Programming Assignment 6:

Context-Sensitive Pointer Analysis

Course “Static Program Analysis” @Nanjing University
Assignments Designed by Tian Tan and Yue Li

1 Assignment Objectives

- Implement a context-sensitive pointer analysis framework for Java.
- Implement on-the-fly call graph construction as part of pointer analysis.
- Implement common context sensitivity variants.

In this programming assignment, you will implement a context-sensitive pointer analysis framework for Java on top of Tai-e. The pointer analysis builds a call graph on the fly. To make the framework work, you also need to implement concrete context sensitivity variants. If your implementation is correct, you can observe that context-sensitive pointer analysis can build more precise call graphs than context-insensitive pointer analysis (CIPTA), and different context sensitivity variants would exhibit different precision as described later.

Similar to the previous assignment, in this assignment, we will teach you how to handle the Java features that are not covered in the lectures, i.e., static field, array, and static method, in a context-sensitive setting, so that you will implement a context-sensitive pointer analysis that can handle all kinds of pointers in Java.

2 Implementing Context-Sensitive Pointer Analysis

2.1 Scope

You will implement the context-sensitive pointer analysis algorithm introduced in Lectures 11 and 12, and the algorithm handles two kinds of pointers, i.e., local variables and instance fields, as well as instance method calls. To achieve a more practical pointer analysis, you need to handle the other two kinds of pointers, i.e., static fields and array indexes, as well as static method calls in this assignment. We will introduce the analysis rules to handle these features in Section 2.2. These rules are very similar to (or even simpler than) the rules you have learnt in the lectures, and you need to figure out how to implement them based on the pointer analysis algorithm.

2.2 New Rules

In this section, we introduce new context-sensitive pointer analysis rules to handle static fields, array indexes, and static method calls.

Static Fields. The handling of static fields is simple, i.e., we just need to pass values between the static field and the variable (with a context). We use $T.f$ to denote the pointer for static field `T.f`, and then define the rules to handle static field stores and loads as follows:

Kind	Statement	Rule (under context c)	PFG Edge
Static Store	`T.f = y`	$\frac{c':o_i \in pt(c:y)}{c':o_i \in pt(T.f)}$	$T.f \leftarrow c:y$
Static Load	`y = T.f`	$\frac{c':o_i \in pt(T.f)}{c':o_i \in pt(c:y)}$	$c:y \leftarrow T.f$

Array Indexes. As explained in Lecture 8, regular pointer analysis does not distinguish between loads and stores to different array indexes (locations). Suppose that $c':o_i$ represents an array object (with a heap context c'), then we use $c':o_i[*]$ to denote the pointer which points to all elements that are stored in any indexes of the array. Based on such treatment, we define the rules to handle array stores and loads as follows:

Kind	Statement	Rule (under context c)	PFG Edge
Array Store	`x[i] = y`	$\frac{c':o_i \in pt(c:x), c'':o_j \in pt(c:y)}{c'':o_j \in pt(c':o_i[*])}$	$c':o_i[*] \leftarrow c:y$
Array Load	`y = x[i]`	$\frac{c':o_i \in pt(c:x), c'':o_j \in pt(c':o_i[*])}{c'':o_j \in pt(c:y)}$	$c:y \leftarrow c':o_i[*]$

