

# Verifying Pointer and String Analyses with Region Type Systems

Lennart Beringer, Robert Grabowski, Martin Hofmann

Ludwig-Maximilians-Universität, Munich, Germany  
Princeton University, Princeton, NJ

LPAR-16  
April 2010, Dakar

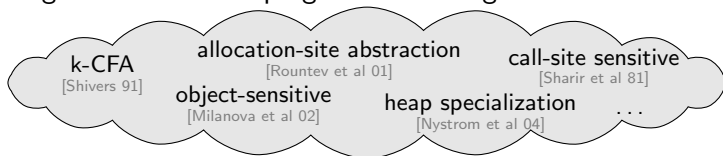
# Pointer analysis

analysis algorithms for Java programs obtaining non-alias relations:



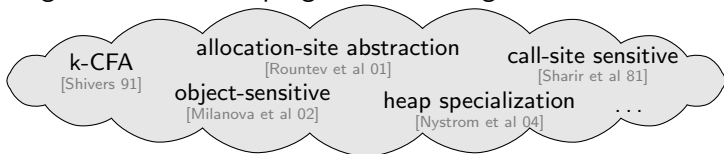
# Pointer analysis

analysis algorithms for Java programs obtaining non-alias relations:



# Pointer analysis

analysis algorithms for Java programs obtaining non-alias relations:



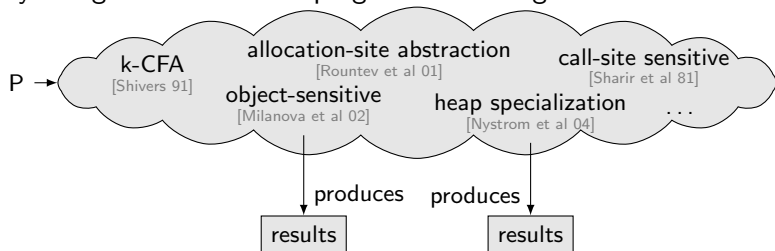
```
class List {  
  Elem elem; List next;  
  List copy() {  
    List l = new List();  
    l.elem = elem.clone();  
    l.next = next.copy();  
    return l;  
  }  
}
```

“x.copy() does not alias with x”

“x.copy().next does not alias with x.next”

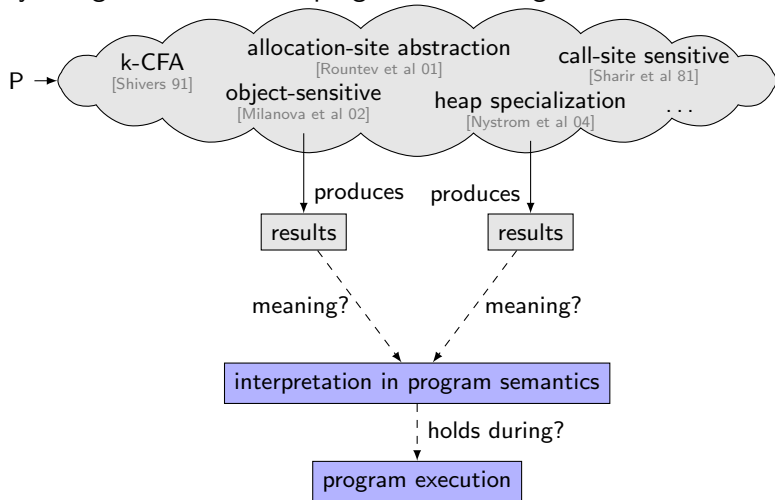
# Pointer analysis

analysis algorithms for Java programs obtaining non-alias relations:



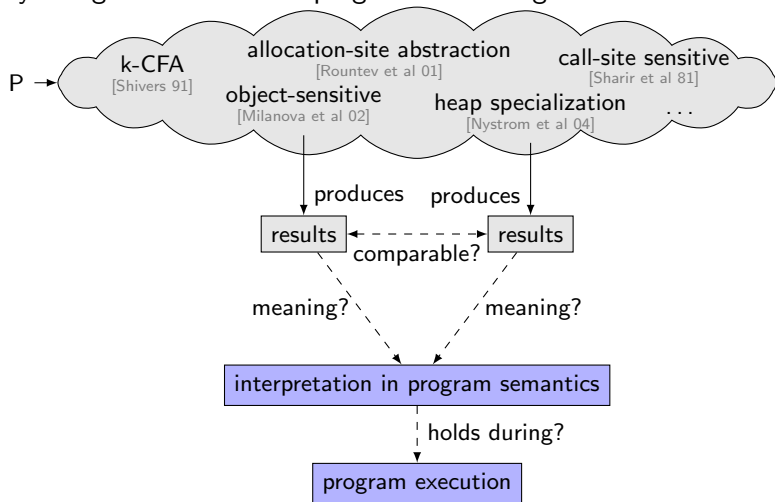
# Pointer analysis

analysis algorithms for Java programs obtaining non-alias relations:



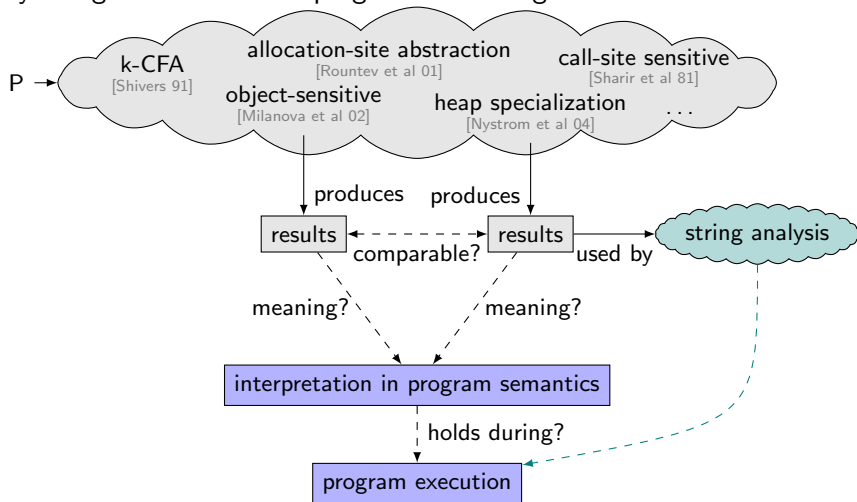
# Pointer analysis

analysis algorithms for Java programs obtaining non-alias relations:



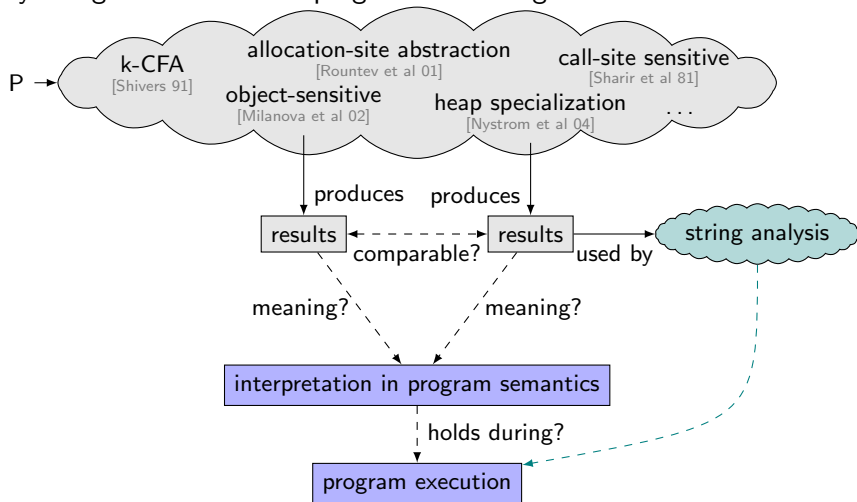
# Pointer analysis

analysis algorithms for Java programs obtaining non-alias relations:

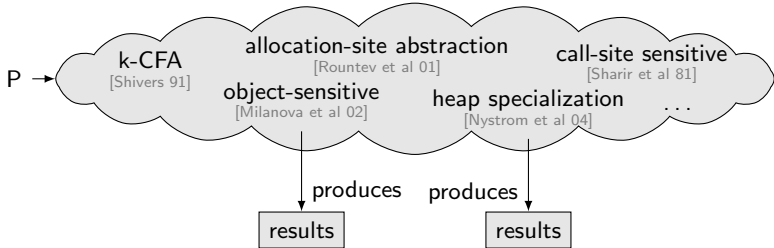


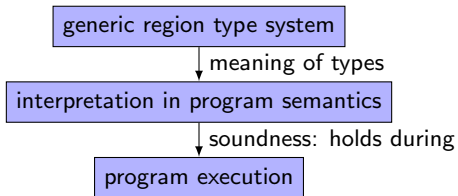
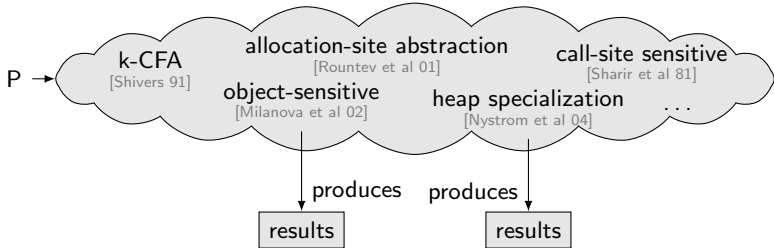
# Pointer analysis

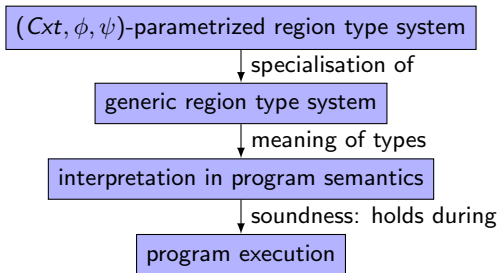
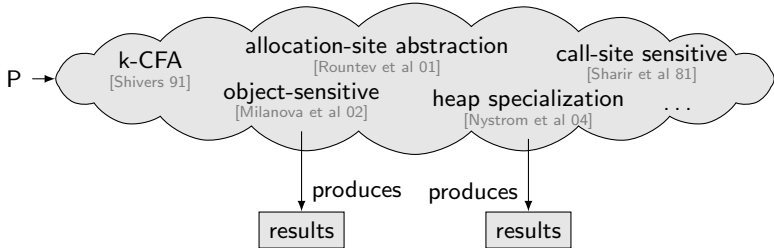
analysis algorithms for Java programs obtaining non-alias relations:

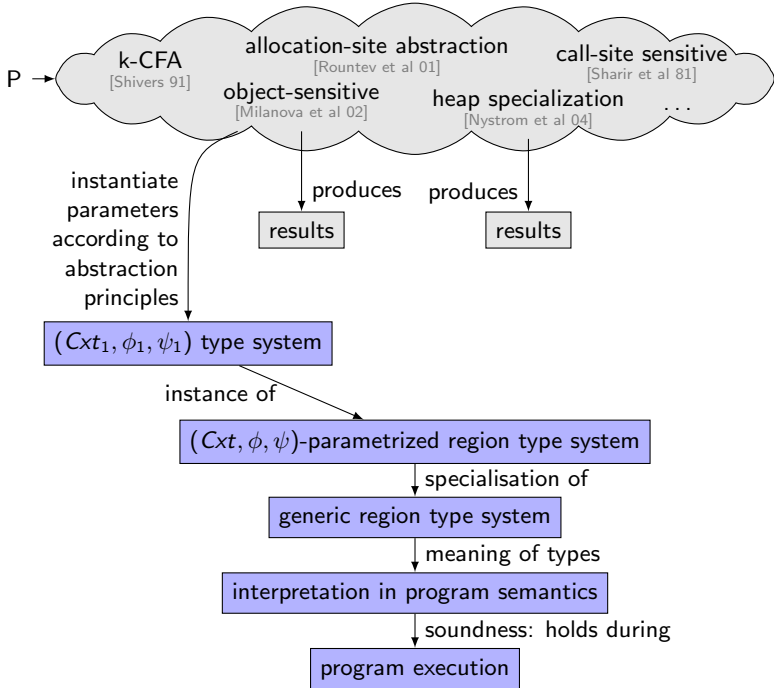


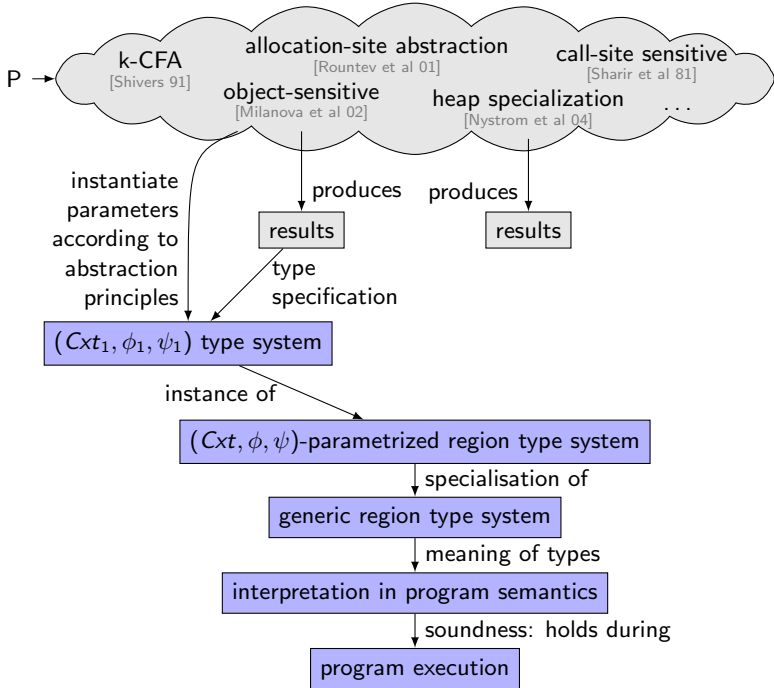
our goal: verify results of pointer analysis algorithms using region type system

