Inference of Field Initialization

Fausto Spoto and Michael D. Ernst

University of Verona, Italy & University of Washington, USA

Honolulu, May 25, 2011, ICSE

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Execution trace

```
NullPointerException
 at Hashtable.put()
 at MyWindow.windowInit()
 at JWindow.<init>()
 at MyWindow.<init>()
 at MyWindow.main()
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Execution trace

```
NullPointerException
 at Hashtable.put()
 at MyWindow.windowInit()
 at JWindow.<init>()
 at MyWindow.<init>()
 at MyWindow.main()
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Execution trace

```
NullPointerException
 at Hashtable.put()
 at MyWindow.windowInit()
 at JWindow.<init>()
 at MyWindow.<init>()
 at MyWindow.main()
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Execution trace

```
NullPointerException
 at Hashtable.put()
 at MyWindow.windowInit()
 at JWindow.<init>()
 at MyWindow.<init>()
 at MyWindow.main()
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Execution trace

```
NullPointerException
 at Hashtable.put()
 at MyWindow.windowInit()
 at JWindow.<init>()
 at MyWindow.<init>()
 at MyWindow.main()
```

Clever tracking of windows by name (from a real story)

```
public class MyWindow extends JWindow {
 private final String name; // never null
 private final static Map<String, MyWindow>
   map = new Hashtable<String, MyWindow>();
 public MyWindow(String name) {
   this.name = name;
   setVisible(true);
 }
 public static void main(String[] args) {
  new MyWindow("first");
  new MyWindow("second");
 }
 @Override
 protected void windowInit() {
  super.windowInit();
  map.put(name, this);
 }
}
```

Execution trace

```
NullPointerException
 at Hashtable.put()
 at MyWindow.windowInit()
 at JWindow.<init>()
 at MyWindow.<init>()
 at MyWindow.main()
```

The notion of *rawness*

Definition (Raw object)

An object is *raw wrt. fields F* iff some field in *F* is not initialized.

Example

Variable `this` is raw inside `windowInit` wrt. field `name`:

```
@Override @Raw
protected void windowInit() {
  ... map.put(name, this);
}
```

Hence there is no guarantee that `name` is already initialized *there*.

Note: assigning `null` into a field makes it initialized

The notion of *rawness*

Definition (Raw object)

An object is *raw wrt. fields F* iff some field in *F* is not initialized.

Example

Variable `this` is raw inside `windowInit` wrt. field `name`:

```
@Override @Raw
protected void windowInit() {
  ... map.put(name, this);
}
```

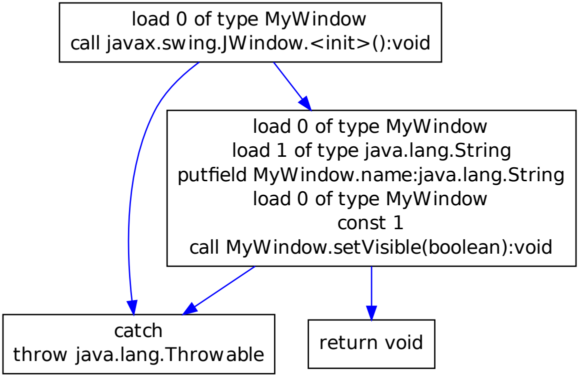
Hence there is no guarantee that `name` is already initialized *there*.

Note: assigning `null` into a field makes it initialized

Our goal: an automatic inference for initialization

1. define a concrete operational semantics of a Java-like language
2. define a constraint-based abstract interpretation of that semantics
3. prove them related by a correctness relation
4. use our abstract interpretation as an inference engine for initialization
5. measure its precision by using nullness analysis
   - but any other analysis could be used instead

Java bytecode as a graph of basic blocks

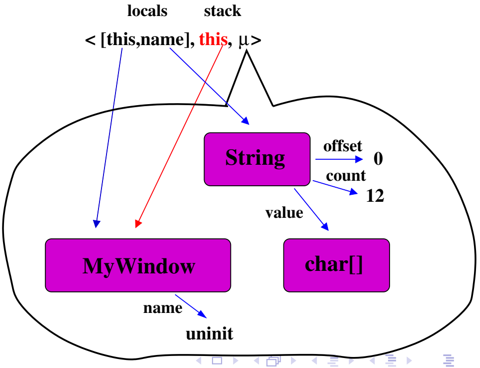


- a graph for each constructor or method
- explicit, inferred types
- resolved field and method references (through class analysis)
- explicit exception handlers

A state is a triple $\langle l \parallel s \parallel \mu \rangle$ of *local variables*, *operand stack* and *heap*, that binds locations to *objects*.