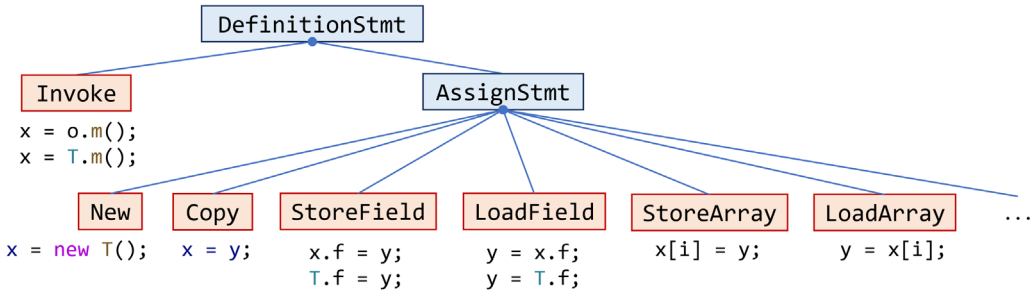
Static Methods. In context-sensitive pointer analysis, the handing of static methods is also the same as that of instance methods, except that 1) we do not need to dispatch on the receiver object to resolve the callee, and 2) we do not need to pass the receiver object. Since static method calls do not require receiver object, its treatment is simpler than that of instance method calls as shown below:

Kind	Statement	Rule (under context c)	PFG Edge
Static Call	l: `r = T.m(a1,…,an)`	$\frac{\begin{array}{c}c^t = \text{Select}(c, l) \\ c':o_u \in pt(c:aj), 1 \le j \le n \\ c'':o_v \in pt(c^t:m_{ret})\end{array}}{\begin{array}{c}c':o_u \in pt(c^t:m_{pj}),\ 1 \le j \le n \\ c'':o_v \in pt(c:r)\end{array}}$	$\begin{array}{c}c:a1 \rightarrow c^t:m^{p1} \\ \dots \\ c:an \rightarrow c^t:m_{pn} \\ c:r \leftarrow c^t:m_{ret}\end{array}$

2.3 Tai-e Classes You Need to Know

In this assignment, you will use some classes that were introduced in the previous assignment. Specifically, to implement context-sensitive pointer analysis, you need to use the same IR classes as the previous assignment, i.e., `IR`, `Var`, `InvokeExp`, and

these subclasses of `DefinitionStmt`:

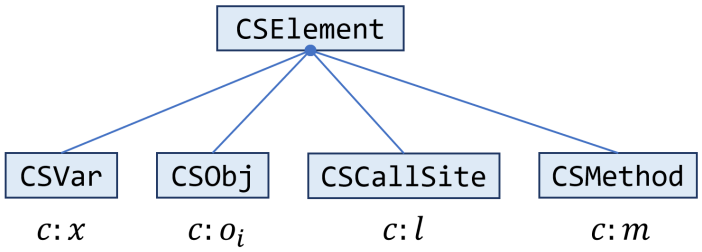


Besides, `JMethod`, `JField`, `Obj`, and `HeapModel` will also be used in this assignment.

Below we introduce the classes that are specific to this assignment, i.e., context-sensitive pointer analysis. These classes are generally simple, and most of them are very similar to the corresponding classes you have used in the previous assignment, except that they bring context information. You should read their source code and comments to decide how to use them.

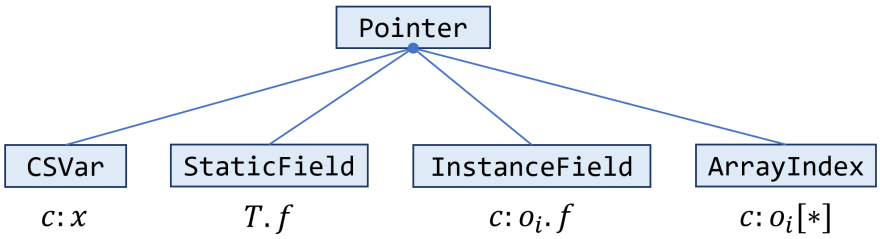
- `pascal.taie.analysis.pta.core.cs.context.Context`
  This class represents contexts in context-sensitive pointer analysis.

- `pascal.taie.analysis.pta.core.cs.element.CSElement`
  This class represents context-sensitive elements in pointer analysis. Each element is associated with a context. It has four subclasses, representing four kinds of context-sensitive elements, as shown below:



  - `CSVar`: represents a variable (`Var`) with a context (`Context`).
  - `CSObj`: represents an abstract object (`Obj`) with a context (`Context`).
  - `CSCallSite`: represents a call site (`Invoke`) with a context (`Context`).
  - `CSMethod`: represents a method (`JMethod`) with a context (`Context`).

