Static Analysis of Java: Can we be Logical?

More Bugs than Program Lines?

- software has growing importance in our daily life
- it becomes more and more complex!
- developers want to eliminate bugs
  - buggy software induces economical losses
  - bugs affect the fame of the developers
  - bugs kill an application on software repositories
- hunting bugs is hard, time-consuming, and expensive

We Want Fewer Bugs:

- programming discipline: visibility modifiers, types, generics, design patterns ...
  partial solution
- testing: definitely necessary but ...
  can often prove only the presence of a bug, not its absence
- code reviewing: certainly useful but ...
  costly and error prone
- syntactical automatic code checkers ...
  if there is a bug, they might (and typically do) miss it
- static analyses based on formal methods:
  they usually come with a correctness guarantee!

A Crowded World?

coverity®

MathWorks®

FORTIFY®
An HP Company

Microsoft®

Klocwork.

UNIVERSITY OF MARYLAND 18 56

FindBugs™
because it's easy

The use of formal methods is still the exception

Static Analysis

Static analysis proves properties of programs before actually running them. When such properties are undecidable (always. . .), we must admit a *don't know* answer

Different approaches:

- simple syntactical tests, type-checking
- more semantical data-flow analyses [Aho, Sethi, Ullman 1986]
- highly detailed proofs through theorem provers
- abstract interpretation, formal and general [Cousot & Cousot 1977]
- model-checking, also formal and general

An Example about Nullness

```
public class List {
  private List next;

  public List(List next) {
    this.next = next;        // safe dereference!
  }

  public void extend(List other) {
    List cursor = this;
    while (cursor != null) {
      other.next = new List(null);
      other = other.next;    // safe dereference!
      cursor = cursor.next;  // safe dereference!
    }
  }
}
```

Abstract Interpretation

The design of a static analysis is complex. Moreover, it is hard to compare static analyses wrt precision

Abstract interpretation [Cousot & Cousot 1977]

A general framework for the design of formally correct static analyses and for their formal comparison:

1. you define the semantics of a computational process
2. you state the property of the computations
3. you build the analysis through abstract interpretation
4. you prove correctness in a standard way
5. and you can also build an *optimal* analysis

Bibliography on Static Analysis

1. P. Cousot & R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, Fourth ACM Symp. Principles of Programming Languages, 1977, pages 238-252
2. A.V. Aho, R. Sethi & J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison Wesley Publishing Company, 1986
3. F. Nielson, H.R. Nielson & C. Hankin, *Principles of Program Analysis*, Springer, 2004

Bibliography on Nullness Analysis

1. Flanagan, Leino, *Houdini, an Annotation Assistant for ESC/Java*, Proc. of the 2001 Int. Symposium of Formal Methods Europe (FME'01)
2. Fähndrich, Leino, *Declaring and Checking non-null Types in an Object-Oriented Language*, Proc. of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)
3. Cielecki, Fulara, Jakubczyk, Jancewicz, *Propagation of JML non-null Annotations in Java Programs*, Proc. of the 4th Int. Symposium on Principles and Practice of Programming in Java (PPPJ'06)
4. Hovemeyer, Pugh, *Finding More null Pointer Bugs, but not Too Many*, Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'07)
5. Male, Pearce, Potanin, Dymnikov, *Java Bytecode Verification for `@NonNull` Types*, Proc. of the 17th Int. Conference on Compiler Construction (CC'2008)
6. Hubert, Jensen, Pichardie, *Semantic Foundations and Inference of non-null Annotations*, Proc. of the 10th Int. Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)

Almost all require manual `@NonNull` annotations

Some are not even correct

Contents

- we define a simple Java-like language and its semantics
- we define an abstraction (nullness analysis) over propositional formulas
- we improve the analysis *wrt* the fields

An Imperative Language with Objects: Commands

A simple imperative language

$$
\begin{aligned}
C ::= {} & \texttt{v := i} \mid \texttt{v := w} \mid \texttt{v := w.f} \mid \texttt{v.f := w} \mid \texttt{v := new C} \mid \texttt{inc v i} \\
& \mid \texttt{v := v}_\texttt{0}\texttt{.m(v}_\texttt{1}\texttt{, \ldots, v}_\texttt{n}\texttt{)} \\
& \mid \texttt{skip} \\
& \mid \texttt{if cond then } C \texttt{ else } C \\
& \mid \texttt{while cond do } C \\
& \mid \texttt{throw} \mid \texttt{try } C \texttt{ catch } C \\
& \mid C; C
\end{aligned}
$$

with $i \in \mathbb{Z}$ and $v, w, v_0, v_1, \ldots, v_n$ variables from a finite set $\mathcal{V}$

A real programming language will include more expressions and commands

The Semantics of the Language: Values and Environments

Values

A *value* is an element of $\mathbb{Z}$ or `null` or a *memory location* in $\mathbb{L}$.

The Semantics of the Language: Values and Environments

Values

A *value* is an element of $\mathbb{Z}$ or `null` or a *memory location* in $\mathbb{L}$.

Environments

An *environment* specifies the value of each variable in scope:

$$\mathbb{E} = \{\eta : \mathcal{V} \mapsto \mathit{Values}\}$$

For instance

If $\mathcal{V} = \{v, x, z\}$ then an environment is $[v \mapsto 11, x \mapsto \texttt{null}, z \mapsto \ell]$ where ℓ is the memory location of an *object*

The Semantics of the Language: Values and Environments

Values

A *value* is an element of $\mathbb{Z}$ or `null` or a *memory location* in $\mathbb{L}$.

Environments

An *environment* specifies the value of each variable in scope:

$$\mathbb{E} = \{\eta : \mathcal{V} \mapsto \mathit{Values}\}$$

For instance

If $\mathcal{V} = \{v, x, z\}$ then an environment is $[v \mapsto 11, x \mapsto \mathtt{null}, z \mapsto \ell]$ where ℓ is the memory location of an *object*

Why not storing the object, directly?

Locations let us model indirect references to objects. This allows us to represent shared data structures (objects reachable from more variables).

The Semantics of the Language: Objects and Memories

Objects

An *object* $o \in \mathbb{O}$ has class $o.$`class` and yields a value $o.f$ for every field f defined in $o.$`class` or in a superclass of $o.$`class`.

The Semantics of the Language: Objects and Memories

Objects

An *object* $o \in \mathbb{O}$ has class $o.$`class` and yields a value $o.f$ for every field f defined in $o.$`class` or in a superclass of $o.$`class`.