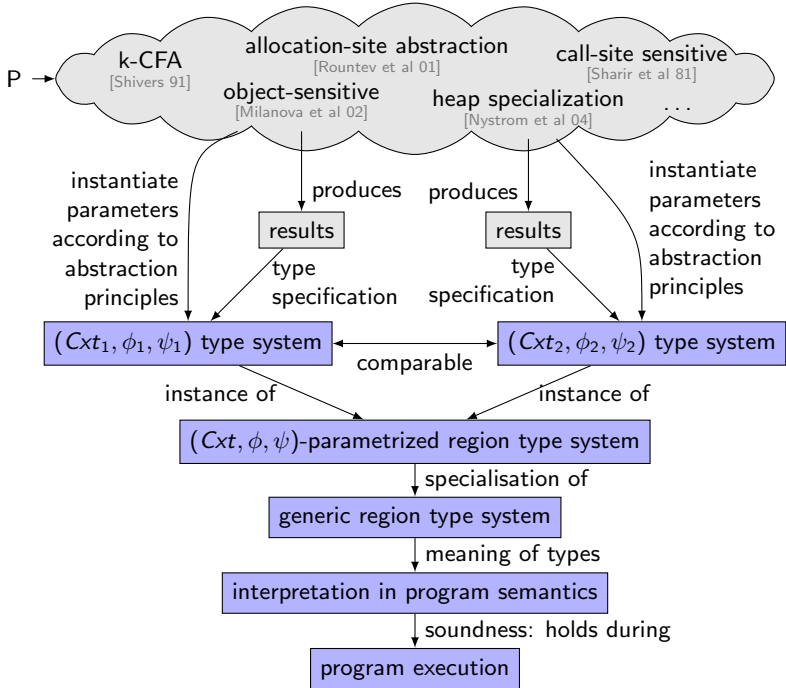


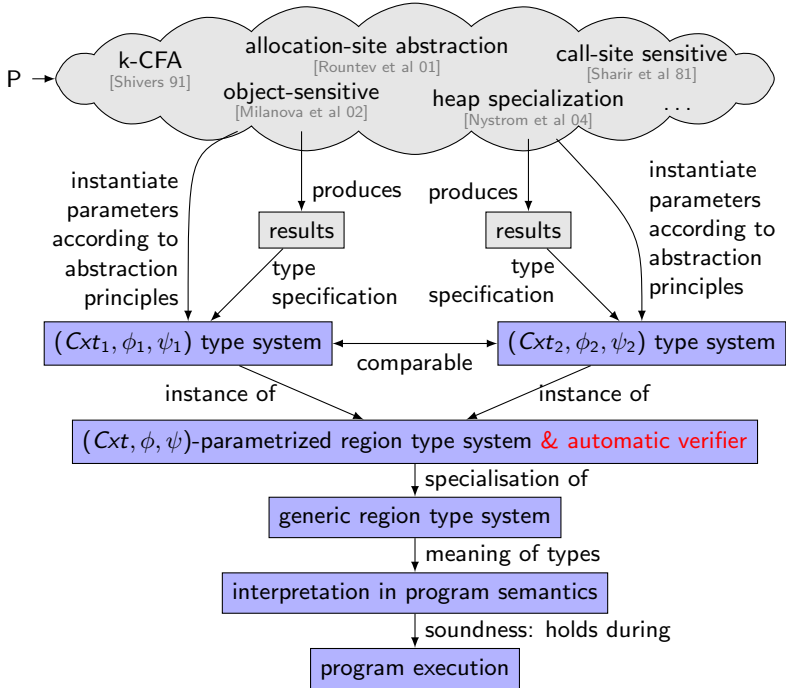


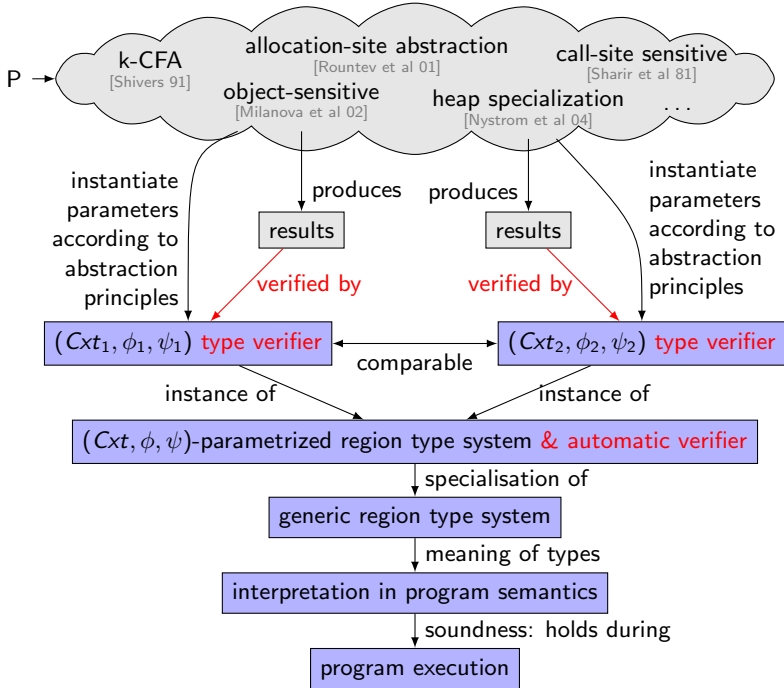


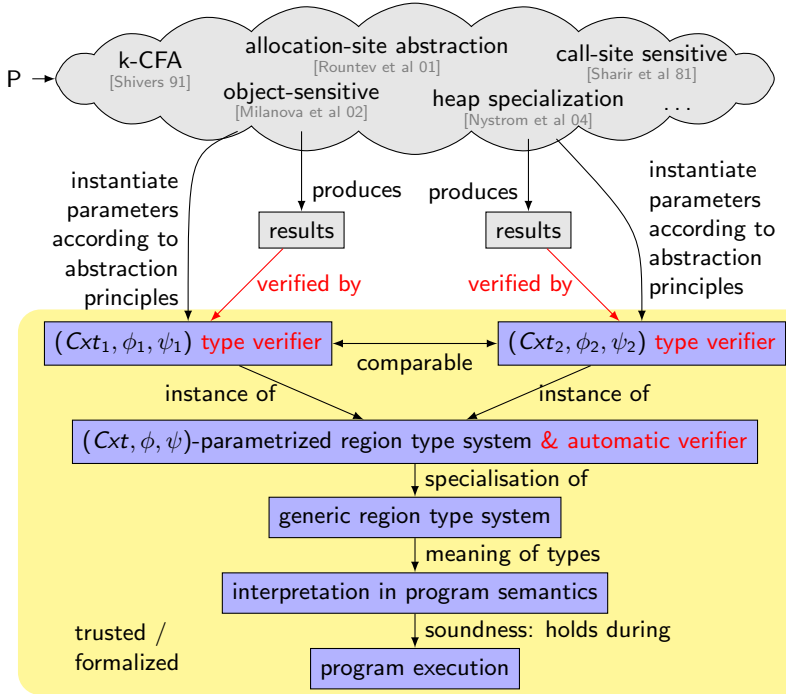


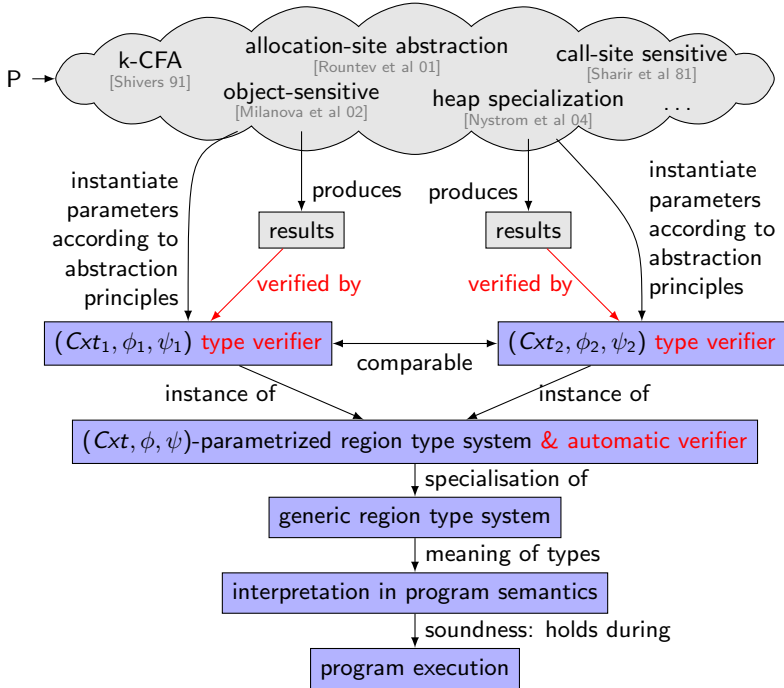


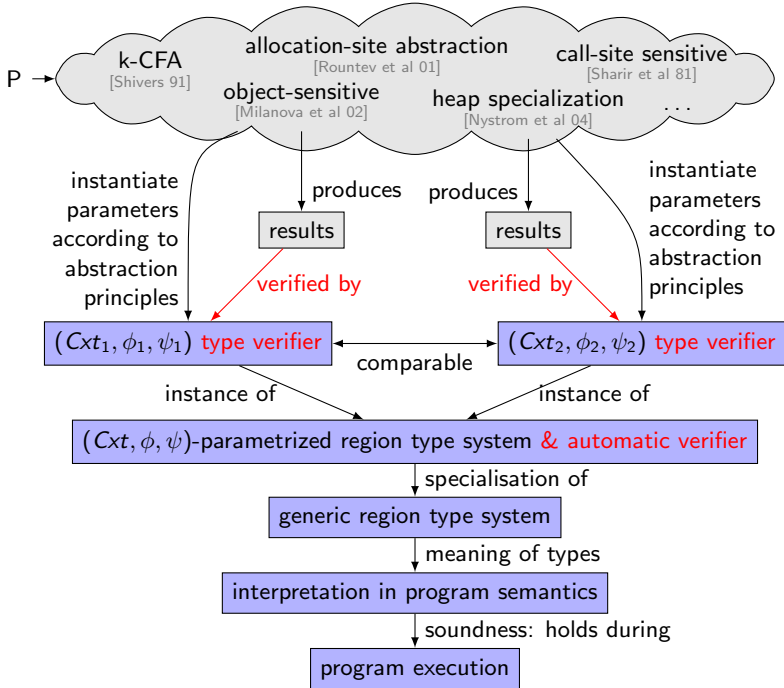












# Java class type system

formalization of program:

- sets of identifiers, field and method typing (types = classes)

$$\begin{aligned}Cls &= \{\text{List}, \text{Elem}\} \\Fld &= \{\text{elem}, \text{next}\} \\Mtd &= \{\text{copy}\}\end{aligned}$$
$$\begin{aligned}FType(\text{List}, \text{elem}) &= \text{Elem} \\FType(\text{List}, \text{next}) &= \text{List} \\MType(\text{List}, \text{copy}) &= () \rightarrow \text{List}\end{aligned}$$

# Java class type system

formalization of program:

- sets of identifiers, field and method typing (types = classes)

$$\begin{aligned}Cls &= \{\text{List}, \text{Elem}\} \\Fld &= \{\text{elem}, \text{next}\} \\Mtd &= \{\text{copy}\}\end{aligned}$$
$$\begin{aligned}FType(\text{List}, \text{elem}) &= \text{Elem} \\FType(\text{List}, \text{next}) &= \text{List} \\MType(\text{List}, \text{copy}) &= () \rightarrow \text{List}\end{aligned}$$

- method implementations in *Featherweight Java Extended with Updates*

# Java class type system

formalization of program:

- sets of identifiers, field and method typing (types = classes)