- `pascal.taie.analysis.pta.core.cs.element.Pointer`
  This class represents the pointers in context-sensitive pointer analysis, i.e., the nodes in the PFG (pointer flow graph). Similar to `*.ci.Pointer` (used in the previous assignment), each `Pointer` is associated with a `PointsToSet`, which can be obtained by calling `getPointsToSet()`. This class also has four subclasses:



which corresponds to the four kinds of pointers (in a context-sensitive setting) in Java as introduced in page 82 of slides for Lecture 8.

- `pascal.taie.analysis.pta.core.cs.element.CSManager`
  This class manages *all* context-sensitive elements and *all* context-sensitive pointers, i.e., you should obtain the instances of all subclasses of `CSElement` and `Pointer` via APIs of this class.

- `pascal.taie.analysis.pta.core.cs.selector.ContextSelector`
  This class is the interface between context-sensitive pointer analysis framework and concrete context sensitivity variants (e.g., call-site sensitivity and object sensitivity). It has 4 APIs, and one of them returns an empty context and the others select contexts for static methods, instance methods, and heap objects, respectively. *In this assignment, all contexts in context-sensitive pointer analysis should be produced by this class*. You will implement several subclasses of this class as explained in Section 3.

- `pascal.taie.analysis.pta.core.cs.CSCallGraph`
  This class represents context-sensitive call graphs. It is very similar to class `DefaultCallGraph` that you have used in previous assignments, except that the call sites and methods are represented by `CSCallSite` and `CSMethod`. You will use its APIs `addReachableMethod(CSMethod)` and `addEdge(Edge)` to modify the call graph.

- `pascal.taie.analysis.pta.pts.PointsToSet`
  This class represents points-to sets, i.e., sets of `CSObj` in context-sensitive pointer analysis. It is iterable, i.e., you could iterate the objects in a points-to set via a *for* loop:

```
PointsToSet pts = ...;
for (CSObj obj : pts) {
    ...
}
```

- `pascal.taie.analysis.pta.pts.PointsToSetFactory`
  This class provides static factory methods for `PointsToSet`. You can use its two versions of `make()` methods to create instances of `PointsToSet`.

- `pascal.taie.analysis.pta.cs.PointerFlowGraph`
  This class represents a pointer flow graph of the program.

- `pascal.taie.analysis.pta.cs.WorkList`
  This class represents the worklist in the pointer analysis algorithm.

- `pascal.taie.analysis.pta.cs.Solver`
  This class implements a context-sensitive pointer analysis solver. It is incomplete, and you need to finish it as explained in Section 2.4.

2.4 Your Task [Important!]

Your first task is to finish the core of context-sensitive pointer analysis framework, i.e., class `Solver`. We have setup heap model and context selector in `Solver`'s constructor, and initialized context sensitivity (C.S.) manager, work list, pointer flow graph, and call graph in `Solver.initialize()`, and stored them in `Solver`'s fields, so that you can directly use them. You need to finish the same five APIs as the previous assignment:

- `void addReachable(CSMethod)`

  This method implements the AddReachable function given in page 88 of the slides for Lecture 12.

  Hints: 1) Same as the previous assignment, do not forget to handle static field stores/loads and static method calls in this method, and we also provide method `Solver.resolveCallee(CSObj,Invoke)` to resolve callees of various kinds of invocations in Java.

  2) In this assignment, you can also handle different kinds of statements in `addReachable()` via *visitor pattern*[1]. In context-sensitive pointer analysis, the `visit(…)` methods of concrete visitor, i.e., inner class `StmtProcessor`, need to access the `CSMethod` and `Context` being processed, and thus we add a constructor to `StmtProcessor` which accepts a `CSMethod` as parameter. In `addReachable()`, you need to create an instance of `StmtProcessor` for each reachable `CSMethod` and use it to process its statements.