Memories

A *memory* is a map from memory locations to objects.

$$\mathbb{M} = \{\mu : \mathbb{L} \mapsto \mathbb{O}\}$$

For instance

If $\ell_1, \ell_2, \ell_3 \in \mathbb{L}$ then a memory is $[\ell_1 \mapsto o_1, \ell_2 \mapsto o_2, \ell_3 \mapsto o_1]$ where o_1 and o_2 are some objects.

The Semantics of the Language: States

Normal and Exceptional States

States $\mathbb{S}$ exist in two versions:

- *Normal* states $\mathbb{S}_n$ are pairs $\langle \eta \,\|\, \mu \rangle \in \mathbb{E} \times \mathbb{M}$
    - they are the result of a computation that ends normally
- *Exceptional* states $\mathbb{S}_e$ are underlined pairs $\underline{\langle \eta \,\|\, \mu \rangle} \in \underline{\mathbb{E} \times \mathbb{M}}$
    - they are the result of a computation that ends with an exception

The Semantics of the Language: Denotations

Denotation

A *denotation* is the *functional meaning* of a command. Namely, it is a (possibly partial) map from the state before the command is executed to the state after the command is executed. The *functional composition* of denotations is written as $\circ$.

Interpretation

An *interpretation* ι for a program is possible choice of the functional meaning of all its commands, that is, a map from each (instance of a) command C to a set of denotations $\iota(C)$.

Sets of denotations allow non-determinism and simplify the notation for the subsequent abstract interpretation.

The Semantics of the Language: Base Cases

$$[\![\texttt{v} := \texttt{i}]\!]\iota = \{\langle \eta \,\|\, \mu \rangle \Rightarrow \langle \eta[\texttt{v} \mapsto \textsf{i}] \,\|\, \mu \rangle\}$$

$$[\![\texttt{v} := \texttt{w}]\!]\iota = \{\langle \eta \,\|\, \mu \rangle \Rightarrow \langle \eta[\texttt{v} \mapsto \eta(\texttt{w})] \,\|\, \mu \rangle\}$$

$$[\![\texttt{v} := \texttt{w.f}]\!]\iota = \left\{ \langle \eta \,\|\, \mu \rangle \Rightarrow \begin{cases} \langle \eta[\texttt{v} \mapsto \mu(\eta(\texttt{w})).\texttt{f}] \,\|\, \mu \rangle & \text{if } \eta(\texttt{w}) \neq \texttt{null} \\ \underline{\langle \eta \,\|\, \mu \rangle} & \text{otherwise} \end{cases} \right\}$$

$$[\![\texttt{v.f} := \texttt{w}]\!]\iota = \left\{ \langle \eta \,\|\, \mu \rangle \Rightarrow \begin{cases} \langle \eta \,\|\, \mu[\eta(\texttt{v}) \mapsto \mu(\eta(\texttt{v}))[\texttt{f} \mapsto \eta(\texttt{w})]] \rangle & \text{if } \eta(\texttt{v}) \neq \texttt{null} \\ \underline{\langle \eta \,\|\, \mu \rangle} & \text{otherwise} \end{cases} \right.$$

The Semantics of the Language: Base Cases

$$[\![\mathtt{v} := \mathtt{new}\ \mathtt{C}]\!]\iota = \left\{ \langle \eta \,\|\, \mu \rangle \Rightarrow \begin{cases} \langle \eta[\mathtt{v} \mapsto \ell] \,\|\, \mu[\ell \mapsto \textsf{default\_object}] \rangle \\ \qquad \text{if } \ell \text{ is a fresh new location} \\ \underline{\langle \eta \,\|\, \mu \rangle} \\ \qquad \text{otherwise (if there is no free memory)} \end{cases} \right.$$

$$[\![\mathtt{inc}\ \mathtt{v}\ \mathtt{i}]\!]\iota = \{ \langle \eta \,\|\, \mu \rangle \Rightarrow \langle \eta[\mathtt{v} \mapsto \eta(\mathtt{v}) + i] \,\|\, \mu \rangle \}$$

$$[\![\mathtt{skip}]\!]\iota = \{ \langle \eta \,\|\, \mu \rangle \Rightarrow \langle \eta \,\|\, \mu \rangle \}$$

The Semantics of the Language: Conditionals

$$\left[\!\!\left[\begin{array}{c}\texttt{if cond then } C_1\\ \texttt{else } C_2\end{array}\right]\!\!\right]\iota = ([\![\texttt{cond}]\!] \circ [\![C_1]\!]\iota) \cup ([\![\neg(\texttt{cond})]\!] \circ [\![C_2]\!]\iota)$$

$$[\![\texttt{v} < \texttt{i}]\!] = \left\{\langle \eta \,\|\, \mu \rangle \Rightarrow \begin{cases} \langle \eta \,\|\, \mu \rangle & \text{if } \eta(v) < i \\ \text{undefined} & \text{otherwise} \end{cases}\right\}$$

$$[\![\neg(\texttt{v} < \texttt{i})]\!] = \left\{\langle \eta \,\|\, \mu \rangle \Rightarrow \begin{cases} \langle \eta \,\|\, \mu \rangle & \text{if } \eta(v) \geq i \\ \text{undefined} & \text{otherwise} \end{cases}\right\}$$

The Semantics of the Language: Conditionals

$$[\![\texttt{v!=null}]\!] = \left\{ \langle \eta \,\|\, \mu \rangle \Rightarrow \begin{cases} \langle \eta \,\|\, \mu \rangle & \text{if } \eta(v) \neq \texttt{null} \\ \text{undefined} & \text{otherwise} \end{cases} \right\}$$

$$[\![\neg(\texttt{v!=null})]\!] = \left\{ \langle \eta \,\|\, \mu \rangle \Rightarrow \begin{cases} \langle \eta \,\|\, \mu \rangle & \text{if } \eta(v) = \texttt{null} \\ \text{undefined} & \text{otherwise} \end{cases} \right\}$$

$$[\![\texttt{true}]\!] = \{ \langle \eta \,\|\, \mu \rangle \Rightarrow \langle \eta \,\|\, \mu \rangle \}$$

$$[\![\neg(\texttt{true})]\!] = \{ \langle \eta \,\|\, \mu \rangle \Rightarrow \text{undefined} \}$$

The Semantics of the Language: Loops

$$[\![\underbrace{\texttt{while cond do } C}_{C'}]\!]\iota = ([\![\texttt{cond}]\!] \circ [\![C]\!]\iota \circ \iota(C')) \cup [\![\neg(\texttt{cond})]\!]$$

