Programming Assignment 7:

Alias-Aware Interprocedural Constant Propagation

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

1 Assignment Objectives

- Implement an alias-aware interprocedural constant propagation for Java.

In this programming assignment, you will continue to improve the precision of constant propagation on top of Assignment 4. Specifically, you will leverage the results of pointer analysis that you implemented in the previous assignment to derive alias information, and use it to handle fields and arrays more precisely in an interprocedural constant propagation.

As in Assignment 4, i.e., only `int` values are considered in constant propagation, but in this assignment, you need to additionally take aliasing into account, namely, handling the stores/loads of fields and arrays more precisely by incorporating alias information; besides, you could ignore some cases as explained in Section 2. We will introduce the necessary knowledge to finish this assignment in the following sections. If your implementation is correct, you can observe that alias-aware interprocedural constant propagation achieves better precision than the one you implemented in Assignment 4 which treats fields and arrays conservatively.

This assignment is more open than previous ones. We only describe “what to do” in this document and leave implementation details to you. You need to figure out “how to do” by yourself.

2 Introduction to Alias-Aware Constant Propagation

In this section, we first introduce aliasing, and then discuss how to handle fields and arrays in the presence of aliasing.

2.1 Aliasing

Aliasing describes a situation in which a data location in memory can be accessed through different symbolic names in the program[1], and different symbolic names that refer to the same memory location are called *aliases*. In Java, the accesses to instance fields and arrays could form aliases. For example, if variables `x` and `y` point to the same

[1] https://en.wikipedia.org/wiki/Aliasing_(computing)

object, then field accesses `x.f` and `y.f` are aliases as they refer to the same field; if variables `a` and `b` point to the same array and `i` and `j` have the same value, then array accesses `a[i]` and `b[j]` are also aliases as they refer to the same location (index) of the same array.

In the presence of aliasing, modifying the value of an instance field/array through one instance field/array access implicitly modifies the values associated with all its aliases. For example, if `x.f`, `y.f` and `z.f` are aliased, then store statement `x.f = 5;` not only modifies the value of `x.f` to 5, but also sets the values of `y.f` and `z.f` to 5. Thus, to precisely analyze fields and arrays in constant propagation, we need to reason about the alias information in the analyzed program.

Note that static fields in Java cannot be aliased, i.e., for a static field `T.f`, it has only one symbolic name (`T.f`) and there is only one way to access it (through `T.f`). Thus, the handing of static fields is simpler than instance fields and arrays as we do not need to concern about aliases.

2.2 Analysis of Instance Fields

Handle Instance Fields More Precisely. In Assignments 2 and 4, we treat instance field loads conservatively by unconditionally setting their LHS variables to NAC:

```
x = a.f; // always set val(x) to NAC in previous assignments
```

In this assignment, we take a step forward towards better precision. When analyzing an instance field load, say *L*, we look up the store statements which modify the aliases of the instance field and meet the stored values to the LHS variable of *L* as shown below:

```
    y = 5;
    p.f = y; // p.f is an alias of a.f
    …
L: x = a.f; // meet val(y) to val(x)
```

In this way, when all values stored to the field (through all possible store statements) are the same constants, say 5, we can obtain a more precise result (`x`=5) than the previous conservative handling (`x`=NAC). In other cases, i.e., the instance field is stored multiple times with different values (e.g., `q.f = 6;` where `q.f` is also an alias of `a.f`), or the instance field is stored a NAC (e.g., `p.f = y;` where `y`=NAC), the value of the LHS variable of the load statement should also be NAC (i.e., `x`=NAC) for soundness.

Compute Alias Information. In this assignment, we leverage the pointer analysis (that you implemented in the previous assignment) to compute alias information. Specifically, for any two instance field accesses, say `x.f` and `y.f`, if there exists overlap between the points-to sets of the base variables `x` and `y`, then we consider `x.f` and `y.f` as alias of each other.

