Applied Static Analysis

Advanced Call Graph Algorithms

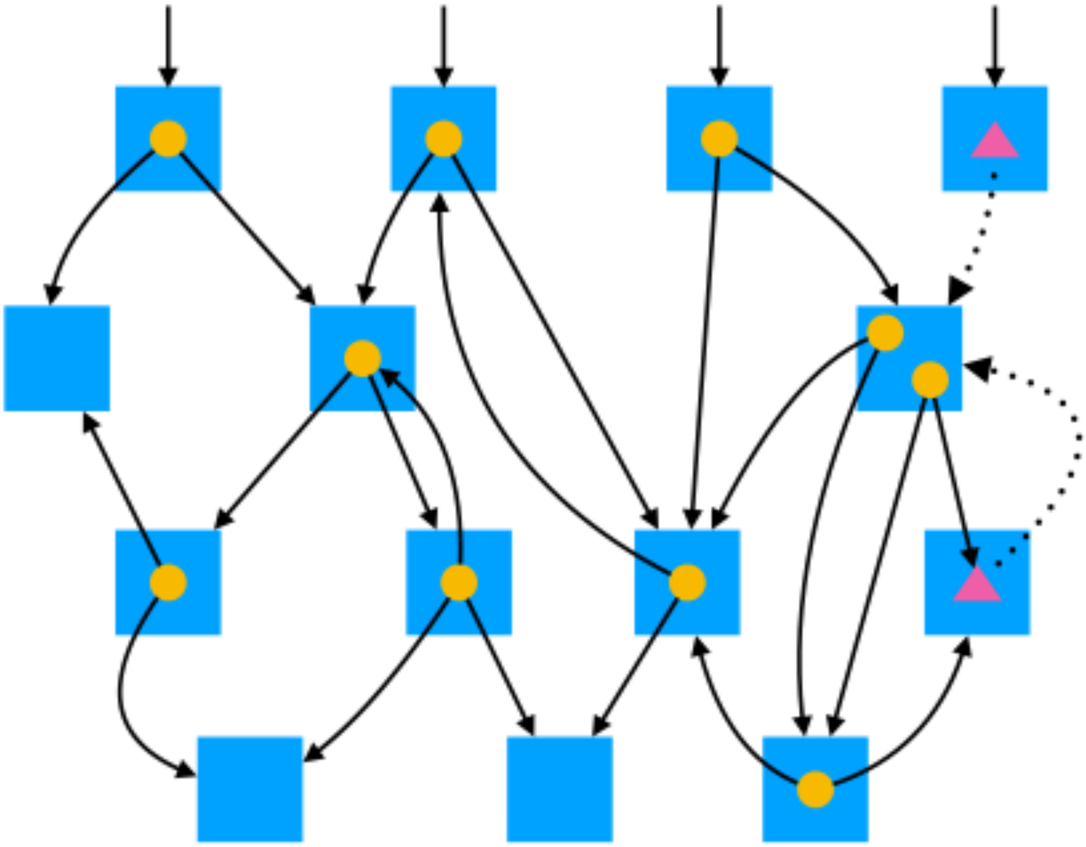
Software Technology Group
Department of Computer Science
Technische Universität Darmstadt
Dr. Michael Eichberg

> If you find any issues, please directly report them: GitHub

Some of the images on the following slides are taken from the paper: Practical Virtual Method Call Resolution for Java [1] or are based on slides created by Eric Bodden.