The Semantics of the Language: Exception Handling

$$[\![\mathtt{throw}]\!]\iota = \{\langle \eta \,\|\, \mu \rangle \Rightarrow \underline{\langle \eta \,\|\, \mu \rangle}\}$$

$$[\![\mathtt{try}\ C_1\ \mathtt{catch}\ C_2]\!]\iota = ([\![C_1]\!]\iota \circ [\![\mathit{normal}]\!]) \cup ([\![C_1]\!]\iota \circ [\![\mathit{catch}]\!] \circ [\![C_2]\!]\iota)$$

$$[\![\mathit{normal}]\!] = \{\langle \eta \,\|\, \mu \rangle \Rightarrow \langle \eta \,\|\, \mu \rangle\}$$

$$[\![\mathit{catch}]\!] = \{\underline{\langle \eta \,\|\, \mu \rangle} \Rightarrow \langle \eta \,\|\, \mu \rangle\}$$

The Semantics of the Language: Exception Handling

$$\llbracket \texttt{throw} \rrbracket \iota = \{ \langle \eta \,\|\, \mu \rangle \Rightarrow \underline{\langle \eta \,\|\, \mu \rangle} \}$$

$$\llbracket \texttt{try } C_1 \texttt{ catch } C_2 \rrbracket \iota = (\llbracket C_1 \rrbracket \iota \circ \llbracket \mathit{normal} \rrbracket) \cup (\llbracket C_1 \rrbracket \iota \circ \llbracket \mathit{catch} \rrbracket \circ \llbracket C_2 \rrbracket \iota)$$

$$\llbracket C_1 ; C_2 \rrbracket \iota = (\llbracket C_1 \rrbracket \iota \circ \llbracket \mathit{exceptional} \rrbracket) \cup (\llbracket C_1 \rrbracket \iota \circ \llbracket \mathit{normal} \rrbracket \circ \llbracket C_2 \rrbracket \iota)$$

$$\llbracket \mathit{normal} \rrbracket = \{ \langle \eta \,\|\, \mu \rangle \Rightarrow \langle \eta \,\|\, \mu \rangle \}$$

$$\llbracket \mathit{exceptional} \rrbracket = \{ \underline{\langle \eta \,\|\, \mu \rangle} \Rightarrow \underline{\langle \eta \,\|\, \mu \rangle} \}$$

$$\llbracket \mathit{catch} \rrbracket = \{ \underline{\langle \eta \,\|\, \mu \rangle} \Rightarrow \langle \eta \,\|\, \mu \rangle \}$$

An Algebraic Definition of the Semantics of Java

We have presented an algebraic definition of the semantics of the kernel of Java. Its building blocks are

- constant sets of denotations: $[\![v < i]\!]$, $[\![normal]\!]$, $[\![v := w]\!]$, ...
- operators over sets of denotations: $\circ$, $\cup$, plug (for method calls)

We are ready to use it for abstract interpretation

Properties of Computations

What is a property of a computation?

It is the set of denotations that satisfy that property!

Properties of Computations

What is a property of a computation?

It is the set of denotations that satisfy that property!

Example: the property “at the end $\mathtt{x}$ is 5”

$\{\delta \mid \text{for all } \langle \eta \parallel \mu \rangle \text{ if } \delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle \text{ then } \eta'(\mathtt{x}) = 5\}$

Properties of Computations

What is a property of a computation?

It is the set of denotations that satisfy that property!

Example: the property "at the end $\mathtt{x}$ is 5"

$\{\delta \mid$ for all $\langle \eta \parallel \mu \rangle$ if $\delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle$ then $\eta'(\mathtt{x}) = 5\}$

```
x:=5
```

Properties of Computations

What is a property of a computation?

It is the set of denotations that satisfy that property!

Example: the property "at the end $\mathtt{x}$ is 5"

$\{\delta \mid$ for all $\langle \eta \parallel \mu \rangle$ if $\delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle$ then $\eta'(\mathtt{x}) = 5\}$

```
x:=5
```

Example: the property "$\mathtt{x}$ is modified into 5"

$\{\delta \mid$ for all $\langle \eta \parallel \mu \rangle$ if $\delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle$ then $\eta(\mathtt{x}) \neq 5$ and $\eta'(\mathtt{x}) = 5\}$

Properties of Computations

What is a property of a computation?

It is the set of denotations that satisfy that property!

Example: the property "at the end $\mathtt{x}$ is 5"

$\{\delta \mid$ for all $\langle \eta \parallel \mu \rangle$ if $\delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle$ then $\eta'(\mathtt{x}) = 5\}$

```
x:=5
```

Example: the property "$\mathtt{x}$ is modified into 5"

$\{\delta \mid$ for all $\langle \eta \parallel \mu \rangle$ if $\delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle$ then $\eta(\mathtt{x}) \neq 5$ and $\eta'(\mathtt{x}) = 5\}$

```
if (x!=5) then x:=5 else while true skip
      if (x!=5) then x:=5 else throw
```

Properties of Computations

Example: the property “$\mathtt{x}$ increases”

$\{\delta \mid \text{for all } \langle \eta \parallel \mu \rangle \text{ if } \delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle \text{ then } \eta(\mathtt{x}) < \eta'(\mathtt{x})\}$

Properties of Computations

Example: the property "$\mathtt{x}$ increases"

$\{\delta \mid$ for all $\langle \eta \,\|\, \mu \rangle$ if $\delta(\langle \eta \,\|\, \mu \rangle) = \langle \eta' \,\|\, \mu' \rangle$ then $\eta(\mathtt{x}) < \eta'(\mathtt{x})\}$

```
if (x<6) then inc x 2 else inc x 1
```

Properties of Computations

Example: the property “x increases”

$\{\delta \mid \text{for all } \langle \eta \parallel \mu \rangle \text{ if } \delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle \text{ then } \eta(\mathtt{x}) < \eta'(\mathtt{x})\}$

```
if (x<6) then inc x 2 else inc x 1
```

Example: the property “at the end x is null”

$\{\delta \mid \text{for all } \langle \eta \parallel \mu \rangle \text{ if } \delta(\langle \eta \parallel \mu \rangle) = \langle \eta' \parallel \mu' \rangle \text{ then } \eta'(\mathtt{x}) = \mathtt{null}\}$

Properties of Computations

Example: the property "$\mathtt{x}$ increases"

$\{\delta \mid \text{for all } \langle\eta \parallel \mu\rangle \text{ if } \delta(\langle\eta \parallel \mu\rangle) = \langle\eta' \parallel \mu'\rangle \text{ then } \eta(\mathtt{x}) < \eta'(\mathtt{x})\}$

```
if (x<6) then inc x 2 else inc x 1
```

Example: the property "at the end $\mathtt{x}$ is $\mathtt{null}$"

$\{\delta \mid \text{for all } \langle\eta \parallel \mu\rangle \text{ if } \delta(\langle\eta \parallel \mu\rangle) = \langle\eta' \parallel \mu'\rangle \text{ then } \eta'(\mathtt{x}) = \mathtt{null}\}$

```
if (x!=null) then while true skip else skip
```

Logic as a Language for Computational Properties

We want to use propositional formulas over the variables of the program as a language to specify properties of nullness in denotations:

- $\check{x}$ means that at the beginning `x` holds `null`
- $\hat{x}$ means that at the end `x` holds `null`
- $\neg\hat{x}$ means that at the end `x` does not hold `null`
- $\hat{x} \vee \hat{y}$ means that at the end `x` holds `null` or `y` holds `null` (or both)
- $\check{x} \to \hat{y}$ means that if at the beginning `x` holds `null` then at the end `y` holds `y`
- ...