  If you are not familiar with visitor pattern, it is totally fine to implement `addReachable()` in your way.

- `void addPFGEdge(Pointer,Pointer)`

  This method implements the AddEdge function given in page 94 of the slides for Lecture 12.

- `void analyze()`

  This method implements the main part, i.e., the *while* loop, of the Solve function given in page 88 of the slides for Lecture 12.

[1] https://refactoring.guru/design-patterns/visitor

Hint: You should handle array stores/loads in this method.

- `PointsToSet propagate(Pointer,PointsToSet)`

  Same as the previous assignment, this method merges two steps of the algorithm. It first computes the difference set ($\Delta = pts - pt(n)$), then propagates *pts* into *pt*(*p*) as described by the Propagate function in page 94 of the slides for Lecture 12. It returns Δ as the result of the call.

- `void processCall(CSVar,CSObj)`

  This method implements the ProcessCall function given in page 88 of the slides for Lecture 12. The hints for implementation of this method are the same as in the previous assignment.

We have provided code skeletons for the above APIs, and your task is to fill in the part with comment "`TODO - finish me`".

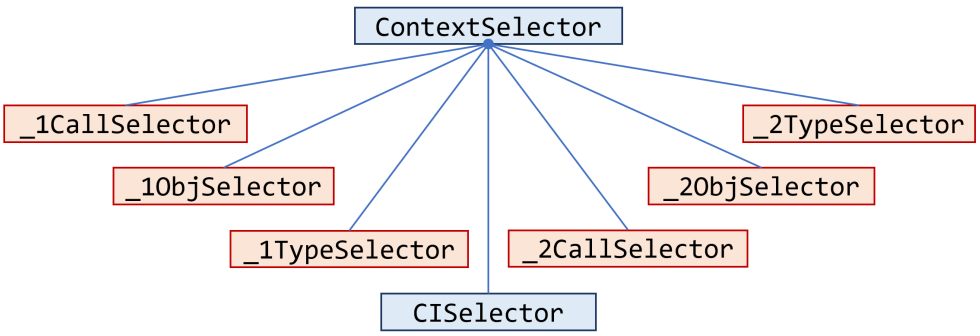
3 Implementing Common Context Sensitivity Variants

3.1 Scope

In this assignment, you will implement three common context sensitivity variants that you have learnt in Lecture 12, i.e., call-site, object and type sensitivity. For each variant, you need to finish two selectors with context limits being 1 and 2, respectively, (i.e., *k*-limiting context abstraction with *k*=1 and *k*=2).

3.2 Tai-e Classes You Need to Know

- `pascal.taie.analysis.pta.core.cs.selector.ContextSelector`
  This class represents context sensitivity variants as we have introduced in Section 2.3. This assignment involves seven subclasses of `ContextSelector`:



  `CISelector`, which implements context insensitivity, is complete and the other six selectors (the ones in red boxes) are not. You need to finish the six context selectors in this assignment, as explained in Section 3.3.

- `pascal.taie.analysis.pta.core.cs.context.ListContext`
  This class implements interface `Context`. It represents each context as a list of context elements. It provides several static factory methods, i.e., `make(…)` methods, to create contexts, and you will use them when implementing the context selectors.

3.3 Your Task [Important!]

Your second task is to implement three common context sensitivity variants by finishing three APIs, i.e., two `selectContext(…)` and one `selectHeapContext(…)`, of the six context selectors[2]. For each *k*-limiting context selector, the limit of the heap contexts is *k*-1, e.g., for 1-call-site sensitivity, the limit of heap context is 0 (essentially no heap context), and for 2-call-site sensitivity, the limit of heap context is 1.