An object o belongs to class $o.\kappa \in \mathbb{K}$ and maps field identifiers f into $o.f$, which can be a value or `uninit`.

```
this.name = name;
        ⇓
load 0 of type MyWindow
load 1 of type String
putfield MyWindow.name
```

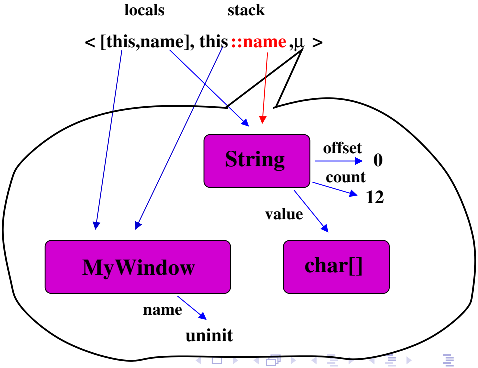


Bytecodes work over states

A state is a triple $\langle l \parallel s \parallel \mu \rangle$ of *local variables*, *operand stack* and *heap*, that binds locations to *objects*.

An object o belongs to class $o.\kappa \in \mathbb{K}$ and maps field identifiers f into $o.f$, which can be a value or `uninit`.

```
this.name = name;
          ⇓
load 0 of type MyWindow
load 1 of type String
putfield MyWindow.name
```

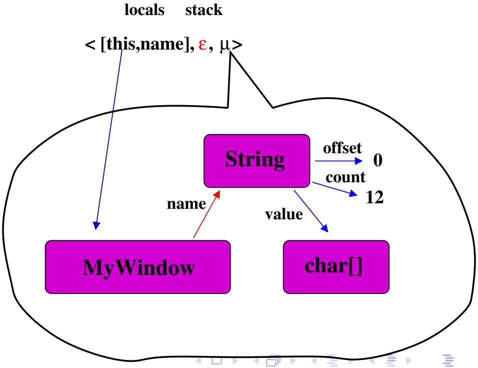


Bytecodes work over states

A state is a triple $\langle l \parallel s \parallel \mu \rangle$ of *local variables*, *operand stack* and *heap*, that binds locations to *objects*.

An object o belongs to class $o.\kappa \in \mathbb{K}$ and maps field identifiers f into $o.f$, which can be a value or `uninit`.

```
this.name = name;
          ⇓
load 0 of type MyWindow
load 1 of type String
putfield MyWindow.name
```



Formalisation of state transformations

`load i of type t`

$$\langle l \parallel s \parallel \mu \rangle \Rightarrow \langle l \parallel l[i] :: s \parallel \mu \rangle$$

`putfield` $\kappa.f$

$$\langle l \parallel top :: rec :: s \parallel \mu \rangle \Rightarrow \langle l \parallel s \parallel \mu[\mu(rec).f \mapsto top] \rangle \quad \text{if } rec \neq \texttt{null}$$

`new` κ (ℓ is fresh, all reference fields in o contain `uninit`)

$$\lambda \langle l \parallel s \parallel \mu \rangle \Rightarrow \langle l \parallel \ell :: s \parallel \mu[\ell \mapsto o] \rangle \quad \text{if there is enough memory}$$

We define an operational semantics over an activation record of states (see the paper for details).

From concrete to abstract

Concrete

We have a concrete notion of states and of state transformers

- concrete states store locations, integers, everything
- we have seen an execution of three bytecodes in sequence

Abstract

We are going to define abstract states and state transformers

- abstract states store the sets of uninitialized fields, only
- we will see the same execution over this abstraction

Abstract Interpretation

- we will define this abstraction systematically
- and link concrete and abstract with a correctness result