The Meaning of a Logical Formula

Nullness *extractor*

$$\mathsf{nullness}(\langle \eta \parallel \mu \rangle) = \{\mathtt{v} \mid \eta(\mathtt{v}) = \mathtt{null}\}$$

$$\mathsf{nullness}(\underline{\langle \eta \parallel \mu \rangle}) = \{\mathtt{v} \mid \eta(\mathtt{v}) = \mathtt{null}\} \cup \{e\}$$

The Meaning of a Logical Formula

Nullness *extractor*

$$\mathsf{nullness}(\langle \eta \parallel \mu \rangle) = \{\mathtt{v} \mid \eta(\mathtt{v}) = \mathtt{null}\}$$

$$\mathsf{nullness}(\underline{\langle \eta \parallel \mu \rangle}) = \{\mathtt{v} \mid \eta(\mathtt{v}) = \mathtt{null}\} \cup \{e\}$$

The property expressed by a formula ϕ

$$\gamma(\phi) = \left\{ \delta \;\middle|\; \begin{array}{l} \text{for all } \sigma \text{ such that } \delta(\sigma) \text{ is defined,} \\ \text{we have } \check{\mathsf{nullness}}(\sigma) \cup \hat{\mathsf{nullness}}(\delta(\sigma)) \models \phi \end{array} \right\}$$

A non-Standard Semantics over Logical Formulas

$$[\![\mathtt{v} := \mathtt{i}]\!]^{\alpha}\iota = \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{v} \wedge \mathsf{unchanged}$$

$$[\![\mathtt{v} := \mathtt{w}]\!]^{\alpha}\iota = \neg\check{e} \wedge \neg\hat{e} \wedge (\check{w} \leftrightarrow \hat{v}) \wedge \mathsf{unchanged}$$

$$[\![\mathtt{v} := \mathtt{w.f}]\!]^{\alpha}\iota = \neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})) \wedge \mathsf{unchanged}$$

$$[\![\mathtt{v.f} := \mathtt{w}]\!]^{\alpha}\iota = \neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\check{v}) \wedge \mathsf{unchanged}$$

unchanged is a formula that states a frame condition: all variables $\mathtt{x}$ never touched by the command keep their nullness: $\check{x} \leftrightarrow \hat{x}$

A non-Standard Semantics over Logical Formulas

$$[\![\mathtt{v} := \mathtt{new\ C}]\!]^\alpha = \neg\check{e} \wedge (\neg\hat{e} \to \neg\hat{\mathtt{v}}) \wedge (\hat{e} \to (\check{v} \leftrightarrow \hat{v})) \wedge \mathsf{unchanged}$$

$$[\![\mathtt{v} < \mathtt{i}]\!]^\alpha = \neg\check{e} \wedge \neg\hat{e} \wedge \mathsf{unchanged}$$

$$[\![\neg(\mathtt{v} < \mathtt{i})]\!]^\alpha = \neg\check{e} \wedge \neg\hat{e} \wedge \mathsf{unchanged}$$

$$[\![\mathtt{v}! = \mathtt{null}]\!]^\alpha = \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{\mathtt{v}} \wedge \mathsf{unchanged}$$

$$[\![\neg(\mathtt{v}! = \mathtt{null})]\!]^\alpha = \neg\check{e} \wedge \neg\hat{e} \wedge \hat{\mathtt{v}} \wedge \mathsf{unchanged}$$

$$\begin{aligned} [\![\mathit{normal}]\!]^\alpha &= \neg\check{e} \wedge \neg\hat{e} \wedge \mathsf{unchanged} \\ [\![\mathit{catch}]\!]^\alpha &= \check{e} \wedge \neg\hat{e} \wedge \mathsf{unchanged} \\ [\![\mathit{exceptional}]\!]^\alpha &= \check{e} \wedge \hat{e} \wedge \mathsf{unchanged} \end{aligned}$$

A non-Standard Semantics over Logical Formulas

$$\cup^\alpha \text{ is } \vee$$

$$\phi_1 \circ^\alpha \phi_2 = \exists_{-} \phi_1[\hat{} \to -] \wedge \phi_2[\check{} \to -]$$

$$\mathrm{plug}^\alpha(\phi) = (\exists_{\hat{}\ \mathrm{but}\ \hat{\mathtt{w}}} \phi)[\check{this} \mapsto \check{v}_0, \check{w}_1 \mapsto \check{v}_1, \ldots, \check{w}_n \mapsto \check{v}_n, \hat{w} \mapsto \hat{v}] \wedge \mathrm{unchanged}$$

A non-Standard Semantics over Logical Formulas

$$\cup^\alpha \text{ is } \vee$$

$$\phi_1 \circ^\alpha \phi_2 = \exists_{-} \phi_1[\hat{} \rightarrow -] \wedge \phi_2[\check{} \rightarrow -]$$

$$\text{plug}^\alpha(\phi) = (\exists_{\hat{}\ \text{but}\ \hat{\mathtt{w}}} \phi)[\check{\mathit{this}} \mapsto \check{v}_0, \check{w}_1 \mapsto \check{v}_1, \ldots, \check{w}_n \mapsto \check{v}_n, \hat{w} \mapsto \hat{v}] \wedge \text{unchanged}$$

This non-standard semantics can be proved to be correct:

$$[\![\mathtt{v} := \mathtt{i}]\!]\gamma(\iota) \subseteq \gamma([\![\mathtt{v} := \mathtt{i}]\!]^\alpha \iota)$$

$$\gamma(\phi_1) \circ \gamma(\phi_2) \subseteq \gamma(\phi_1 \circ^\alpha \phi_2)$$

$$\vdots$$

Example 1

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l\|l} \texttt{w := new C} & \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge \mathsf{unchanged} \\ \texttt{w.f := v} & \end{array}$$