To select method contexts, you need to implement

- `Context selectContext(CSCallSite,JMethod)`
- `Context selectContext(CSCallSite,CSObj,JMethod)`

of each selector, and to select heap contexts, you need to implement

- `Context selectHeapContext(CSMethod,Obj)`

Below we list the six context selectors you need to finish.

- `pascal.taie.analysis.pta.core.cs.selector._1CallSelector`
  This class implements 1-call-site sensitivity

- `pascal.taie.analysis.pta.core.cs.selector._1ObjSelector`
  This class implements 1-object sensitivity

- `pascal.taie.analysis.pta.core.cs.selector._1TypeSelector`
  This class implements 1-type sensitivity

- `pascal.taie.analysis.pta.core.cs.selector._2CallSelector`
  This class implements 2-call-site sensitivity

- `pascal.taie.analysis.pta.core.cs.selector._2ObjSelector`
  This class implements 2-object sensitivity

- `pascal.taie.analysis.pta.core.cs.selector._2TypeSelector`
  This class implements 2-type sensitivity

❖ Hints: 1) For how to implement call-site, object and type sensitivity, please refer to pages 103, 144, and 171 of the slides of Lecture 12.

2) In call-site sensitivity, the context selection for static methods is the same as

[2] Note that we design in this way for simplicity of the assignment. Actually, Tai-e provides configurable context selectors in terms of context limit *k*, for different context-sensitivity variants.

instance methods, i.e., at a static call, we add the call site to the caller context to compose the callee context. In object and type sensitivity, the convention of handling static methods is to *directly use the caller context* as the context of the callee (namely, the target method of the static call). You should select contexts for static methods as described above.

3) The last parameter of `selectContext(…)` and `selectHeapContext(…)` is not used in this assignment.

We have provided code skeletons for the above classes, and your task is to fill in the part with comment "`TODO - finish me`".

4 Run and Test Your Implementation

You can run the analysis as described in *Tai-e Manual for Assignments*. Tai-e performs context-sensitive pointer analysis for the program, and outputs the points-to sets of all kinds of pointers and the resulting call graph:

```
-------------- Pointer analysis statistics: --------------
#var pointers:               0 (insens) / 0 (sens)
#var points-to:              0 (insens) / 0 (sens)
#static field points-to:     0 (sens)
#instance field points-to:   0 (sens)
#array points-to:            0 (sens)
#reachable methods:          0 (insens) / 0 (sens)
#call graph edges:           0 (insens) / 0 (sens)
----------------------------------------
Points-to sets of all variables

Points-to sets of all static fields

Points-to sets of all instance fields

Points-to sets of all array indexes

#reachable methods: 0
---------- Reachable methods: ----------

#call graph edges: 0
---------- Call graph edges: ----------
----------------------------------------
```