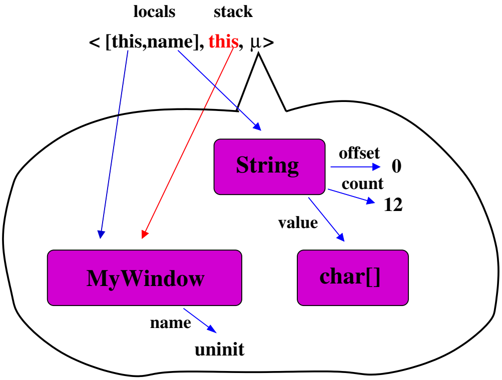
Our abstraction of the concrete states

Abstraction of $\langle [l_0 \dots l_{i-1}] \parallel s_{j-1} :: \cdots :: s_0 \parallel \mu \rangle$

- variable-wise: $\langle [l_0^\alpha \dots l_p^\alpha] \parallel s_q^\alpha :: \cdots :: s_0^\alpha \parallel f_1^\alpha \dots f_r^\alpha \rangle$
- $l_k^\alpha = \begin{cases} \emptyset & \text{if } l_k \in \mathbb{Z} \cup \{\texttt{null}\} \\ \{f \mid \mu(l_k).f = \texttt{uninit}\} & \text{if } l_k \in \mathbb{L} \end{cases}$
- $s_k^\alpha = \begin{cases} \emptyset & \text{if } s_k \in \mathbb{Z} \cup \{\texttt{null}\} \\ \{f \mid \mu(s_k).f = \texttt{uninit}\} & \text{if } s_k \in \mathbb{L} \end{cases}$
- $f_k^\alpha = \begin{cases} \emptyset \\ \quad \text{if } f_k \text{ has primitive type} \\ \{f \mid \text{there exists } \ell \in \mathbb{L} \text{ s.t. } \mu(\mu(\ell).f_k).f = \texttt{uninit}\} \\ \quad \text{if } f_k \text{ has reference type} \end{cases}$

Example of abstract execution

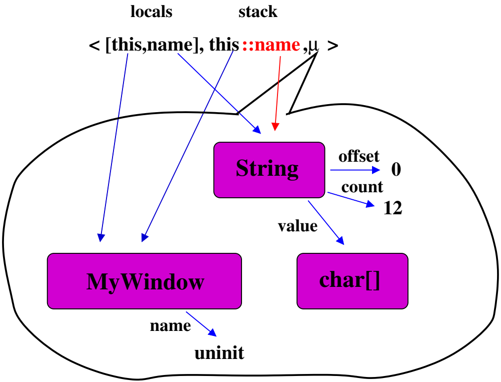
```
load 0 of type MyWindow
```



$$\langle \underbrace{[\{\texttt{name}\}, \emptyset]}_{\text{locals}} \parallel \underbrace{\{\texttt{name}\}}_{\text{stack}} \parallel \underbrace{\underbrace{\emptyset}_{\texttt{name}}, \underbrace{\emptyset}_{\texttt{value}}}_{\text{fields}} \rangle$$

Example of abstract execution

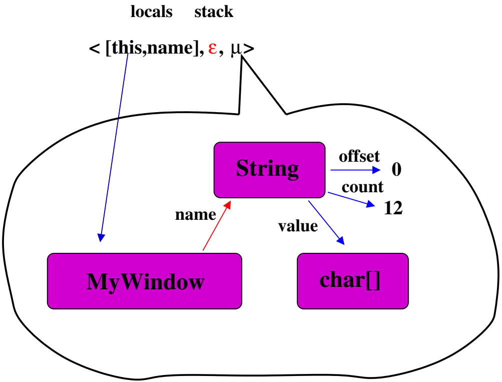
`load 1 of type String`



$$\langle \underbrace{[\{\texttt{name}\},\emptyset]}_{\text{locals}} \parallel \underbrace{\{\texttt{name}\},\emptyset}_{\text{stack}} \parallel \underbrace{\underbrace{\emptyset}_{\texttt{name}},\underbrace{\emptyset}_{\texttt{value}}}_{\text{fields}} \rangle$$

Example of abstract execution

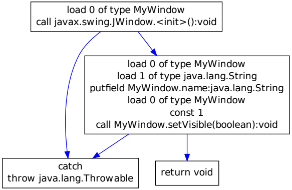
`putfield MyWindow.name`

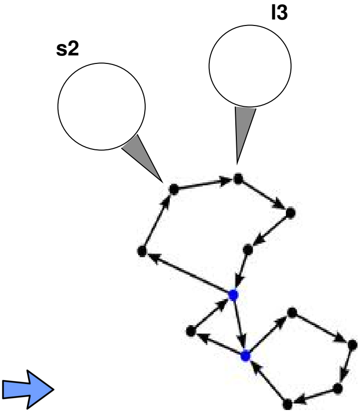


$$\langle \underbrace{[\emptyset, \emptyset]}_{\text{locals}} \parallel \underbrace{\epsilon}_{\text{stack}} \parallel \underbrace{\underbrace{\emptyset}_{\texttt{name}}, \underbrace{\emptyset}_{\texttt{value}}}_{\text{fields}} \rangle$$