Example 1

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l} \texttt{w} := \texttt{new C} & \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v}) \\ \texttt{w.f} := \texttt{v} & \end{array}$$

Example 1

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l} \texttt{w} := \texttt{new C} & \neg\check{e} \wedge (\neg\hat{e} \to \neg\hat{w}) \wedge (\hat{e} \to (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v}) \\ \texttt{w.f} := \texttt{v} & \end{array}$$

Hence the dereference in `w.f:=v` never throws a null-pointer exception

$$\neg\check{e} \wedge (\neg\hat{e} \to \neg\hat{w}) \wedge (\hat{e} \to (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v}) \models (\neg\hat{e} \to \neg\hat{w})$$

Example 2

We assume that only variables v and w are in scope.

$$\begin{array}{l||l} \texttt{w := new C} & \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge \mathsf{unchanged} \\ \texttt{v := w} & \\ \texttt{w.f := v} & \end{array}$$

Example 2

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l} \texttt{w} := \texttt{new C} & \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v}) \\ \texttt{v} := \texttt{w} & \\ \texttt{w.f} := \texttt{v} & \end{array}$$

Example 2

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l}
\texttt{w} := \texttt{new C} & \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v}) \\
\texttt{v} := \texttt{w} & \neg\check{e} \wedge \neg\hat{e} \wedge (\check{w} \leftrightarrow \hat{v}) \wedge \text{unchanged} \\
\texttt{w.f} := \texttt{v} &
\end{array}$$

Example 2

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l}
\texttt{w} := \texttt{new C} & \neg\check{e} \wedge (\neg\hat{e} \to \neg\hat{w}) \wedge (\hat{e} \to (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v}) \\
\texttt{v} := \texttt{w} & \neg\check{e} \wedge \neg\hat{e} \wedge (\check{w} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w}) \\
\texttt{w.f} := \texttt{v} &
\end{array}$$

Example 2

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{r}\left.\begin{array}{r}\texttt{w} := \texttt{new C}\\ \texttt{v} := \texttt{w}\end{array}\right\}\\ \texttt{w.f} := \texttt{v}\end{array}\;\left\|\;\begin{array}{l}\neg\check{e}\wedge\neg\hat{e}\wedge\neg\hat{v}\wedge\neg\hat{w}\\ \\ \end{array}\right.$$

Example 2

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{rl} \left.\begin{array}{r}\texttt{w} := \texttt{new C}\\ \texttt{v} := \texttt{w}\end{array}\right\} & \Bigg\| \; \neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{v} \wedge \neg\hat{w} \\ \texttt{w.f} := \texttt{v} \qquad & \Big\| \end{array}$$

Hence the dereference in `w.f:=v` never throws a null-pointer exception

$$\neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{v} \wedge \neg\hat{w} \models (\neg\hat{e} \rightarrow \neg\hat{w})$$

Example 3

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{r||l}
\texttt{v := w} & \neg\check{e} \wedge \neg\hat{e} \wedge (\check{w} \leftrightarrow \hat{v}) \wedge \mathsf{unchanged} \\
\texttt{w.f := v} & \\
\texttt{v.g := w} &
\end{array}$$

Example 3

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{r||l} \texttt{v} := \texttt{w} & \neg\check{e} \wedge \neg\hat{e} \wedge (\check{w} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w}) \\ \texttt{w.f} := \texttt{v} & \\ \texttt{v.g} := \texttt{w} & \end{array}$$

Example 3

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l} \texttt{v} := \texttt{w} & \neg\check{e} \wedge \neg\hat{e} \wedge (\check{w} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w}) \\ \texttt{w.f} := \texttt{v} & \neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\check{w}) \wedge \text{unchanged} \\ \texttt{v.g} := \texttt{w} & \end{array}$$

Example 3

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{r||l}
\texttt{v := w} & \neg\check{e} \wedge \neg\hat{e} \wedge (\check{w} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w}) \\
\texttt{w.f := v} & \neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\check{w}) \wedge (\check{w} \leftrightarrow \hat{w}) \wedge (\check{v} \leftrightarrow \hat{v}) \\
\texttt{v.g := w} &
\end{array}$$

Example 3

We assume that only variables v and w are in scope.

$$\begin{array}{l||l} \left.\begin{array}{l}\texttt{v} := \texttt{w}\\ \texttt{w.f} := \texttt{v}\end{array}\right\} & \neg\check{e} \wedge (\check{v} \leftrightarrow \hat{w}) \wedge (\check{w} \leftrightarrow \hat{w}) \wedge (\hat{w} \leftrightarrow \hat{e}) \\ \texttt{v.g} := \texttt{w} & \end{array}$$

Example 3

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l}\left.\begin{array}{l}\texttt{v} := \texttt{w}\\ \texttt{w.f} := \texttt{v}\end{array}\right\} \\ \texttt{v.g} := \texttt{w}\end{array} \Bigg\| \; \neg\check{e} \wedge (\check{v} \leftrightarrow \hat{w}) \wedge (\check{w} \leftrightarrow \hat{w}) \wedge (\hat{w} \leftrightarrow \hat{e})$$

Hence the dereference in `v.g:=w` never throws a null-pointer exception

$$\neg\check{e} \wedge (\check{v} \leftrightarrow \hat{w}) \wedge (\check{w} \leftrightarrow \hat{w}) \wedge (\hat{w} \leftrightarrow \hat{e}) \models (\neg\hat{e} \rightarrow \neg\hat{v})$$

Example 4

We assume that only variables v and w are in scope.

```
if v = null
  then
    while true do
      skip
  else
    skip
v.f := w
```

$\neg\check{e} \wedge \neg\hat{e} \wedge \hat{v} \wedge \mathsf{unchanged}$

Example 4

We assume that only variables v and w are in scope.

```
if v = null
  then
    while true do
      skip
  else
    skip
v.f := w
```

$$\neg\check{e} \wedge \neg\hat{e} \wedge \hat{v} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$$

Example 4

We assume that only variables `v` and `w` are in scope.

`if v = null`	$\neg\check{e} \land \neg\hat{e} \land \hat{v} \land (\check{v} \leftrightarrow \hat{v}) \land (\check{w} \leftrightarrow \hat{w})$
`then`	
`while true do`	$\neg\check{e} \land \neg\hat{e} \land \mathsf{unchanged}$
`skip`	
`else`	
`skip`	
`v.f := w`	

Example 4

We assume that only variables `v` and `w` are in scope.