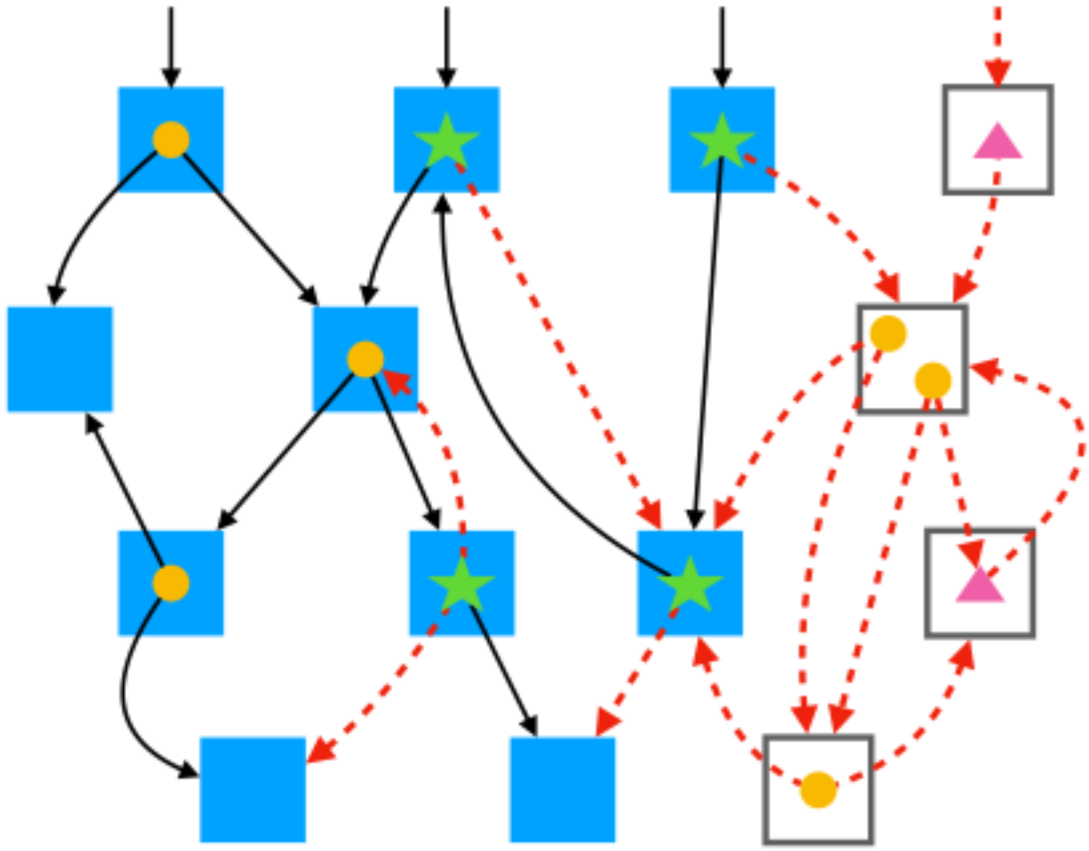
Conservative Call Graph

Let’s assume that we have computed a call graph that is a conservative approximation of the runtime call graph.



- Blue box = method
- Yellow dot = potentially polymorphic call site
- Red triangle = static call site
- edge with arrow = call edge (a call edge without a source is an entry point)

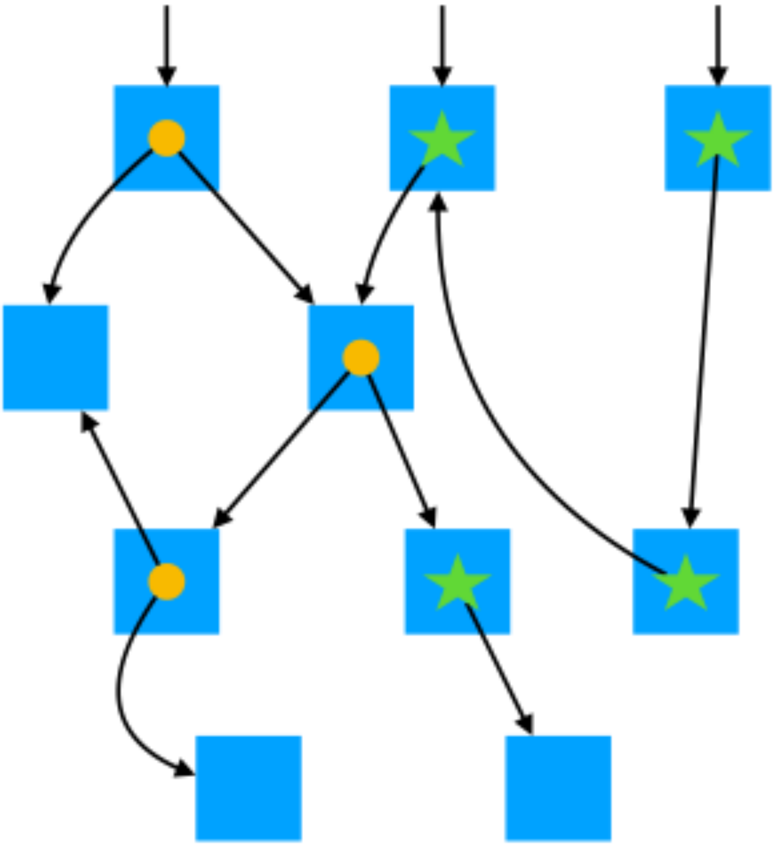
Refining a Conservative Call Graph



- Blue box = *(potentially) reachable* method | white box with gray border = unreachable method
- Yellow dot = potentially polymorphic call site | green star = polymorphic call site that is now monomorphic
- Red triangle = static call site
- edge with arrow = call edge (a call edge without a source is an entry point; a dashed call edge is an edge that can be eliminated)

Pruned Call Graph

The goal of a call graph refinement is to ...
- remove call edges
- identify unreachable methods
- determine monomorphic call edges



Declared/Variable Type Analysis (D/VTA)

Goal: Resolution of virtual methods and interface calls (in Java) using a technique that...