From program code to an abstract graph

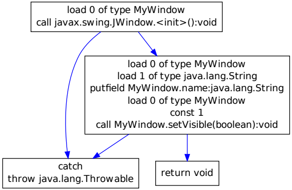
nodes stand for local variables, stack elements, fields...

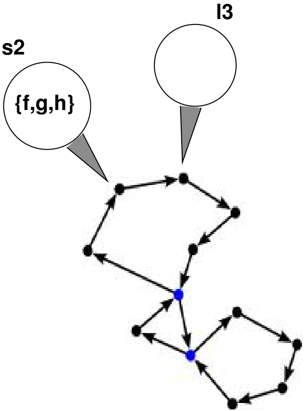




From program code to an abstract graph

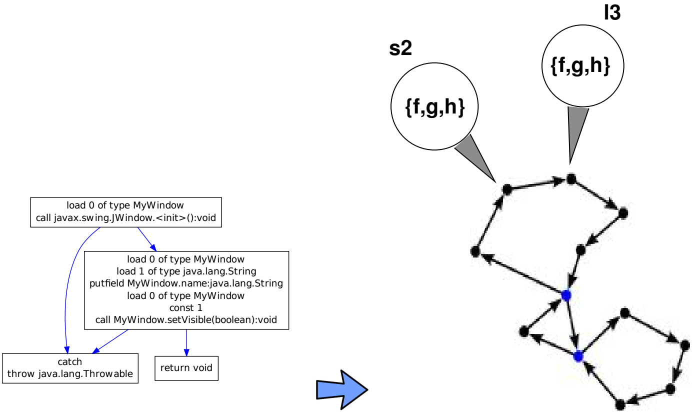
nodes contain a set of non-initialized fields





From program code to an abstract graph

arcs propagate those sets from source to sink (set inclusion)

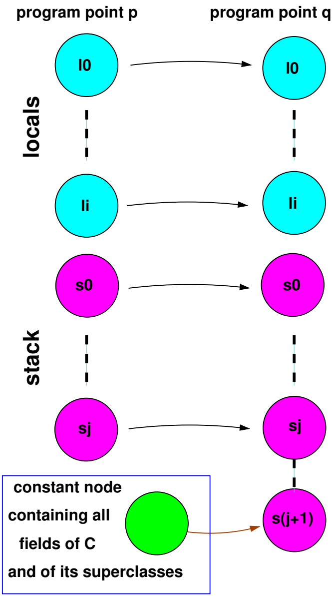


Propagation of uninitialized fields

{point p}
new C
{point q}

Nodes contain fields not yet initialized, for that local variable or stack element

Arcs propagate those fields from source to sink

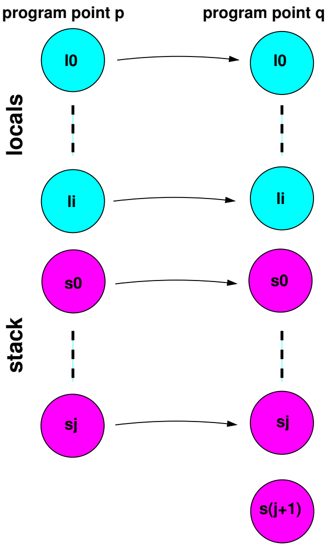


Propagation of uninitialized fields

{point p}
`const v`
{point q}

Nodes contain fields not yet initialized, for that local variable or stack element

Arcs propagate those fields from source to sink

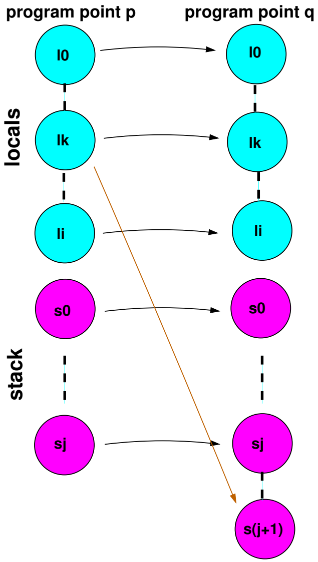


Propagation of uninitialized fields

```
{point p}
load k of type t
{point q}
```