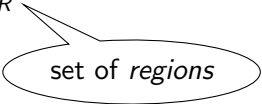
$$\begin{aligned}Cls &= \{\text{List}, \text{Elem}\} \\Fld &= \{\text{elem}, \text{next}\} \\Mtd &= \{\text{copy}\}\end{aligned}$$
$$\begin{aligned}FType(\text{List}, \text{elem}) &= \text{Elem} \\FType(\text{List}, \text{next}) &= \text{List} \\MType(\text{List}, \text{copy}) &= () \rightarrow \text{List}\end{aligned}$$

- method implementations in *Featherweight Java Extended with Updates*

type system assigns judgements:  $\Gamma \vdash e : C$

# Region types


Region type system assigns a *refined* type:

$$\Gamma \vdash e : C_R$$


set of *regions*

# Region types

Region type system assigns a *refined* type:

$$\Gamma \vdash e : C_R$$


set of *regions*

regions:

- intuition: disjoint classification of objects
- represent disjoint sets of possible concrete memory locations
- infinitely many regions  $r \in Reg$  available
- in type: location is in one of regions in  $R$
- different usage: scoping & garbage collection [Tofte et al. 97]

# Region type system

annotated class specification:

- types = classes with region sets
- differentiate field and method types for same class by region
- have (possibly infinite) set of types for each method

$$\begin{aligned} FType(\text{List}, r, \text{elem}) &= \text{Elem}_{\{t\}} \\ FType(\text{List}, r, \text{next}) &= \text{List}_{\{r\}} \\ MType(\text{List}, r, \text{copy}) &= \{() \longrightarrow \text{List}_{\{s\}}, \\ &\quad () \longrightarrow \text{List}_{\{r\}}\} \end{aligned}$$
  
$$\begin{aligned} FType(\text{List}, s, \text{elem}) &= \text{Elem}_{\{t\}} \\ FType(\text{List}, s, \text{next}) &= \text{List}_{\{s\}} \\ MType(\text{List}, s, \text{copy}) &= \{() \longrightarrow \text{List}_{\{s\}}\} \end{aligned}$$

# Region type system

annotated class specification:

- types = classes with region sets
- differentiate field and method types for same class by region
- have (possibly infinite) set of types for each method

$$\begin{aligned} FType(\text{List}, r, \text{elem}) &= \text{Elem}_{\{t\}} \\ FType(\text{List}, r, \text{next}) &= \text{List}_{\{r\}} \\ MType(\text{List}, r, \text{copy}) &= \{() \longrightarrow \text{List}_{\{s\}}, \\ &\quad () \longrightarrow \text{List}_{\{r\}}\} \end{aligned}$$
  
$$\begin{aligned} FType(\text{List}, s, \text{elem}) &= \text{Elem}_{\{t\}} \\ FType(\text{List}, s, \text{next}) &= \text{List}_{\{s\}} \\ MType(\text{List}, s, \text{copy}) &= \{() \longrightarrow \text{List}_{\{s\}}\} \end{aligned}$$

given  $FType$  and  $MType$ :

- ① when are they legal?
- ② how to determine it automatically?
- ③ relation to pointer analysis results?

## Typing rules (extract)

Object creation: can pick an arbitrary region for new objects

$$\frac{}{\Gamma \vdash \text{new } C : C_{\{r\}}}$$

## Typing rules (extract)

Object creation: can pick an arbitrary region for new objects

$$\overline{\Gamma \vdash \text{new } C : C_{\{r\}}}$$

Method call: for all regions that may contain  $x$ , there must be a suitable method type:

$$\frac{\forall r \in R. \exists (\overline{\tau}_{arg}', \tau'_{res}) \in MType(C, r, m). \overline{\tau}_{arg} <: \overline{\tau}_{arg}' \wedge \tau'_{res} <: \tau_{res}}{x : C_R, \overline{y} : \overline{\tau}_{arg} \vdash x.m(\overline{y}) : \tau_{res}}$$

## Typing rules (extract)

Object creation: can pick an arbitrary region for new objects

$$\overline{\Gamma \vdash \text{new } C : C_{\{r\}}}$$

Method call: for all regions that may contain  $x$ , there must be a suitable method type:

$$\frac{\forall r \in R. \exists (\overline{\tau_{arg}'}, \tau'_{res}) \in MType(C, r, m). \overline{\tau_{arg}} <: \overline{\tau_{arg}'} \wedge \tau'_{res} <: \tau_{res}}{x : C_R, \overline{y} : \overline{\tau_{arg}} \vdash x.m(\overline{y}) : \tau_{res}}$$

*Well-typed* program: for all announced method types  $MType(C, r, m)$ , a corresponding type for the method body can be derived.

# Soundness

Given annotated tables  $FType$  and  $MType$ . If

- $\Gamma \vdash e_1 : C_R$  and
- $\Gamma \vdash e_2 : D_S$  and
- $R \cap S = \emptyset$

then

- $e_1$  and  $e_2$  do not alias, i.e. are different objects

# Soundness

Given annotated tables  $FType$  and  $MType$ . If

- $\Gamma \vdash e_1 : C_R$  and
- $\Gamma \vdash e_2 : D_S$  and
- $R \cap S = \emptyset$

then

- $e_1$  and  $e_2$  do not alias, i.e. are different objects
  - more precisely: there is a mapping from locations to regions

# Soundness

Given annotated tables  $FType$  and  $MType$ . If

- $\Gamma \vdash e_1 : C_R$  and
- $\Gamma \vdash e_2 : D_S$  and
- $R \cap S = \emptyset$

then

- $e_1$  and  $e_2$  do not alias, i.e. are different objects
  - more precisely: there is a mapping from locations to regions
  - really precise: Isabelle formalization of proofs

# Towards verification of pointer analysis results

- ② how to determine type derivation automatically?
- ③ relation to pointer analysis results?