`if v = null`	$\neg\check{e} \land \neg\hat{e} \land \hat{v} \land (\check{v} \leftrightarrow \hat{v}) \land (\check{w} \leftrightarrow \hat{w})$
`  then`	
`    while true do`	$\neg\check{e} \land \neg\hat{e} \land (\check{v} \leftrightarrow \hat{v}) \land (\check{w} \leftrightarrow \hat{w})$
`      skip`	
`  else`	
`    skip`	
`v.f := w`	

Example 4

We assume that only variables `v` and `w` are in scope.

```
if v = null          || $\neg\check{e} \wedge \neg\hat{e} \wedge \hat{v} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$
  then               ||
    while true do    || $\neg\check{e} \wedge \neg\hat{e} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$
      skip           || $\neg\check{e} \wedge \neg\hat{e} \wedge \mathsf{unchanged}$
  else               ||
    skip             || $\neg\check{e} \wedge \neg\hat{e} \wedge \mathsf{unchanged}$
v.f := w             ||
```

Example 4

We assume that only variables `v` and `w` are in scope.

`if v = null`	$\neg\check{e} \land \neg\hat{e} \land \hat{v} \land (\check{v} \leftrightarrow \hat{v}) \land (\check{w} \leftrightarrow \hat{w})$
`  then`	
`    while true do`	$\neg\check{e} \land \neg\hat{e} \land (\check{v} \leftrightarrow \hat{v}) \land (\check{w} \leftrightarrow \hat{w}) \equiv \phi$
`      skip`	$\neg\check{e} \land \neg\hat{e} \land (\check{v} \leftrightarrow \hat{v}) \land (\check{w} \leftrightarrow \hat{w}) \equiv \phi$
`  else`	
`    skip`	$\neg\check{e} \land \neg\hat{e} \land (\check{v} \leftrightarrow \hat{v}) \land (\check{w} \leftrightarrow \hat{w})$
`v.f := w`	

Example 4

We assume that only variables `v` and `w` are in scope.

`if v = null`	$\neg\check{e} \wedge \neg\hat{e} \wedge \hat{v} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$
`then`	

`while true do`
`skip`
	$\mathsf{lfp} \equiv ((\phi \circ^{\alpha} \phi \circ^{\alpha} \mathsf{lfp}) \vee \mathsf{false})$

`else`	
`skip`	$\neg\check{e} \wedge \neg\hat{e} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$
`v.f := w`	

Example 4

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l}
\texttt{if v = null} & \neg\check{e}\wedge\neg\hat{e}\wedge\hat{v}\wedge(\check{v}\leftrightarrow\hat{v})\wedge(\check{w}\leftrightarrow\hat{w})\\
\quad\texttt{then} & \\
\left.\begin{array}{l}\quad\quad\texttt{while true do}\\ \quad\quad\quad\texttt{skip}\end{array}\right\} & \mathsf{false}\\
\quad\texttt{else} & \\
\quad\quad\texttt{skip} & \neg\check{e}\wedge\neg\hat{e}\wedge(\check{v}\leftrightarrow\hat{v})\wedge(\check{w}\leftrightarrow\hat{w})\\
\texttt{v.f := w} &
\end{array}$$

Example 4

We assume that only variables `v` and `w` are in scope.

```
if v = null
 then
  while true do
   skip
 else
  skip
v.f := w
```

$$\neg \check{e} \wedge \neg \hat{e} \wedge \neg \check{v} \wedge \neg \hat{v} \wedge (\check{w} \leftrightarrow \hat{w})$$

Example 4

We assume that only variables `v` and `w` are in scope.

$$\left.\begin{array}{l}\left.\begin{array}{l}\texttt{if v = null}\\ \quad\texttt{then}\\ \qquad\texttt{while true do}\\ \qquad\quad\texttt{skip}\\ \quad\texttt{else}\\ \qquad\texttt{skip}\end{array}\right\}\\ \texttt{v.f := w}\end{array}\right\| \; \neg\check{e}\wedge\neg\hat{e}\wedge\neg\check{v}\wedge\neg\hat{v}\wedge(\check{w}\leftrightarrow\hat{w})$$

Hence the dereference in `v.f:=w` never throws a null-pointer exception

$$\neg\check{e}\wedge\neg\hat{e}\wedge\neg\check{v}\wedge\neg\hat{v}\wedge(\check{w}\leftrightarrow\hat{w}) \models (\neg\hat{e}\rightarrow\neg\hat{v})$$

Example 5

We assume that only variables v and w are in scope.

```
try
  w := new C
catch
  skip
w.f := v
```

$$\neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge \mathsf{unchanged}$$

Example 5

We assume that only variables `v` and `w` are in scope.

```
try
  w := new C
catch
  skip
w.f := v
```

$$\neg\check{e} \land (\neg\hat{e} \rightarrow \neg\hat{w}) \land (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \land (\check{v} \leftrightarrow \hat{v})$$

Example 5

We assume that only variables `v` and `w` are in scope.

`try`	
`  w := new C`	$\neg\check{e} \land (\neg\hat{e} \to \neg\hat{w}) \land (\hat{e} \to (\check{w} \leftrightarrow \hat{w})) \land (\check{v} \leftrightarrow \hat{v})$
`catch`	
`  skip`	$\neg\check{e} \land \neg\hat{e} \land \text{unchanged}$
`w.f := v`	

Example 5

We assume that only variables `v` and `w` are in scope.

`try`	
`  w := new C`	$\neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v})$
`catch`	
`  skip`	$\neg\check{e} \wedge \neg\hat{e} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$
`w.f := v`	

Example 5

We assume that only variables `v` and `w` are in scope.

`try`	
`  w := new C`	$\neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v})$
`catch`	
`  skip`	$\neg\check{e} \wedge \neg\hat{e} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$
`w.f := v`	

Example 5

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l}
\begin{array}{l}
\left.\begin{array}{l}
\texttt{try}\\
\quad\texttt{w := new C}\\
\texttt{catch}\\
\quad\texttt{skip}
\end{array}\right\}\\
\texttt{w.f := v}
\end{array}
&
\neg\check{e} \wedge \neg\hat{e} \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\neg\hat{w} \vee (\check{w} \leftrightarrow \hat{w}))
\end{array}$$

Example 5

We assume that only variables v and w are in scope.

$$\begin{array}{l} \left.\begin{array}{l}\texttt{try}\\ \quad\texttt{w} := \texttt{new C}\\ \texttt{catch}\\ \quad\texttt{skip}\end{array}\right\}\\ \texttt{w.f} := \texttt{v}\end{array} \;\Bigg\|\; \neg\check{e}\wedge\neg\hat{e}\wedge(\check{v}\leftrightarrow\hat{v})\wedge(\neg\hat{w}\vee(\check{w}\leftrightarrow\hat{w}))$$

Hence we cannot prove that the dereference in `w.f:=v` never throws a null-pointer exception

$$\neg\check{e}\wedge\neg\hat{e}\wedge(\check{v}\leftrightarrow\hat{v})\wedge(\neg\hat{w}\vee(\check{w}\leftrightarrow\hat{w})) \not\models (\neg\hat{e}\rightarrow\neg\hat{w})$$

Example 6

We assume that only variables `v` and `w` are in scope.

$$
\begin{array}{l||l}
\texttt{w} := \texttt{new C} & \neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v}) \\
\texttt{w.f} := \texttt{w} & \\
\texttt{w} := \texttt{w.f} & \\
\texttt{w.f} := \texttt{w} &
\end{array}
$$

Example 6

We assume that only variables `v` and `w` are in scope.