Precision of Instance Field Handling. You may notice that the handling of instance fields described above ignores the order of load and store statements, i.e., it is essentially flow-insensitive. Such handling may lose precision like other flow-insensitive analyses. For example, in the code snippet below:

```
1 p = q = a;
2 p.f = 555;
3 x = a.f; // x -> NAC in the current handling
4 q.f = 666;
```

The analysis finds out that both `p.f` and `q.f` are aliases of `a.f`, then it meets all values stored in aliases of `a.f`, i.e., 555 and 666, to LHS variable of load statement of `a.f`, i.e., `x`, and thus the analysis concludes that `x` is NAC after line 3, which is imprecise.

There are some other analysis approaches that could achieve better precision than the above handling, e.g., tracking the values of *access paths* (where an access path is a variable and a sequence of field names, e.g., `p.f` and `q.f.g`) together with variables in a flow-sensitive manner. However, this requires complicated modifications to the analysis. Hence, we choose a straightforward strategy in this assignment for simplicity.

2.3 Analysis of Static Fields

The handling of static fields is simpler than that of instance fields as it does not need to consider aliases. When analyzing a load statement of a static field, say `x = T.f;`, you just need to look up the store statements of the *same* field (`T.f`) in the program, and meet the stored values to the LHS variable (`x`) of the load statement. This can be done without the pointer and alias information.

Note that the values of static fields may come from field initializer[2] or static initializer[3] as shown at lines 2 and 7 in the following code snippet:

```
1 class A {
2     static int one = 1; // field initializer
3     static int two;
4
5     static { // static initializer
6         one = 1; // generated for field initializer at line 2
7         two = 2;
8     }
9 }
```

Specifically, a field initializer (e.g., line 2) would be compiled to a store statement (e.g., line 6) in the static initializer (e.g., static block in lines 5-8), and thus only dealing with static initializers is enough. However, the call graph builders in the assignments do not

[2] https://docs.oracle.com/javase/specs/jls/se11/html/jls-8.html#jls-8.3.2

[3] https://docs.oracle.com/javase/specs/jls/se11/html/jls-8.html#jls-8.7

process static initializers (but they are handled in Tai-e), thus such store statements are unreachable in the ICFG. For simplicity, we ignore field initializers and static initializers in this assignment.

2.4 Analysis of Arrays

The handling of arrays is similar to that of instance fields. When analyzing a load statement of an array, say `x = a[i];`, you need to look up the statements that store values to the aliases of `a[i]`, and meet the stored values to `x`. However, handling arrays is more complicated, since when checking whether two array accesses, say `a[i]` and `b[j]`, are aliased, you need to consider not only the points-to sets of base variables `a` and `b` but also the values of index variables `i` and `j`. Interestingly, you can use constant propagation to resolve the index values *on the fly* as they are also of type `int`.

Suppose that the points-to sets of `a` and `b` are overlapped, then we use the results of constant propagation for `i` and `j` to determine whether `a[i]` and `b[j]` are aliased or not, as follows (this design considers the monotonicity and soundness of the analysis):

`a[i]` and `b[j]`	`j`=UNDEF	`j`=c_2	`j`=NAC
`i`=UNDEF	Not aliased	Not aliased	Not aliased
`i`=c_1	Not aliased	Aliased iff c_1=c_2	Aliased
`i`=NAC	Not aliased	Aliased	Aliased

2.5 Assumption About Field/Array Initialization

Since our handling of fields and arrays is essentially flow-insensitive, when analyzing load of a field/array, we need to meet *all* values that are stored to the field/array for soundness. Unlike local variables which cannot be used before initialized, in Java, fields (including both instance and static fields) and arrays can be loaded even though they are not explicitly initialized, because Java always implicitly initializes them with a default value, e.g., 0 for type `int`[4]. For example, in this code snippet:

```
1 class A {
2     int f; // f is implicitly initialized to 0 by default
3 }
4 …
5 A a = new A();
6 // a.f = 5;
7 int x = a.f;
```

Although field `f` of object `new A` has not been explicitly initialized, it can still be loaded at line 7, and the loaded value is 0 (as Java implicitly initializes it to 0).