Solution: parametrized version of the type system

Obeys principles used by algorithm with respect to

- label choice at object creation
- behavior at call sites

# Abstraction principles

## Call-graph abstraction

Context-sensitive pointer analysis distinguishes different calls to a method via its calling *context*, which parametrizes obtained pointer information of each method:

- invocation site
- region of receiver object
- (finite) stack of these

## Object abstraction

Variations exist how labels are chosen to distinguish objects:

- object allocation site
- context during allocation
- combination of both

# Parametrized type system

We capture these principles by four parameters  $PP$ ,  $Cxt$ ,  $\psi$  and  $\phi$ .

$PP \ni i$  is a distinct labeling of program points:

$$e^i$$

$Cxt \ni z$  is a finite set of contexts. They are used to parametrize typing judgements:

$$\Gamma; z \vdash e^i : \tau$$

The function  $\psi$  models object abstraction:

$$\psi : Cxt \times PP \rightarrow Reg$$

It determines the region of the object in the rule for `new`  $C$ .

## Parametrized type system

We assign just one method type  $MType(C, r, m, z)$  per  $z \in \mathcal{Z}$ :

$$MType : Cls \times Reg \times Mtd \times Cxt \rightarrow \overline{Typ} \times Typ$$

The transfer function  $\phi$  models context switch:

$$\phi : Cxt \times Cls \times Reg \times Mtd \times PP \rightarrow Cxt$$

The function selects the method type in the rule for  $x.m(\bar{y})$ .

## Instantiation of parameters

<i>Reg=</i>	<i>object abstraction</i>	<i>principle</i>
<i>PP</i>	$\psi(z, i) = i$	allocation site abstraction
$PP \times PP = Cxt$	$\psi((i_1, i_2), i_0) = (i_0, i_1)$	object-sensitive allocation site abstraction
$PP = Cxt$	$\psi(i_c, i) = i_c$	heap specialization

<i>Cxt=</i>	<i>context transfer</i>	<i>principle</i>
$\{z_0\}$	$\phi(z, C, r, m, i) = z_0$	context-insensitivity
<i>Reg</i>	$\phi(z, C, r, m, i) = r$	object-sensitive 1-CFA
$\bigcup_{n \in \{1, \dots, k\}} Mtd^n$	$\phi(z, C, r, m, i) = (m :: z) _k$	method identifier <i>k</i> -CFA

# Instantiation of parameters

<i>Reg=</i>	<i>object abstraction</i>	<i>principle</i>
<i>PP</i>	$\psi(z, i) = i$	allocation site abstraction
$PP \times PP = Cxt$	$\psi((i_1, i_2), i_0) = (i_0, i_1)$	object-sensitive allocation site abstraction
$PP = Cxt$	$\psi(i_c, i) = i_c$	heap specialization

<i>Cxt=</i>	<i>context transfer</i>	<i>principle</i>
$\{z_0\}$	$\phi(z, C, r, m, i) = z_0$	context-insensitivity
<i>Reg</i>	$\phi(z, C, r, m, i) = r$	object-sensitive 1-CFA
$\bigcup_{n \in \{1, \dots, k\}} Mtd^n$	$\phi(z, C, r, m, i) = (m :: z) _k$	method identifier <i>k</i> -CFA

stack restricted to *k* entries

# Soundness and automatic verification

## Soundness

parametrized type system is more restrictive than generic region type system

- bounded number of regions and method types
- $\psi$  and  $\phi$  constrain choice of region and method type
- also simplifies finding type derivation

## Automatic verification

in the paper: syntax-directed algorithmic version

- derivable in parametrized system
- suggests a type checker
- results of pointer analysis can be automatically verified once abstraction principles have been suitably transformed into parameter instantiations

# Verifying string analysis

String analysis:

- determine possible values of string variables
- used to enforce a security policy, i.e. prevention of code injection and cross-site scripting attacks

Can be built on top of pointer analysis [Cregut et al. '05]:

- use regions to track objects of class *String*
- produce string value assertions in terms of regions, e.g. “all strings in region  $r$  only contain numbers”

We have extended language and type system for string analysis:

- extensional formalization of grammar-based string analysis, similar to Java String Analyzer [Christensen et al. '03]

# Summary

## Contribution:

- framework for semantic interpretation of results of common pointer analysis algorithms
- region type system to declaratively describe pointer assertions
- transformation into algorithmic version that enables embedding and automatic verification of results of a specific analysis

## Future work:

- more powerful type system: flow-/path-sensitivity
- sound enforcement of string-based security policies and “best practices” using type systems

# Backup slide: Featherweight Java Extended with Updates

- FJEU [Hofmann et al. '06]; based on FJ [Igarashi et al. '99]
- idealized object-oriented language, models Java's core

## Syntax

$$e ::= \text{null} \mid x \mid \text{let } x = e_1 \text{ in } e_2 \mid \text{if } x = y \text{ then } e_1 \text{ else } e_2 \mid \\ \text{new } C \mid x.f \mid x.f := y \mid x.m(\bar{y})$$