`w := new C`	$\neg\check{e} \wedge (\neg\hat{e} \rightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{w} \leftrightarrow \hat{w})) \wedge (\check{v} \leftrightarrow \hat{v})$
`w.f := w`	$\neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\check{w}) \wedge (\check{v} \leftrightarrow \hat{v}) \wedge (\check{w} \leftrightarrow \hat{w})$
`w := w.f`	
`w.f := w`	

Example 6

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l}
\texttt{w := new C} & \neg\check{e}\wedge(\neg\hat{e}\rightarrow\neg\hat{w})\wedge(\hat{e}\rightarrow(\check{w}\leftrightarrow\hat{w}))\wedge(\check{v}\leftrightarrow\hat{v}) \\
\texttt{w.f := w} & \neg\check{e}\wedge(\neg\hat{e}\leftrightarrow\neg\check{w})\wedge(\check{v}\leftrightarrow\hat{v})\wedge(\check{w}\leftrightarrow\hat{w}) \\
\texttt{w := w.f} & \neg\check{e}\wedge(\neg\hat{e}\leftrightarrow\neg\check{w})\wedge(\hat{e}\rightarrow(\check{w}\leftrightarrow\hat{w}))\wedge(\check{v}\leftrightarrow\hat{v}) \\
\texttt{w.f := w} &
\end{array}$$

Example 6

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l||l} \left.\begin{array}{l}\texttt{w} := \texttt{new C}\\ \texttt{w.f} := \texttt{w}\end{array}\right\} & \neg\check{e}\wedge\neg\hat{e}\wedge(\check{v}\leftrightarrow\hat{v})\wedge\neg\hat{w}\\ \texttt{w} := \texttt{w.f} & \neg\check{e}\wedge(\neg\hat{e}\leftrightarrow\neg\check{w})\wedge(\hat{e}\rightarrow(\check{w}\leftrightarrow\hat{w}))\wedge(\check{v}\leftrightarrow\hat{v})\\ \texttt{w.f} := \texttt{w} & \end{array}$$

Example 6

We assume that only variables v and w are in scope.

$$\begin{array}{l||l} \left.\begin{array}{l} \texttt{w} := \texttt{new C} \\ \texttt{w.f} := \texttt{w} \\ \texttt{w} := \texttt{w.f} \end{array}\right\} & \neg\check{e} \wedge \neg\hat{e} \wedge (\check{v} \leftrightarrow \hat{v}) \\ \texttt{w.f} := \texttt{w} & \end{array}$$

Example 6

We assume that only variables `v` and `w` are in scope.

$$\left.\begin{array}{l}\left.\begin{array}{l}\texttt{w := new C}\\ \texttt{w.f := w}\\ \texttt{w := w.f}\end{array}\right\}\\ \texttt{w.f := w}\end{array}\right\| \; \neg\check{e} \wedge \neg\hat{e} \wedge (\check{v} \leftrightarrow \hat{v})$$

Example 6

We assume that only variables `v` and `w` are in scope.

$$\begin{array}{l}\left.\begin{array}{l}\mathtt{w := new\ C}\\ \mathtt{w.f := w}\\ \mathtt{w := w.f}\end{array}\right\}\\ \mathtt{w.f := w}\end{array}\ \Bigg\|\ \neg\check{e}\wedge\neg\hat{e}\wedge(\check{v}\leftrightarrow\hat{v})$$

Hence we cannot prove that the dereference in `w.f:=w` never throws a null-pointer exception. Imprecise!

$$\neg\check{e}\wedge\neg\hat{e}\wedge(\check{v}\leftrightarrow\hat{v})\not\models(\neg\hat{e}\rightarrow\neg\hat{w})$$

Boolean Formulas for Nullness Analysis: Pros and Cons

Pros

- simple, theoretically clean
- efficient (binary decision diagrams)
- completely flow and context sensitive
- precise *wrt* local variables and exceptions

Boolean Formulas for Nullness Analysis: Pros and Cons

Pros

- simple, theoretically clean
- efficient (binary decision diagrams)
- completely flow and context sensitive
- precise *wrt* local variables and exceptions

Cons

- no approximation for fields
- no approximation for arrays

The Meaning of Implication

$\check{x} \rightarrow \hat{y}$

This is the set of denotations such that, if `x` is `null` in the input, then `y` is `null` in the output.

In terms of functional composition

$$\gamma(\check{x} \rightarrow \hat{y}) = \{\delta \mid \text{for all } \delta' \in \hat{x} \text{ we have } \delta' \circ \delta \in \hat{y}\}$$

Only in terms of γ

$$\gamma(\check{x} \rightarrow \hat{y}) = \gamma(\hat{x}) \rightarrow \gamma(\hat{y})$$

where

$$X \rightarrow Y = \{\delta \mid \text{for all } \delta' \in X \text{ we have } \delta' \circ \delta \in Y\}$$

$\rightarrow$ is the *linear refinement* of Giacobazzi & Scozzari '98

Oracle Semantics for the Fields

Our previous definition

$$[\![\texttt{v} := \texttt{w.f}]\!]^{\alpha}\iota = \neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})) \wedge \text{unchanged}$$

This corresponds to a pessimistic oracle $O = \emptyset$: no field is definitely non-`null` ⇒ imprecise but definitely correct

Oracle Semantics for the Fields

Our previous definition

$[\![\texttt{v} := \texttt{w.f}]\!]^{\alpha}\iota = \neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})) \wedge \mathsf{unchanged}$

This corresponds to a pessimistic oracle $O = \emptyset$: no field is definitely non-`null` ⇒ imprecise but definitely correct

Another definition

$[\![\texttt{v} := \texttt{w.f}]\!]^{\alpha}\iota = \neg\check{e} \wedge (\neg\hat{e} \leftrightarrow \neg\hat{w}) \wedge (\hat{e} \rightarrow (\check{v} \leftrightarrow \hat{v})) \wedge (\neg\hat{e} \rightarrow \neg\hat{v}) \wedge \mathsf{unchanged}$

This corresponds to an optimistic oracle $O = \{\textit{all fields}\}$: all fields are definitely non-`null` ⇒ precise but in general incorrect

Oracle Semantics for the Fields

More generally...