In the presence of such implicit initialization, even though an instance field is stored

[4] https://docs.oracle.com/javase/specs/jls/se11/html/jls-4.html#jls-4.12.5

with a non-zero constant value in the program (e.g., uncomment line 6 in the above code snippet), we still have to treat it as NAC, because it holds two values, i.e., 0 (default value) and the stored value (e.g., 5). Accordingly, the analysis result of line 7 is `x`=NAC. As a result, our handling of fields and arrays is basically useless in terms of precision, as it treats virtually all loaded values as NAC.

For this situation, we make a reasonable assumption for the programs being analyzed. We assume that ***each field and each array in the program must be explicitly initialized (i.e., stored) before any load***. Under this assumption, every loaded value must come from the store statements in the program, so that we can ignore the default value (0) of fields and arrays from implicit initialization.

3 Implementing Alias-Aware Constant Propagation

3.1 Tai-e Classes You Need to Know

Most of the needed classes in this assignment have been introduced in Assignments 4 and 6. There are only two new classes you need to know in this assignment.

- `pascal.taie.analysis.pta.PointerAnalysisResult`
  This class provides APIs to query various results of pointer analysis. You will use it to compute alias information. Note that this class provides APIs to query points-to results with contexts (e.g., `getPointsToSet(CSVar)`) and without contexts (e.g., `getPointsToSet(Var)`), respectively. Since interprocedural constant propagation is unaware of context information of pointer analysis, you should use the points-to results without contexts.

- `pascal.taie.ir.exp.ArrayAccess`
  This class represents array access expressions, e.g., `a[i]`. Its instances are stored in `StoreArray` and `LoadArray` statements.

3.2 Your Task [Important!]

In this assignment, you need to finish the APIs of the two classes that you have modified in Assignment 4 (this assignment is based on Assignment 4), i.e., six APIs of `pascal.taie.analysis.dataflow.inter.InterConstantPropagation`:

- `boolean transferCallNode(Stmt,CPFact,CPFact)`
- `boolean transferNonCallNode(Stmt,CPFact,CPFact)`
- `CPFact transferNormalEdge(NormalEdge,CPFact)`
- `CPFact transferCallToReturnEdge(CallToReturnEdge,CPFact)`
- `CPFact transferCallEdge(LocalEdge,CPFact)`
- `CPFact transferReturnEdge(LocalEdge,CPFact)`

and two APIs of `pascal.taie.analysis.dataflow.inter.InterSolver`:

- `void initialize()`
- `void doSolve()`

This time, you need to handle instance fields, static fields and arrays more precisely as described in Section 2. Again, this assignment is more open, and thus you should resolve all implementation details by yourself.

Similar to Assignment 4, you need to finish class `ConstantPropagation` to make `InterConstantPropagation` work. You could copy your implementation from Assignment 2. Besides, you need to finish classes `Solver` and `_2ObjSelector` (the default context selector in this assignment) for context-sensitive pointer analysis which is used to construct call graph and to compute alias information. You could copy your implementation from Assignment 6.

Hints: 1) `InterConstantPropagation` has an `initialize()` method which will be invoked before the solver starts. It contains the code to obtain the `PointerAnalysisResult`. If your implementation requires initialization work, you can do it in this method.

2) You can add APIs and fields to `InterConstantPropagation` and `InterSolver` if necessary.

3) `InterConstantPropagation` holds the instance of `InterSolver` in its field `solver`, thus you can call `solver`'s APIs from `InterConstantPropagation`.