A program consists of a class hierarchy  $(Cls, \preceq)$ , for each class  $C \in Cls$  field names  $f \in Fld$  and method names  $m \in Mtd$ , and for each method a body  $e$ .

$$\begin{array}{ll} Cls & = \{List, Elem\} \\ fields(List) & = \{elem, next\} \\ methods(List) & = \{copy\} \end{array} \quad \begin{array}{l} mbody(List, copy) = \\ \text{let } l = \text{new List in} \\ \text{let } _ = l.elem := elem \text{ in} \\ \text{let } _ = l.next := next.copy() \text{ in } l \end{array}$$

## Operational semantics

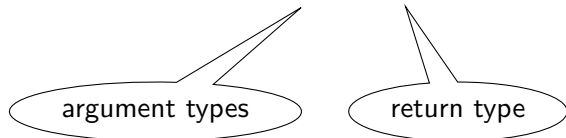
$$(s, h) \vdash e \Downarrow v, h'$$

## Backup slide: Region types

region:  $r, s, t, \dots \in \text{Reg}$   
sets of region:  $R, S, T, \dots \in \wp(\text{Reg})$   
types:  $\tau \in \text{Typ} = \text{Cls} \times \wp(\text{Reg})$

For a type  $(C, R)$ , we write  $C_R$ .

$\Gamma : \text{Var} \rightarrow \text{Typ}$   
 $F\text{Type} : \text{Cls} \times \text{Reg} \times \text{Fld} \rightarrow \text{Typ}$   
 $M\text{Type} : \text{Cls} \times \text{Reg} \times \text{Mtd} \rightarrow \wp(\overline{\text{Typ}} \times \text{Typ})$



## Backup slide: Region type system

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \quad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}$$

$$\frac{}{\Gamma \vdash \text{new } C : C_{\{r\}}} \quad \frac{}{\Gamma, x : \tau \vdash x : \tau} \quad \frac{}{\Gamma \vdash \text{null} : \tau}$$

$$\frac{\Gamma, x : C_{RNS}, y : D_{RNS} \vdash e_1 : \tau \quad \Gamma, x : C_R, y : D_S \vdash e_2 : \tau}{\Gamma, x : C_R, y : D_S \vdash \text{if } x = y \text{ then } e_1 \text{ else } e_2 : \tau}$$

$$\frac{\forall r \in R. \exists (\bar{\sigma}', \tau') \in MType(C, r, m). \bar{\sigma} <: \bar{\sigma}' \wedge \tau' <: \tau}{\Gamma, x : C_R, \bar{y} : \bar{\sigma} \vdash x.m(\bar{y}) : \tau}$$

$$\frac{\forall r \in R. FType(C, r, f) <: \tau}{\Gamma, x : C_R \vdash x.f : \tau} \quad \frac{\forall r \in R. \tau <: FType(C, r, f)}{\Gamma, x : C_R, y : \tau \vdash x.f := y : \tau}$$

## Backup slide: Soundness result

Locations:  $l \in Loc$

Heap typings:  $\Sigma \in Loc \rightarrow Reg \times Cls$

$$\frac{}{\Sigma \vdash \text{null} : \tau} \quad \frac{\Sigma(l) = (C, r)}{\Sigma \vdash l : C_{\{r\}}} \quad \frac{\Sigma \vdash v : \sigma \quad \sigma <: \tau}{\Sigma \vdash v : \tau}$$

$$\Sigma \vdash s : \Gamma \iff \forall x \in \text{dom}(\Gamma). \Sigma \vdash s(x) : \Gamma(x)$$

$$h \models \Sigma \iff \forall l \in \text{dom}(\Sigma). l \in \text{dom}(h) \wedge \Sigma \models h(l) : \Sigma(l)$$

where

$$\Sigma \models (C, F) : (D, r) \iff C \preceq D \wedge \text{dom}(F) = \text{fields}(C) \wedge \\ \forall f \in \text{fields}(C). \Sigma \vdash F(f) : F\text{Type}(C, r, f)$$

### Soundness Theorem

Fix a well-typed program  $P$ . For all  $\Sigma, \Sigma', \tau, s, h, e, v, h'$  with  $\Gamma \vdash e : \tau$  and  $\Sigma \vdash s : \Gamma$  and  $(s, h) \vdash e \Downarrow v, h'$  and  $h \models \Sigma$  there exists some  $\Sigma' \sqsupset \Sigma$  such that  $\Sigma' \vdash v : \tau$  and  $h' \models \Sigma'$ .

## Backup slide: String analysis

### Extended syntax

$$w \in \Sigma^*$$
$$e ::= \dots \mid \text{new String}(w) \mid x.\text{concat}(y)$$

### String operation contexts

$\Omega$ : set over string operations:

$$\omega ::= \text{newstr } w r \mid \text{concat } r s t \mid \text{unknown } W r$$

Types:  $C_R$  or  $\text{String}_R$

$$\frac{\text{newstr } w r \in \Omega}{\Gamma \vdash \text{new String}(w) : \text{String}_{\{r\}}}$$
$$\frac{\forall r \in R, s \in S. \text{concat } r s t \in \Omega}{\Gamma, x : \text{String}_R, y : \text{String}_S \vdash x.\text{concat}(y) : \text{String}_{\{t\}}}$$