The points-to sets and call graph are empty as you have not finished the analysis yet. After you implement the analysis, the output should be:

```
-------------- Pointer analysis statistics: --------------
#var pointers:               19 (insens) / 19 (sens)
#var points-to:              31 (insens) / 31 (sens)
#static field points-to:     0 (sens)
#instance field points-to:   4 (sens)
#array points-to:            0 (sens)
#reachable methods:          8 (insens) / 8 (sens)
#call graph edges:           11 (insens) / 11 (sens)
----------------------------------------
```

```
Points-to sets of all variables
[]:<A: B get()>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[]:<A: B get()>/temp$0 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<A: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}, []:NewObj{<OneObject: void m()>[3@L8] new A}]
[]:<A: void doSet(B)>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}, []:NewObj{<OneObject: void m()>[3@L8] new A}]
[]:<A: void doSet(B)>/p -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<A: void set(B)>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}, []:NewObj{<OneObject: void m()>[3@L8] new A}]
[]:<A: void set(B)>/b -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<B: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<OneObject: void m()>/a1 -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[]:<OneObject: void m()>/a2 -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[]:<OneObject: void m()>/b1 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[]:<OneObject: void m()>/b2 -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<OneObject: void m()>/temp$0 -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[]:<OneObject: void m()>/temp$1 -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[]:<OneObject: void m()>/temp$2 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[]:<OneObject: void m()>/temp$3 -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<OneObject: void m()>/temp$4 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<OneObject: void m()>/x -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<java.lang.Object: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}, []:NewObj{<OneObject: void m()>[3@L8] new A},
 []:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]

Points-to sets of all static fields

Points-to sets of all instance fields
[]:NewObj{<OneObject: void m()>[0@L7] new A}.f -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:NewObj{<OneObject: void m()>[3@L8] new A}.f -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}, []:NewObj{<OneObject: void m()>[9@L10] new B}]

Points-to sets of all array indexes


 #reachable methods: 8
 ---------- Reachable methods: ----------
 <A: B get()>
 <A: void <init>()>
 <A: void doSet(B)>
 <A: void set(B)>
 <B: void <init>()>
 <OneObject: void m()>
 <OneObject: void main(java.lang.String[])>
 <java.lang.Object: void <init>()>

 #call graph edges: 11
 ---------- Call graph edges: ----------
 <A: void <init>()>[0@L17] invokespecial %this.<java.lang.Object: void <init>()>(); -> [<java.lang.Object: void <init>()>]
 <A: void set(B)>[0@L21] invokevirtual %this.<A: void doSet(B)>(b); -> [<A: void doSet(B)>]
 <B: void <init>()>[0@L33] invokespecial %this.<java.lang.Object: void <init>()>(); -> [<java.lang.Object: void <init>()>]
 <OneObject: void m()>[1@L7] invokespecial temp$0.<A: void <init>()>(); -> [<A: void <init>()>]
 <OneObject: void m()>[4@L8] invokespecial temp$1.<A: void <init>()>(); -> [<A: void <init>()>]
 <OneObject: void m()>[7@L9] invokespecial temp$2.<B: void <init>()>(); -> [<B: void <init>()>]
 <OneObject: void m()>[10@L10] invokespecial temp$3.<B: void <init>()>(); -> [<B: void <init>()>]
 <OneObject: void m()>[12@L11] invokevirtual a1.<A: void set(B)>(b1); -> [<A: void set(B)>]
 <OneObject: void m()>[13@L12] invokevirtual a2.<A: void set(B)>(b2); -> [<A: void set(B)>]
 <OneObject: void m()>[14@L13] temp$4 = invokevirtual a1.<A: B get()>(); -> [<A: B get()>]
 <OneObject: void main(java.lang.String[])>[0@L3] invokestatic <OneObject: void m()>(); -> [<OneObject: void m()>]
 ----------------------------------------
```

Note that above is the result of context-insensitivity. You can configure Tai-e to analyze the input program using other variants you implement. You just need to edit line 3 of `plan.yml` in `tai-e/` directory, i.e., `cs: ci`, and change `ci` to other variants below:

- `ci`: context insensitivity (complete and ready to use)
- `1-call`: 1-call-site sensitivity
- `1-obj`: 1-object sensitivity
- `1-type`: 1-type sensitivity
- `2-call`: 2-call sensitivity
- `2-obj`: 2-object sensitivity
- `2-type`: 2-type sensitivity

