Software Engineering
Programming Languages

Building a Data Flow Graph for Java

Lukas Pietzschmann, Denis Neumüller, Florian Sihler

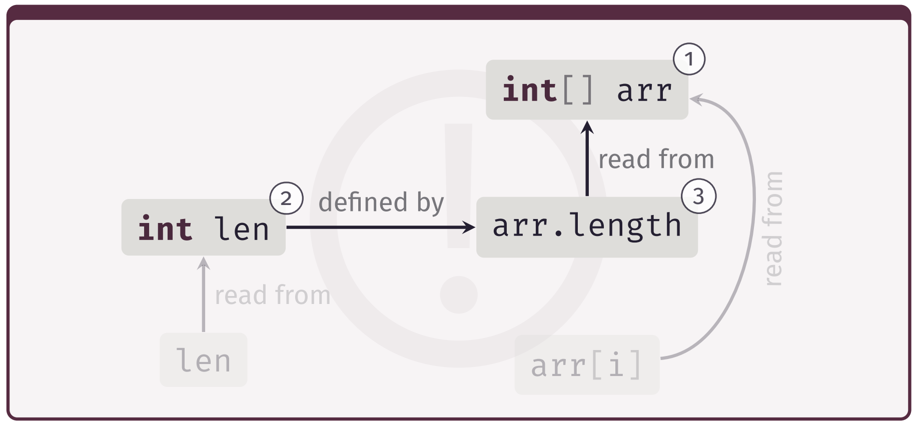
universität uulm

```
void foo(int[] arr①){
  int len② = arr③.length;
  for(int i = 0; i < len; ++i) {
    System.out.println(arr[i]);
  }
}
```

The Problem

Is `len` dependent on `arr`?

Yes, because there's a path between **int** `len` ② and **int**`[] arr` ①.



The Algorithm

We keep a set of declarations whose data flow we are interested in — the *active set*. When traversing the AST, we look at every piece of code, but we're especially interested in (1) variables and (2) assignments, as they primarily influence the data flow.

For every variable we encounter, we get its declaration and add it to the active set[1]. Additionally, we create a new node inside the graph. This node can have one of two types: (1) *Def*, if the element is a declaration or on the left-hand side of an assignment and (2) *Use*, otherwise.

Now we can hook up the newly added node. For this purpose, we get all recently added elements that *share a declaration* with the current element — *e.g.* in the figure on the right, all a's share a declaration. We then connect the newly added node to all recently added nodes, only *omitting Use -> Use* edges.

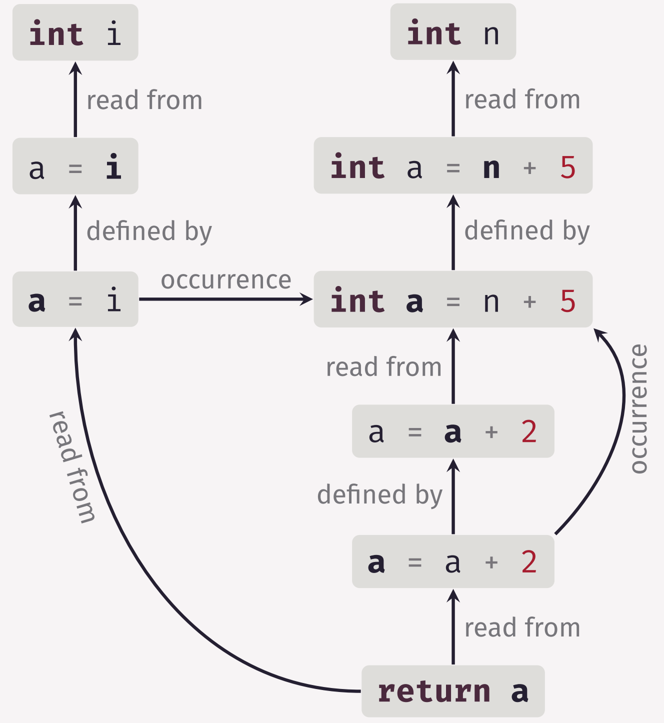
Assignments are already largely handled by the rules above. Variables on the left and right-hand side are correctly inserted, we just have to connect the assignee to active elements in the assignment. This is done by getting all active elements from the right-hand side and drawing an edge from the *left-hand side to them*.

The types of two nodes alone lets us infer the type of the edge connecting them. We distinguish between three edge types: (1) *Read From*: `Use` → `Def`, (2) *Defined By*: `Def` → `Use`, and (3) *Occurrence*: `Def` → `Def`.

Getting the data flow to, from, or between elements can be narrowed down to a *graph reachability problem*. (1) *To*: We return all elements that can be reached from the given element. (2) *From*: We return all elements that can reach the given element. (3) *Between*: We return all paths between two elements.

An Example

```
public int foo(int n, int i) {
  int a = n + 5;
  if(Math.random() > 0.5) {
    a = i;
  } else {
    a = a + 2;
  }
  return a;
}
```



Future Work

While basic cases are already handled well, a variety of situations are taken care of poorly or not at all. Here are some areas we want to improve in:

Methods When a method is called, we definitely want to trace the data flow of all arguments through the method. We currently have basic support for this, but work still needs to be put into it.

Recursion If a method calls itself — either directly or indirectly — we need to recognize this when generating and traversing the data flow graph.

Arrays We currently treat arrays as a single *thing*. Ideally, we would keep track of every index that is accessed.

Control Structures We currently only treat `If-Else` statements correctly. Support for other control structures (*e.g.* loops) is definitely needed.

Project Organization

Organization We're doing *weekly meetings* to review the current progress, discuss decisions and / or potential issues, and prioritize and schedule future work. If important decisions were made in our meeting, we record them in a *wiki* so that we can still understand later why we decided the way we did.

Technology Stack The project is developed using *Java 17*. Additionally, we employ *Gradle* (Groovy) as a build and dependency management system.

The current AlDeSCo-Prototype uses *Spoon* to parse and analyze any given Java source code. We then traverse Spoon's AST in order to build the data flow graph, which is represented by *JGraphT*'s data structures. The actual spanning of the data flow graph is completely done by ourselves without the help of a library.

QA In order to assure that no functionality breaks over time, we use a small[2] but (hopefully 😉) expanding *test suite*.

To ensure the employment of best practices, we often *review* newly added code in our weekly meetings. This massively helps in keeping the code easy to understand and well readable. However, if we overlook a small error, it will most likely be caught by TeamScale, our *static code analysis* platform.

We also use *assertions* whenever possible to make sure our mental model aligns with the actual execution of the code.

Documentation The above-mentioned GitLab *wikis* are not only used for meeting protocols, but also to document important aspects of the code. However, most of our documentation is written inline using the *JavaDoc* syntax.

[1] If an element is already active, the set will kindly reject it.

[2] Only the test suite targeting the data flow is currently kinda small. AlDeSCo's main test suite contains ≈ 470 tests.

07/28/2023 Institute of Software Engineering and Programming Languages Ulm University