Inferring Region Types via an Abstract Notion of Environment Transformation

Chuangjie Xu (j.w.w. Ulrich Schöpp)

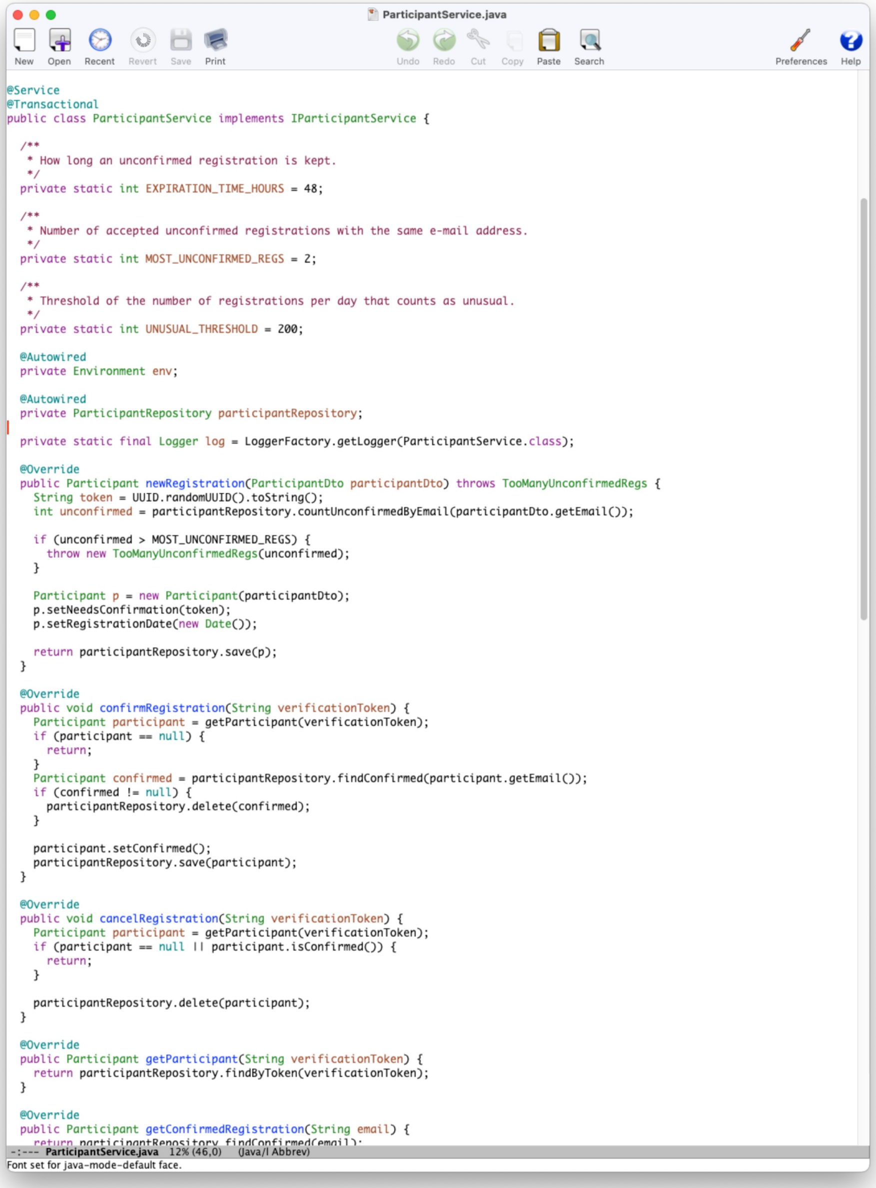
APLAS'22, 5 Dec 2022, Auckland, New Zealand

Enforcing Secure Programming Guidelines

- Does my Java program follow secure programming guidelines such as
  - *“All inputs must be sanitized.”*
  - *“Any access to sensitive data must be authorized.”*
  - *“Any access to sensitive data must be logged.”*
  - …

- Can guidelines be verified continuously and incrementally?