Nodes contain fields not yet initialized, for that local variable or stack element

Arcs propagate those fields from source to sink

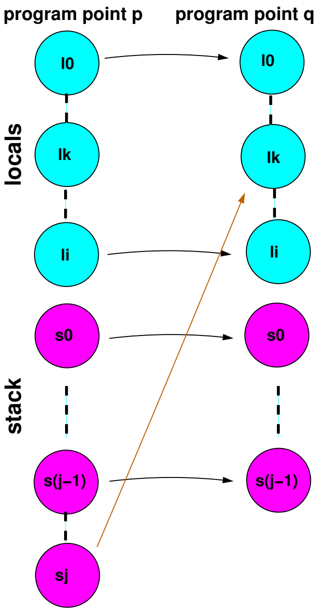


Propagation of uninitialized fields

{point p}
`store k of type t`
{point q}

Nodes contain fields not yet initialized, for that local variable or stack element

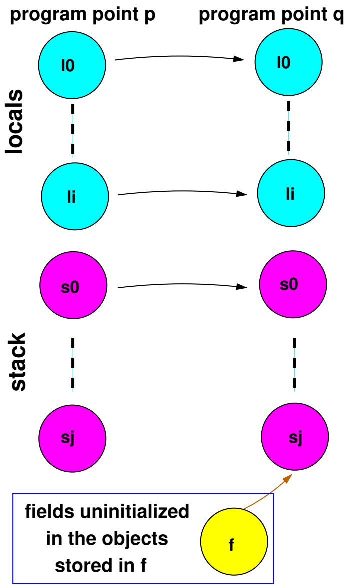
Arcs propagate those fields from source to sink



Propagation of uninitialized fields

{point p}
getfield f
{point q}

Fields are approximated in a context insensitive way.

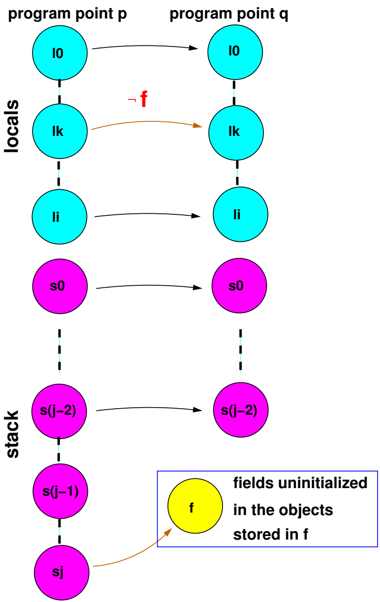


Propagation of uninitialized fields

{point p}
`putfield f`
{point q}

If local l_k is a definite alias of the stack element s_{j-1} at p.

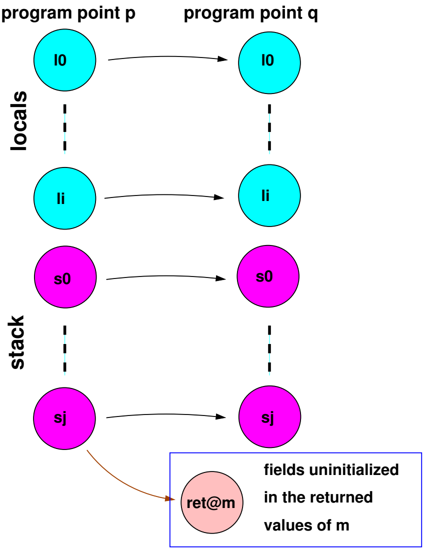
There might be more definite aliases: all are considered.