- scales linearly with the program size
- is more accurate than (CHA and) RTA
- simple to implement

i.e., their goal was to identify monomorphic call sites to reduce costs of virtual method calls and to identify possibilities for method inlining.

Basic idea: **take variable assignments into account**!

In a sense these are also refinements of RTA. Here, we are concerned about the types of objects that reach each variable.

Mono- vs. Polymorphic calls

Here, a call site is considered to be monomorphic if the *number of call edges* (not runtime types) is **one**.

Consider the following example:

```
class S { void m(); }
class X extends S { void m(){} }
class Y extends S { void m(){} }
...
void m(boolean b, X x, Y y) {
    S o = b ? x : y;
    // The following call site is monomorphic!
    // Independent of the concrete runtime type the method
    // <Object>.toString is called.
    s.toString();

    // The following call site is polymorphic!
    s.m();
}
```

Basic observation

Given a code representation such as SOOT’s Jimple or OPAL’S TACAI:

For a type `A` to reach a receiver `o` (`o.m()`) there must be some path through the program which starts with the creation of A (`new A()`) assigned to some variable `x` which is followed by some chain of assignments of the form `x1 = x` , … , `o = xn` .

Basics of (D/V)TA

- both are built on top of a conservative call graph (e.g., one computed by CHA, RTA or even VTA itself) which is then pruned.
- The implementation is defined on top of a three-address code like representation where we have no aliasing between variables
- both are flow-insensitive

Both representations: SOOT’s Jimple and OPAL’s TACAI representations satisfy the requirements.

The experiments have shown that using CHA as the foundation delivers nearly the same results as using RTA or even applying VTA on top of itself. Hence, the overall performance of building VTA on top of CHA is best.

Basic steps of VTA

1. Compute initial conservative call graph
2. Build so-called type-propagation graph
3. Collapse strongly-connected components
4. Propagate types in one iteration (afterwards the call graph can be pruned)

The original VTA algorithm is pessimistic; it builds on top of a conservative algorithm. An optimistic implementation would implement compute the call graph on the fly (iteratively starting from the entry points) and is potentially more precise.

The Type Propagation Graph

- The nodes represent variables in the program
- The edges b → a represent assignments of the form `a = b`

Given a conservative call graph and a class C which is reachable.

- A node is generated for every reference-typed (and accessed) field f (`C.f`)
- For every reachable method (`C.m`):
    - a node is generated for every reference-typed formal parameter pi (`C.m.pi`)
    - a node is generated for every reference-typed local variable li (`C.m.li`)
    - a node is generated for the implicit `this` parameter (in case of instance methods)
    - a node is generated for the return value of `m` (`C.m.return`)

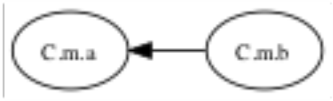
The initial reaching type information is generated by assignments of the form: `b = new A().`

Generating the type propagation graph – *normal* assignments

In the following `C.m.a` is the representative of the variable `a` defined in method `m` in class `C`. Note that it is a prerequisite of the algorithm that every variable is assigned only once/has a single definition site.

`a` and `b` are locals or parameters.

```
class C {
    void m(...) {
        a = b;
        a = b[i];
        a[i] = b;
}       }
```



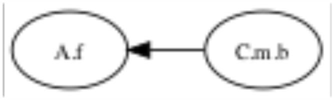
In case of a normal assignment we create an edge between the representative of the right-hand side of the assignment and the representative for the left-hand side of the assignment.

Generating the type propagation graph – field assignments

`a` could be a local or a parameter (including `this`)

```
class C {
    void m(...) {
        a.f = b;

    }
}
```



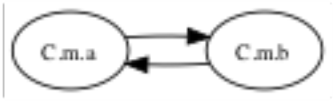
In this case the type propagation is done w.r.t. the declaring class of the field (`A`) and not the instance `a` ; hence the analysis is field-based.

