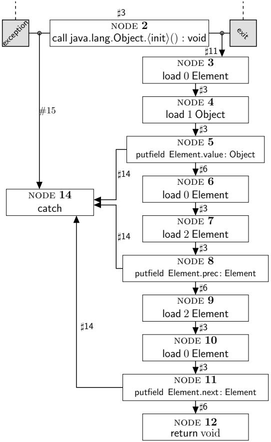
3 Control Flow Graph



Constraint Based Static Analysis

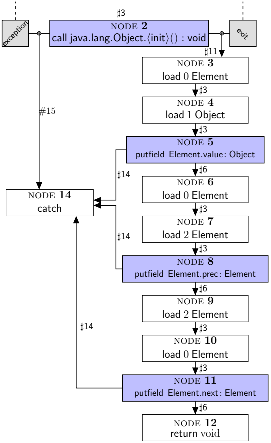
From the Control Flow Graph we build the Abstract Constraint Graph

- **Nodes** represent bytecode instructions.
- **Arcs** represent the abstract semantics.
- Each node is decorated with an abstract set I.
- Each arc is decorated with a propagation rule.
- **Propagation Rules $\sharp i$**
  - defined for each type of arc, depending on its sources;
  - state how the information in each node is propagated.



Propagation Rules

- Their definitions can became complex whenever they exploit other static analyses.
- The unreachability and non-cyclicity information is propagated along the arcs of the ACG until reaching a fix-point.
- It exists since they are all monotonic functions.
- The fix-point is the maximal solution of the ACG with respect to the partial order $\supseteq$.



Example: putfield $\kappa.f$:t

ins: $\Sigma \rightarrow \Sigma'$

local vars stack vars

$$\lambda\langle\underbrace{\langle l, \overbrace{s_{j-1} :: s_{j-2} :: s}^{\text{stack vars}}\rangle}_{\rho}, \mu\rangle \;.\; \langle\langle l, s\rangle, \mu[(\mu(s_{j-2}).\phi)(f) \mapsto s_{j-1}]\rangle$$

$$s_{j-2} \overset{f}{\rightsquigarrow} s_{j-1}$$

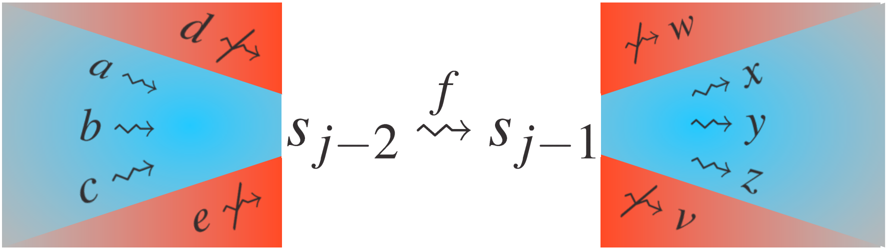
It changes the paths between locations!

How to correctly propagate the information w.r.t this instruction?

KEY IDEA: exploit the result of the *possible* reachability analysis.

$$\langle x, y\rangle \notin \mathcal{MR}_\tau \Longrightarrow x \not\rightsquigarrow y$$

Example: putfield $\kappa.f$:t (cont.)



e.g., field-sensitive unreachability

- for each $d \not\rightsquigarrow^F w$ such that $d \not\rightsquigarrow s_{j-2} \vee s_{j-1} \not\rightsquigarrow w$, F does not change after the putfield node.
- for each $a \not\rightsquigarrow^F x$ such that $\langle a, s_{j-2}\rangle, \langle s_{j-1}, x\rangle \in \mathcal{MR}_\tau$, F probably changes:

  for sure, after the putfield, F does not contain the field **f**!

Conclusions

1. Build an under-approximated analysis to state two field-sensitive properties.

Conclusions

1. Build an under-approximated analysis to state two field-sensitive properties.
2. Exploit the *abstract interpretation* framework to prove its correctness.

Conclusions

1. Build an under-approximated analysis to state two field-sensitive properties.
2. Exploit the *abstract interpretation* framework to prove its correctness.
   - each propagation rule $\Pi^{\sharp i}$ correctly approximates the set of states obtained by the correspondent instruction $\mathsf{ins}^{\sharp i}$ execution:

$$\text{for each } I \in \mathsf{A}, \qquad \mathsf{ins}(\gamma(I)) \subseteq \gamma(\Pi(I))$$

Conclusions

1. Build an under-approximated analysis to state two field-sensitive properties.
2. Exploit the *abstract interpretation* framework to prove its correctness.
   - each propagation rule $\Pi^{\sharp i}$ correctly approximates the set of states obtained by the correspondent instruction $\mathsf{ins}^{\sharp i}$ execution:
     for each $I \in A$, $\quad \mathsf{ins}(\gamma(I)) \subseteq \gamma(\Pi(I))$
   - the analysis correctly approximates the semantics of the program with respect to the two properties defined:
     let $\Rightarrow^* \langle \boxed{\mathsf{ins}} \parallel \sigma \rangle$ be an execution and I_{ins} the approx information,
     $\sigma \in \gamma(I_{\mathsf{ins}})$

Conclusions

1. Build an under-approximated analysis to state two field-sensitive properties.
2. Exploit the *abstract interpretation* framework to prove its correctness.
   - each propagation rule $\Pi^{\sharp i}$ correctly approximates the set of states obtained by the correspondent instruction $\mathsf{ins}^{\sharp i}$ execution:
     for each $I \in \mathsf{A}$, $\quad \mathsf{ins}(\gamma(I)) \subseteq \gamma(\Pi(I))$
   - the analysis correctly approximates the semantics of the program with respect to the two properties defined:
     let $\Rightarrow^* \langle \boxed{\mathsf{ins}} \parallel \sigma \rangle$ be an execution and I_{ins} the approx information,
     $\sigma \in \gamma(I_{\mathsf{ins}})$

Future works: implementing this analysis in Julia Tool to improve the precision of its **termination checker**.

Thank You

Thank You!