Interprocedural analysis: `return`

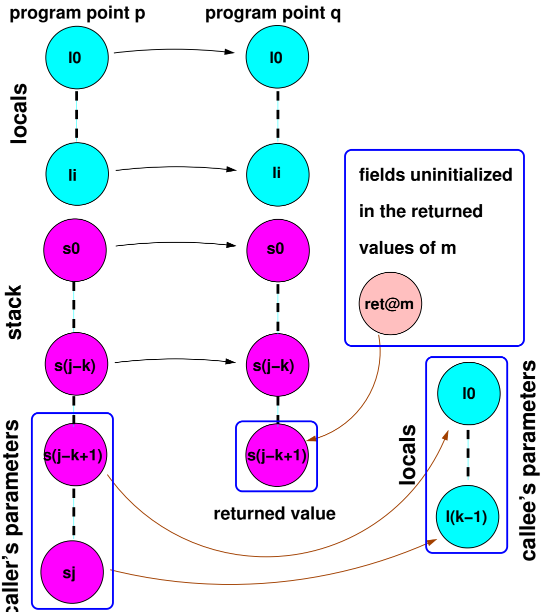
{point p}
`return type`
{point q}

A simpler rule applies when there is no returned value.



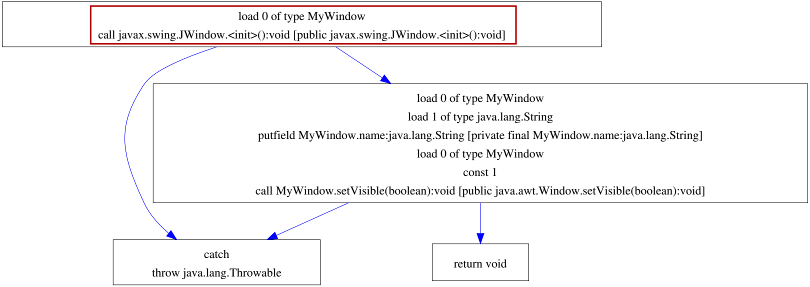
Interprocedural analysis: `call`

{point p}
call m
{point q}



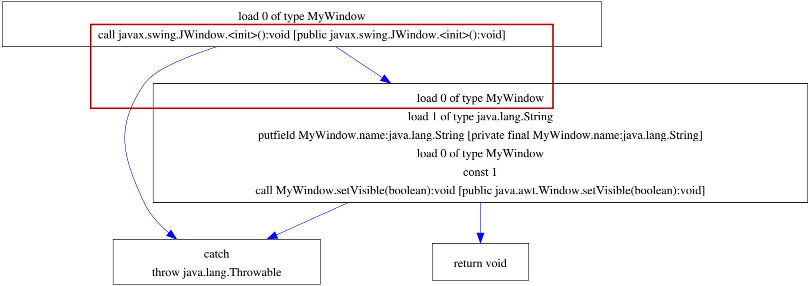
Putting everything together

- The previous graph construction rules are applied for any p and for every intraprocedural successor q of p
- A single p may have zero, one or more successors q



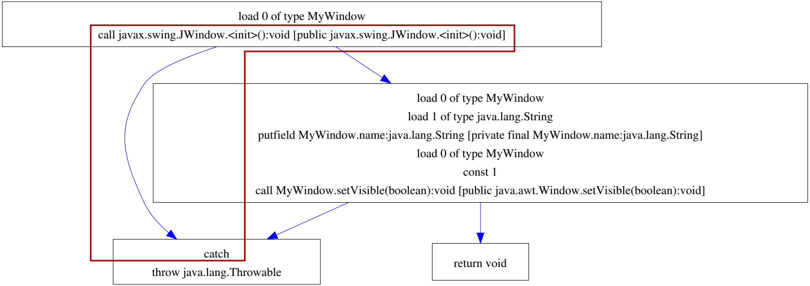
Putting everything together

- The previous graph construction rules are applied for any p and for every intraprocedural successor q of p
- A single p may have zero, one or more successors q



Putting everything together

- The previous graph construction rules are applied for any p and for every intraprocedural successor q of p
- A single p may have zero, one or more successors q



More details in the paper

- The actual graph construction is more complex
- Exceptions
- Propagation of side-effects
- Optimizations: most nodes are collapsed when they definitely have the same approximation

Correctness of the analysis

Solution of the graph

- a solution is a set of non-initialized fields for each node
- arcs stand for set inclusion
- arcs labeled with $\neg f$ stand for inclusion of everything but f
- a minimal solution can be computed through a fixpoint engine

Correctness

For each program point p, every time the operational semantics reaches p in a state $\langle [l_0 \ldots l_{i-1}] \parallel s_{j-1} :: \cdots :: s_0 \parallel \mu \rangle$, we have that

- each l_i^α is included in the solution of node `li` at p
- each s_j^α is included in the solution of node `sj` at p
- each f_k^α is included in the solution of node `fk`

Inferring rawness annotations

Definition (Raw object, reminder)

An object is *raw wrt. fields F* iff some field in F is not initialized.