Generating the type propagation graph – handling Arrays

Arrays are considered as one large aggregate.

Please recall that any array type inherits from `Object` and implements the interfaces `Cloneable` and `Serializable`.

```
class C {
    void m() {
        // if either a or b has type:
        // Object, Seriablizable, Cloneable or some array type:
        a = b;
}       }
```



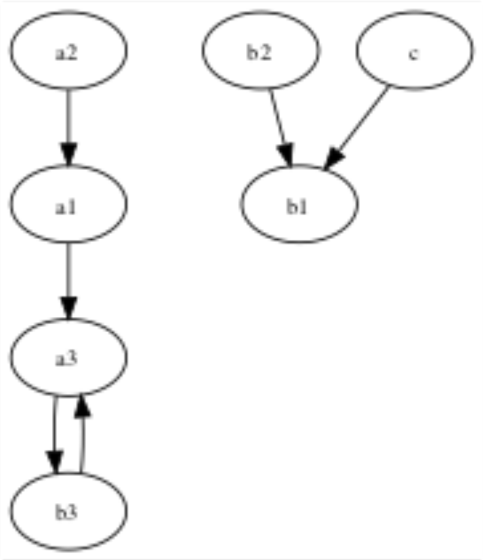
Hence, if we have an assignment that is potentially related to an array, we propagate the type in both directions. This is required to handle potential aliasing. Imagine a code snippet such as: `A[] a = new A[10]; Object o1 = a; A[] b = (A[]) o1` – in this case `a`, `o1` and `b` are all referring to the *same* array. Hence, an assignment to `b[1]` is also an assignment to `a[1]`!

VTA Example

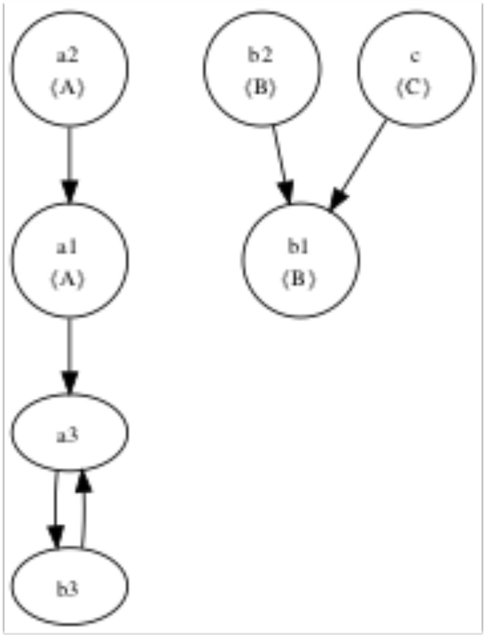
```
A a1,a2,a3;
B b1,b2,b3;
C c;
a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```

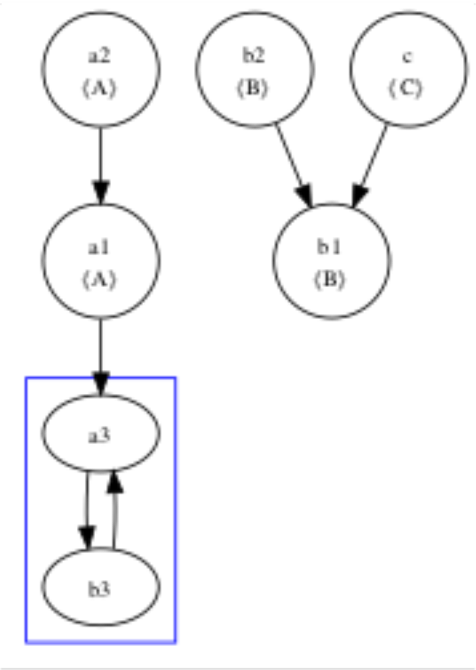
VTA Example - 1. build the graph



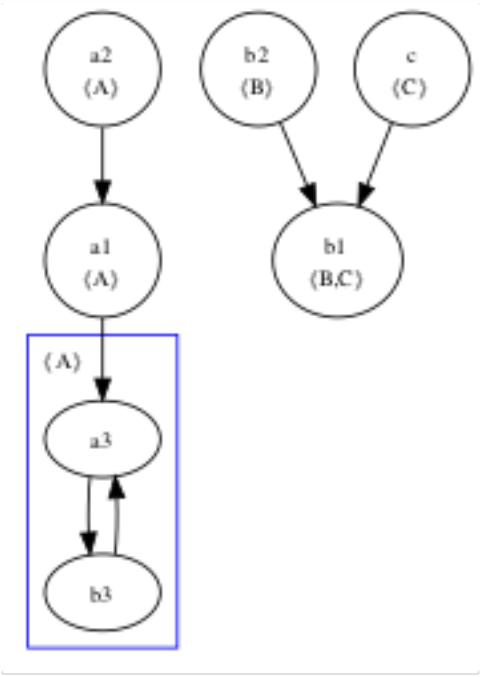
VTA Example - 2. assign initial types



VTA Example - 3. strongly connected components



VTA Example - 4. final type propagation graph



VTA Assessment

- (the originally proposed algorithm) requires an initial call graph
- more precise than RTA
- relatively fast
- imprecision remains (fields of different objects are modeled as single node.)

Declared Type Analysis (DTA)

Basically identical to VTA except that *we use the declared type of a variable as the representative*. Hence, we put all variables the same declared type into the same equivalence class.

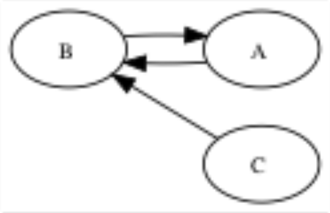
Therefore, the type propagation graph will be much smaller and the propagation will be much faster when compared to VTA; however, the precision will suffer.

DTA Example

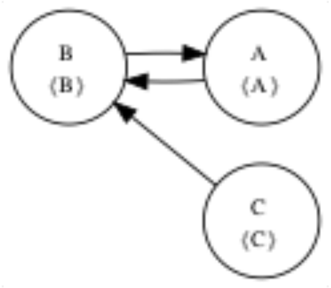
```
A a1,a2,a3;
B b1,b2,b3;
C c;
a1 = new A();
a2 = new A();
b1 = new B();
b2 = new B();
c = new C();