Given an oracle O (*i.e.*, a set of fields assumed to hold always a non-`null` value when they are read), we define

$$[\![\mathtt{v} := \mathtt{w.f}]\!]^{\alpha}\iota = \begin{cases} \neg\check{e} \land (\neg\hat{e} \leftrightarrow \neg\hat{w}) \land (\hat{e} \to (\check{v} \leftrightarrow \hat{v})) \land \text{unchanged} \\ \quad \text{if } \mathtt{f} \notin O \\ \neg\check{e} \land (\neg\hat{e} \leftrightarrow \neg\hat{w}) \land (\hat{e} \to (\check{v} \leftrightarrow \hat{v})) \land (\neg\hat{e} \to \neg\hat{v}) \land \text{unchang} \\ \quad \text{if } \mathtt{f} \in O \end{cases}$$

We get an abstract semantics (a nullness analysis) parameterised *wrt* O. That semantics might be incorrect if O is not correct

Looking for a Correct Oracle

Theorem

1. *If O is correct (that is, if it only contains fields that actually hold a non-`null` value when they are read) then the induced nullness analysis is correct*
2. *The larger O, the more precise is the induced nullness analysis*

Fine, but how do we find a correct and possibly large oracle?

Looking for a Correct Oracle

Theorem

Let P be a program and O an oracle:

1. *apply the nullness analysis induced by O*
2. *collect the set O' of those fields $f \in O$, defined in some class κ, such that:*
   - *are always initialised in all constructors of κ (syntactical property)*
   - *and are always assigned in P to a non-`null` value (semantical property) according to the analysis above*
3. *call F_P that transformation. Hence $O' = F_P(O)$*

We have:

1. $O \supseteq F_P(O)$
2. *if $O = F_P(O)$ then O is correct*

Looking for a Correct Oracle

Corollary: Finding a correct oracle

Let $O = \{all\ fields\}$. Then

$$O \supseteq F_P(O) \supseteq F_P(F_P(O)) \supseteq F_P(F_P(F_P(O))) \supseteq \ldots$$

is a decreasing chain and converges to a correct oracle in a finite number of steps

Every application of F_P is a nullness analysis:

- the number of applications is bounded by the cardinality of the reference fields in the program. In practice, never more than 4 applications are needed to reach the fixpoint
- only the first application is (relatively) expensive. The others are fast thanks to caching

The Quest for Precision

The analysis described so far is relatively fast and proves around 85% of all dereferences safe in typical Java programs

Better precision is achieved with extra analyses that spot:

- fields/expressions that are locally non-null
- arrays that only contain non-null elements
- collections or maps that only contain/map non-null elements

We typically prove 98% of all dereferences safe then

None of these analyses uses logic, but they are based on formal methods

Bibliography

1. A. Tarski, *A Lattice-theoretical Fixpoint Theorem and its Applications*, Pacific J. Math. volume 5, pages 285-309, 1955
2. R. Giacobazzi & F. Scozzari, *A Logical Model for Relational Abstract Domains*, ACM TOPLAS 20(5), pages 1067-1109, 1998
3. F. Spoto, *Nullness Analysis in Boolean Form*, Software Engineering and Formal Methods (SEFM), pages 21-30, 2008
4. F. Spoto, *Precise null-Pointer Analysis*, Software and System Modeling, 10(2): pages 219-252, 2011
5. D. Nikolić & F. Spoto, *Inference of Class Invariants for Arrays*, submitted, 2011

Try it yourself online

`http://www.juliasoft.com`

`http://julia.scienze.univr.it/runs/android/results.html`

`http://julia.scienze.univr.it/runs/android2/results.html`

`http://julia.scienze.univr.it/runs/gwt/results.html`

Thanks

Thank you!

The Semantics of the Language: Programs

A *program* defines *classes* and *methods* inside those classes. Each method has the form

```
m(w_1, ..., w_n)
  execute C then return w
```

The Semantics of the Language: Method Calls

$$[\![\mathtt{v} := \mathtt{v}_0.\mathtt{m}(\mathtt{v}_1,\ldots,\mathtt{v}_\mathtt{n})]\!]\iota = \mathsf{plug}(\iota(C))$$

where

$$\begin{array}{l} \mathtt{m}(\mathtt{w}_1,\ldots,\mathtt{w}_\mathtt{n}) \\ \quad \mathtt{execute\ C\ then\ return\ w} \end{array}$$

$$\mathsf{plug}(\delta) = \left\{ \begin{array}{l} \langle\eta \,\|\, \mu\rangle \Rightarrow \langle\eta[\mathtt{v}\mapsto\eta'(\mathtt{w})] \,\|\, \mu'\rangle \\ \text{if } \delta(\langle\left[\begin{array}{c} \mathtt{this}\mapsto\eta(\mathtt{v}_0), \mathtt{w}_1\mapsto\eta(\mathtt{v}_1), \\ \ldots, \mathtt{w}_\mathtt{n}\mapsto\eta(\mathtt{v}_\mathtt{n}) \end{array}\right] \|\, \mu\rangle) = \langle\eta' \,\|\, \mu'\rangle \end{array} \right\}$$

$$\cup \left\{ \begin{array}{l} \langle\eta \,\|\, \mu\rangle \Rightarrow \underline{\langle\eta \,\|\, \mu'\rangle} \\ \text{if } \delta(\langle\left[\begin{array}{c} \mathtt{this}\mapsto\eta(\mathtt{v}_0), \mathtt{w}_1\mapsto\eta(\mathtt{v}_1), \\ \ldots, \mathtt{w}_\mathtt{n}\mapsto\eta(\mathtt{v}_\mathtt{n}) \end{array}\right] \|\, \mu\rangle) = \underline{\langle\eta' \,\|\, \mu'\rangle} \end{array} \right\}$$

The Semantics of the Language: Fixpoint Interpretation

Denotational Semantics

The denotational semantics of a program is the minimal fixpoint of the transformer of interpretation:

$$T(\iota) = C \Rightarrow [\![C]\!]\iota$$

(Tarksi'55)

We can compute it as the limit of the sequence

$$\begin{aligned}\iota_0 &= C \Rightarrow \emptyset \\ \iota_1 &= T(\iota_0) \\ \iota_2 &= T(\iota_1) \\ &\dots\end{aligned}$$