We can use our analysis to annotate each program variable v that might hold raw objects (w.r.t. F):

- build the graph
- find its minimal solution
- consider the approximation of the node for v
- if it intersects F, then it gets annotated as `@Raw`
- by correctness of the approximation, this annotation is correct

Julia

An inference engine of Java program properties based on abstract interpretation

- nullness and rawness analysis are distinct analyses
  - Julia performs nullness analysis and infers a set of non-`null` fields F
  - then it performs initialization analysis and builds the `@Raw` annotations wrt. F

Experiments: integration Julia/Checker Framework

Julia

An inference engine of Java program properties based on abstract interpretation

jaif file

The Checker Framework

A generic type-checker for Java program properties based on annotation types

The jaif file contains nullness (`@Nullable`, `@NonNull`, `@PolyNull`) and initialization (`@Raw`) annotations of the program under analysis.

Experiments: a cheap analysis

program	size (lines)	time (sec.) total	time (sec.) init.	dereferences safe / all (%)
AFU	13892	209	**2**	5071 / 5143 (98.6)
JFlex	14987	118	**2**	8624 / 8753 (98.5)
plume	19652	321	**2**	8360 / 8457 (98.8)
Daikon	112077	2151	**10**	70747/75062 (94.3)

program	inferred annotations @NonNull/all (%)	@Raw/all (%)
AFU	649 / 854 (76.0)	**10** / 1124 (0.9)
JFlex	591 / 741 (79.8)	**3** / 1109 (0.3) optimal result
plume	675 / 912 (74.0)	**1** / 1118 (0.1) optimal result
Daikon	7145/10435 (68.5)	**97**/15153 (0.6)

Experiments: comparison to Nit

A tool inferring nullness and initialization (one abstract domain)

Hubert, Jensen, Pichardie. *Semantic foundations and inference of non-null annotations.* Formal Methods for Open Object-based Distributed Systems (FMOODS'08)

- sound theory
- crashes on all tests
- we could run it on a subset of AFU
- no `@Raw` annotations for receivers, return, inner types
- output contained errors

AFU	time (s.) tot.	time (s.) init.	dereferences safe/all (%)	inferred annotations `@NonNull`/all (%)	inferred annotations `@Raw`/all (%)
Julia	86	1	2683/2725 (98.5)	340/405 (83.9)	**10**/553 (1.8)
Nit	10	?	3145/3887 (80.9)	316/502 (63.0)	**63**/502 (12.5)

Experiments: comparison to JastAdd

A tool for type inference and checking

Ekman, Hedin. *Pluggable checking and inferencing of non-null types for Java*. Journal of Object Technology, 2007.

- no sound theory
- crashes on all tests but for JFlex
- does not deal with static fields
- imprecise: the receiver of a constructor is `@Raw`, always, also in helper functions

	time (s.)		dereferences	inferred annotations	
JFlex	tot.	init.	safe/all (%)	`@NonNull`/all (%)	`@Raw`/all (%)
Julia	118	2	8624/8753 (98.5)	591/741 (79.8)	**3**/1109 (0.3)
JastAdd	3	?	?/? (?)	389/? (?)	**14**/? (?)

Experiments: comparison to human-written annotations

- the plume library has a full manual annotation wrt. `@Nullable` and `@Raw`
  - 7 `@Raw` annotations, 3 `@Raw` warning suppressions
- the jaif file generated by Julia is different
  - 1 `@Raw` annotation only
  - the 6 extra are human errors: the developers removed them
  - the 3 warning suppressions are weaknesses in the type-checker
- main difference: rawness is binary for the type-checker, but not for Julia
- similar results for Daikon

Conclusion

- an inference technique for field initialization
- useful whenever a property of a field holds *after* its initialization
- fully implemented and effective
  - its results improve manual annotations
  - or can be used as a starting point for manual annotation
- proved correct through a graph-based abstract interpretation, not limited to initialization analysis:
  - class analysis
  - aliasing analysis
  - full arrays/collections analysis
- Julia: `http://julia.scienze.univr.it`
- The Checker Framework: `http://types.cs.washington.edu/checker-framework`