a1 = a2;
a3 = a1;
a3 = b3;
b3 = (B) a3;
b1 = b2;
b1 = c;
```

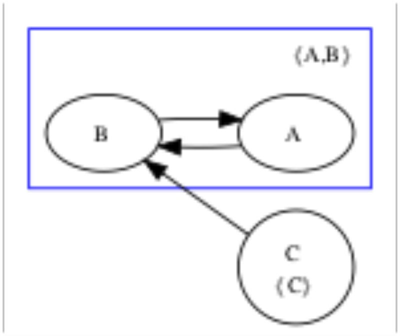
DTA Example - 1. build the graph



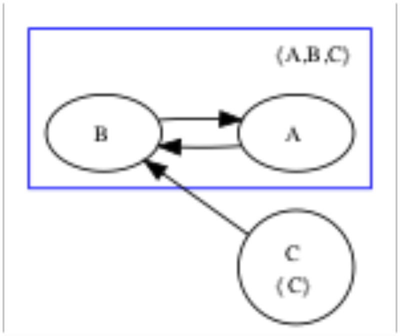
DTA Example - 2. assign initial types



DTA Example - 3. strongly connected components



DTA Example - 4. final type propagation graph



DTA Assessment

- (the originally proposed algorithm) requires an initial call graph
- more precise than RTA; less precise then VTA
- relatively fast (~10 time less expensive than VTA)
- significantly less precise than VTA

Outlook - using generic type information

Take type parameters into consideration:

```
class C[X](x : X){ def toString : String = {x.toString(); }}
class SubC(x : String) extends C[String](x)
...
/* CS */ new SubC("Hello World").toString
```

At the given call site `CS` the call of `<SubC>.toString` is effectively a call on `<X>.toString` which calls `<String>.toString`; however standard call graph algorithms don’t consider generic type information and will resolve the call to `<Object>.toString`.

Papers which discuss this topic in greater detail are: [2] and [3]

References

1. Practical Virtual Method Call Resolution for Java; Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Valleé-Rai, Patrick Lam, Etienne Gagnon and Charles Godin; OOPSLA 2000, ACM↩

2. Call Graphs for Languages with Parameteric Polymorphism; Dmitry Petrashko, Vlad Ureche, Ondřej Lhoták and Martin Odersky; OOPSLA 2016, ACM↩

3. Constructing Call Graphs of Scala Programs; Karim Ali, Marianna Rapport, Ondřej Lhoták, Julian Dolby and Frank Tip; ECOOP 2014, Springer↩