4 Run and Test Your Implementation

You can run the analyses as described in *Tai-e Manual for Assignments*. In this assignment, Tai-e first performs a context-sensitive pointer analysis which builds a call graph for the program, then constructs ICFG based on the call graph, and finally runs interprocedural constant propagation on the ICFG. To help debugging, it outputs the call graph and the results of interprocedural constant propagation:

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

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

```
-------------------- <InstanceField: void main(java.lang.String[])> (inter-constprop) --------------------
[0@L4] temp$0 = new A; null
[1@L4] invokespecial temp$0.<A: void <init>()>(); null
[2@L4] a1 = temp$0; null
[3@L5] temp$1 = 111; null
[4@L5] a1.<A: int f> = temp$1; null
[5@L6] x = a1.<A: int f>; null
[6@L7] temp$2 = new A; null
[7@L7] invokespecial temp$2.<A: void <init>()>(); null
[8@L7] a2 = temp$2; null
[9@L8] temp$3 = 222; null
[10@L8] a2.<A: int f> = temp$3; null
[11@L9] y = a2.<A: int f>; null
[12@L9] return; null
```

The call graph is empty (0 reachable method) and OUT facts are `null` as you have not finished the analyses yet. After you implement the analyses, the output should be:

```
#reachable methods: 3
---------- Reachable methods: ----------
<A: void <init>()>
<InstanceField: void main(java.lang.String[])>
<java.lang.Object: void <init>()>

#call graph edges: 3
---------- Call graph edges: ----------
<A: void <init>()>[0@L13] invokespecial %this.<java.lang.Object: void <init>()>(); -> [<java.lang.Object: void <init>()>]
<InstanceField: void main(java.lang.String[])>[1@L4] invokespecial temp$0.<A: void <init>()>(); -> [<A: void <init>()>]
<InstanceField: void main(java.lang.String[])>[7@L7] invokespecial temp$2.<A: void <init>()>(); -> [<A: void <init>()>]
----------------------------------------

-------------------- <InstanceField: void main(java.lang.String[])> (inter-constprop) --------------------
[0@L4] temp$0 = new A; {}
[1@L4] invokespecial temp$0.<A: void <init>()>(); {}
[2@L4] a1 = temp$0; {}
[3@L5] temp$1 = 111; {temp$1=111}
[4@L5] a1.<A: int f> = temp$1; {temp$1=111}
[5@L6] x = a1.<A: int f>; {temp$1=111, x=111}
[6@L7] temp$2 = new A; {temp$1=111, x=111}
[7@L7] invokespecial temp$2.<A: void <init>()>(); {temp$1=111, x=111}
[8@L7] a2 = temp$2; {temp$1=111, x=111}
[9@L8] temp$3 = 222; {temp$1=111, temp$3=222, x=111}
[10@L8] a2.<A: int f> = temp$3; {temp$1=111, temp$3=222, x=111}
[11@L9] y = a2.<A: int f>; {temp$1=111, temp$3=222, x=111, y=222}
[12@L9] return; {temp$1=111, temp$3=222, x=111, y=222}
```

Same as Assignment 4, Tai-e outputs the ICFG of the program it analyzes to folder `output/`, which is stored as .dot files which can be visualized by Graphviz.

We provide class `pascal.taie.analysis.dataflow.analysis.constprop.InterCPAliasTest` as the test drivers for alias-aware interprocedural constant propagation, and you could use them to test your implementation as described in *Tai-e Manual for Assignments*.

We encourage you to use your implementation of Assignment 4 to analyze the test cases in this assignment, and observe the precision differences between alias-unaware and

alias-aware constant propagations. In addition, this assignment uses 2-object-sensitive pointer analysis by default to build call graph and compute alias information. We encourage you to analyze some test cases (e.g., `ObjSens.java`) with other context sensitivity variants (as explained in Assignment 6), e.g., context insensitivity, and observe how the precision of pointer analysis affects the precision of other analyses that depend on it. To do so, you may copy other context selectors from Assignment 6.

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

- `InterConstantPropagation.java`
- `InterSolver.java`
- `ConstantPropagation.java` (optional[5])

The naming convention your submission is: `<STUDENT_ID>-<NAME>-A7.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!

[5] If your submission contains this file, we will grade your submission with your provided version of `ConstantPropagation.java`; otherwise, we will use our version to grade your submission.