- A lightweight tool for this can help programmers to avoid making typical errors during the development.



GuideForce[1]

GuideForce develops effect type systems for lightweight static analysis.

- Trace properties of programs
  - Functions of interests emit events, e.g.:
    `Server.login()` emits a *login* event; `Connection.close()` emits a *close* event
  - Each execution of a program generates a (finite or infinite) trace of events.
  - A guideline = a set of allowed event traces
- The type system has effect annotations to give information about the possible traces.
  - E.g., `login() ? readData() : close();` : `type & {`*login read*`,` *login close*`}`
- Type inference ⇒ effect computation
- If "effect ⊆ guideline", then ✅ the program adheres to the guideline.

[1] DFG project number 250888164

Key Concepts

- Effect Annotations

  Capture information of terminating and nonterminating runs modularly in the type system.

```
verifyAuthorization() … & {auth} , ∅
readSensitiveData() … & {access} , ∅
LogAccess() … & {log} , ∅
serve() … & {auth, auth access}* · {log} , {auth, auth access}^ω
```

- Region Types

  Represent properties of values such as provenance information:

  `Null | CreatedAt(l) | Unknown | Tainted | Untainted | ...`

  They improve the precision of trace-property analysis:

  - Objects in different regions are analyzed separately
  - Method with inputs in different regions has different effects

Type Inference in the Previous Work

- Redundant analysis

```
...
object1.foo(input1);
...
object1.foo(input2);
...
object2.foo(input3);
...
```

```
void foo(C d) {
   ...
   ...
   ...
}
```

In [GHL12, BGH13, EHZ17, ESX21], the code of the method `foo` is analyzed multiple times, one for each invocation if the objects and inputs are in different regions.

- Goal: A type inference algorithm that analyzes the code only once.

Idea of Our New Type Inference Algorithm

```
x = y.f;        ........ 
y = new C();    ........ 
y.f = x         ........ 
```

$[x :\mapsto y.f]$

$[y :\mapsto C]$

$[y.f :\geq x]$

$$env = (y : A,\ A.f : B)$$

$$\downarrow$$

$$\sigma = [x :\mapsto y.f,\ y :\mapsto C,\ C.f :\geq y.f]$$

$$\downarrow$$

$$\sigma(env) = (x : B,\ y : C,\ \boxed{A.f : B},\ C.f : B)$$

Generate a summary to each method – abstract environment transformation

- Equality constraint $x :\mapsto y.f$ for variable assignment $x = y.f$
- Subtyping constraints $y.f :\geq x$ for field assignment $y.f = x$ (weak update for fields)
- Composition and join – constraint generation
- Instantiation $env \mapsto \sigma(env)$ – constraint solving
- Get the return type of the method from the updated environment $\sigma(env)$

Types, Environments and Constraints

- Assume a finite set of atomic types $Typ = \{A, B, C, \ldots\}$, and call a set of atomic types a type.
  - Write $\perp$ to denote the empty set
  - Write A to denote the singleton $\{A\}$
  - Write $A \vee B \vee C$ to denote the set $\{A, B, C\}$
- A typing environment is a mapping $Var \cup Typ \times Fld \rightharpoonup \mathcal{P}(Typ)$
  - E.g., $(x: A,\ A.f: B \vee C)$
  - $A.f$ represents the field f of any object of type A
- Possible value v of a constraint $x :\mapsto v$ or $y.f :\geq v$ can be
  - a variable x,
  - a type A,
  - a field access path $x.f.g.h$ or $A.f.g$ (?)
  - a set containing any of the above $x \vee A \vee y.g$

Access Graphs

- The lengths of access paths may be unbounded. Consider

```
Node last() {
  if (next == null) {return this;}
  else {return next.last();}
}
```

  The return type can be the same of `this`, `this.next`, `this.next.next`, ...