For example, if you set `cs: 1-obj`, then the output should be

```
-------------- Pointer analysis statistics: --------------
#var pointers:                19 (insens) / 28 (sens)
#var points-to:               28 (insens) / 28 (sens)
#static field points-to:      0 (sens)
#instance field points-to:    2 (sens)
#array points-to:             0 (sens)
#reachable methods:           8 (insens) / 15 (sens)
#call graph edges:            11 (insens) / 14 (sens)
----------------------------------------
Points-to sets of all variables
[NewObj{<OneObject: void m()>[0@L7] new A}]:<A: B get()>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[NewObj{<OneObject: void m()>[0@L7] new A}]:<A: B get()>/temp$0 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[NewObj{<OneObject: void m()>[0@L7] new A}]:<A: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[NewObj{<OneObject: void m()>[0@L7] new A}]:<A: void doSet(B)>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[NewObj{<OneObject: void m()>[0@L7] new A}]:<A: void doSet(B)>/p -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[NewObj{<OneObject: void m()>[0@L7] new A}]:<A: void set(B)>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[NewObj{<OneObject: void m()>[0@L7] new A}]:<A: void set(B)>/b -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[NewObj{<OneObject: void m()>[0@L7] new A}]:<java.lang.Object: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[NewObj{<OneObject: void m()>[3@L8] new A}]:<A: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[NewObj{<OneObject: void m()>[3@L8] new A}]:<A: void doSet(B)>/%this -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[NewObj{<OneObject: void m()>[3@L8] new A}]:<A: void doSet(B)>/p -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[NewObj{<OneObject: void m()>[3@L8] new A}]:<A: void set(B)>/%this -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[NewObj{<OneObject: void m()>[3@L8] new A}]:<A: void set(B)>/b -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[NewObj{<OneObject: void m()>[3@L8] new A}]:<java.lang.Object: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[NewObj{<OneObject: void m()>[6@L9] new B}]:<B: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[NewObj{<OneObject: void m()>[6@L9] new B}]:<java.lang.Object: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[NewObj{<OneObject: void m()>[9@L10] new B}]:<B: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[NewObj{<OneObject: void m()>[9@L10] new B}]:<java.lang.Object: void <init>()>/%this -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<OneObject: void m()>/a1 -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[]:<OneObject: void m()>/a2 -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[]:<OneObject: void m()>/b1 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[]:<OneObject: void m()>/b2 -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<OneObject: void m()>/temp$0 -> [[]:NewObj{<OneObject: void m()>[0@L7] new A}]
[]:<OneObject: void m()>/temp$1 -> [[]:NewObj{<OneObject: void m()>[3@L8] new A}]
[]:<OneObject: void m()>/temp$2 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[]:<OneObject: void m()>/temp$3 -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]
[]:<OneObject: void m()>/temp$4 -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[]:<OneObject: void m()>/x -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]

Points-to sets of all static fields

Points-to sets of all instance fields
[]:NewObj{<OneObject: void m()>[0@L7] new A}.f -> [[]:NewObj{<OneObject: void m()>[6@L9] new B}]
[]:NewObj{<OneObject: void m()>[3@L8] new A}.f -> [[]:NewObj{<OneObject: void m()>[9@L10] new B}]

Points-to sets of all array indexes
```

The [...] before each variable and object represent its context.

In addition, Tai-e outputs the IRs for the classes of the program it analyzes to folder `output/`. The IRs are stored as .tir files which can be opened by general text editors.

We provide test driver `pascal.taie.analysis.pta.CSPTATest` for this assignment, and you could use it to test your implementation.

We encourage you to use different context sensitivity variants to analyze the test cases in this assignment (e.g., `OneObject.java`), and observe precision differences of the resulting points-to sets and call graphs computed by different context-sensitive pointer analyses.

5 General Requirements

- In this assignment, your only goal is correctness. Efficiency is not your concern.
- **DO NOT** distribute the assignment package to any others.
- Last but not least, do **NOT** plagiarize. The work must be all your own!

6 Submission of Assignment

Your submission should be a zip file, which contains your implementation of

- `Solver.java`
- `_1CallSelector.java`
- `_1ObjSelector.java`
- `_1TypeSelector.java`
- `_2CallSelector.java`
- `_2ObjSelector.java`
- `_2TypeSelector.java`

The naming convention your submission is: `<STUDENT_ID>-<NAME>-A6.zip`
Please submit your assignment to 教学立方.

7 Grading

The points will be allocated for correctness. We will use your submission to analyze the given test files from the `src/test/resources/` directory, as well as other tests of our own, and compare your output to that of our solution.

Good luck!