- Use finite representation of access paths, such as **access graphs**:

E.g.,

this.ℰ = { this }

this.<next,Ø,next> = { this.next }

this.<next,{(next,next)},next> = { this.next.next, this.next.next.next,, … }

Abstract Environment Transformations

- Constraints
  - We call $\bigvee b_i.G_i$ a term, where $b_i \in Var \cup Typ$ and G a field graph
  - $x :\mapsto \bigvee b_i.G_i$
  - $a.G :\geq \bigvee b_i.G_i$ where $a \in Var \cup Typ$ and G nonempty
- Abstract transformation $[x_1 :\mapsto u_1, \ \dots, \ x_n :\mapsto u_n, \ b_1.G_1 :\geq v_1, \ \dots, \ b_m.G_m :\geq v_m]$
  - All the keys x_i and $b_j.G_j$ are different
  - $u_i \neq x_i$
  - $v_i \neq \bot$
- Example:

$[x :\mapsto y.f, y :\mapsto C, C.f :\geq y.f]$ for the code

```
x = y.f;
y = new C();
y.f = x
```

Operations on Abstract Transformations

- Instantiation $env \mapsto \sigma(env)$
  - A least fixed-point algorithm to solve constraints
  - Computing reachable fields $A.f$ in access graphs $B.\langle h, E, f\rangle$
- Composition $\sigma\theta$
  - Variable substitution, essentially
  - E.g., $[x.f :\mapsto x \vee y.g][x :\mapsto C] = [C.f :\mapsto C \vee y.g,\ x :\mapsto C]$
- Join $\sigma \vee \theta$
  - Pointwise defined
  - E.g., $[x :\mapsto C] \vee [x :\mapsto D,\ y :\mapsto z] = [x :\mapsto C \vee D,\ y :\mapsto y \vee z]$

Theorem:

- $\sigma\big(\theta(env)\big) \sqsubseteq (\sigma\theta)(env)$
- $\sigma(env) \sqcup \theta(env) \sqsubseteq (\sigma \vee \theta)(env)$

Type Inference via Abstract Transformations

We work with some region type system on Featherweight Java and choose Typ to be the set of regions.

Step 1: Compute an abstract method table $T: Cls \times Mtd \rightharpoonup ATrans \times Tm$

- $T(C, m) = (\sigma, u)$: summary of method m to use in the analysis (Step 2)
  - The transformation σ captures the change of types in m
  - The term u will be instantiated to a return type of m
- T is compute via a fixed-point algorithm using $[\![e]\!]: ATrans \times Tm$

$$[\![\texttt{if } x = y \texttt{ then } e_1 \texttt{ else } e_2]\!] := [\![e_1]\!] \vee [\![e_2]\!]$$

$$[\![\texttt{let } x = e_1 \texttt{ in } e_2]\!] := [\![e_2]\!]([x :\mapsto t]\theta) \quad \text{where } (\theta, t) = [\![e_1]\!]$$

...

$$[\![x.f := y]\!] := ([x.f :\geq y], y)$$

$$[\![x^C.m(\bar{y})]\!] := T(C, m)[\texttt{this} :\mapsto x, \mathit{args}(C, m) :\mapsto \bar{y}]$$

Type Inference via Abstract Transformations

Step 2: Use T to compute the method and field typing of the program

For example, we compute the type of a method m of class C as follows:

- Suppose the object where m lives is in region r and the inputs of m are in regions $s_1, \dots, s_n$
- $(\sigma, u) = T(C, m)$
  - The transformation σ captures the change of types in m
  - The term u will be instantiated to a return type of m
- Update the environment $env = \sigma(\text{this}: r,\ x_1: s_1,\ \dots,\ x_n: s_n,\ \dots)$ where $x_1, \dots, x_n$ are the arguments of m
- Instantiate $u[env]$ to get the return type of m

Conclusion and Discussion

- We introduce a theory of **abstract environment transformations** to summarize how types are changed in a program

- We introduce an **inference algorithm** to compute region information of (Featherweight) Java programs which can **avoid redundant code analysis**.

- The inference algorithm can be extended to a more efficient region-sensitive analysis of trace properties.

Thank you!