A Relational Static Semantics for Call Graph Resolution

August 1, 2019

Jinan University

Abstract. The problem of resolving virtual method and interface calls in Object-Oriented languages has been a long standing challenge to the program analysis community. In this paper, we propose a new approach called type flow analysis that represent the propagation of type information between program variables by a group of relations without the help of a heap abstraction. We prove that regarding the precision on reachability of class information to variables, our method produces results equivalent to that one can derive from a points-to analysis. Moreover, in practice, our method consumes lower time and space usage, as supported by the experimental results.

1 Optimization

Algorithm 1 split

Input: **valueToSplit:**set, **splitter**:$Value \times Value \times Type \times Field$
Output: **isSplit:**$boolean$, **include:**set, **exclude:**set

```
isSplit=false;
include=set();
exclude=set();
for each v in valueToSplit do
  relations = getRelations(v);
  for each r in relations do
    if r match splitter then
      include.add(v);
      isSplit=true;
      break;
    end if
  end for
end for
if isSplit then
  exclude=valueToSplit-include;
end if
```

2 Experiment

We evaluated our approach by measuring its performance in terms of generated callsites and runtime on 13 benchmark test cases. *Compress*, *crypto* are from the SPECjvm2008 suite, and the rest of those are from the Dacapo suite. We randomly selected these test cases based on its size increasing. All of our experiments were conducted on an Intel i5-8250U processor at 1.60 GHz and 8 GB memory running Ubuntu 16.04LTS, with the Openjdk 1.8.0.
We compared our approach against CHA and PTA implemented by Soot, the most popular static analysis framework for Java. We run context-insensitive PTA because our approach is context-insensitive, thus results will be more comparable. Our evaluation addressed the following three research questions:

$RQ1$: How efficient is our approach compared with the traditional class hierarchy analysis and points-to analysis?
$RQ2$: Is our approach more accurate than CHA or PTA?
$RQ3$: How significantly does our optimization algorithm achieve to reduce space consumption?

2.1 Implementation

The implemention of our approach is written in Java. We use Soot as our basic framework to extract SSA based representation of the benchmark code. Also, we compared against CHA and PTA implemented by Soot. Our approach optional provides visualization of relations we generated for friendly analysis.

2.2 RQ1:Efficiency

In the first research question, we executed 10 times on each benchmark test case with CHA, PTA and TFA. We calculated runtime consumption on average and each different kind of relation our approach generated. As shown in Table 1, our approach is more efficient than PTA, and generally more efficient than CHA, especially when the program callsite increasing. Time cost in our approach is basically depending on the size of generated relations. More specifically, most of the time is consumed to calculate the fixpoint. For example, CHA, PTA analyze the benchmark *bootstrap* about 23.74 and 34.13 seconds, respectively. TFA only excutes about 0.03 second. The reason is mostly because the relations only reach to 661 in total. There are 3 benchmarks, *xalan, antlr, batik*, can not be analyzed by PTA because it raise the jvm GC overhead limit error. From what we can know, this error is thrown because the jvm garbage collector is taking an excessive amount of time (by default 98% of all CPU time of the process) and recovers very little memory in each run (by default 2% of the heap). We put the label "N/A" in Table 1

Table 1. Time cost with different analysis

bench	$\mathbf{T}_{cha}(s)$	$\mathbf{T}_{pta}(s)$	$\mathbf{T}_{tfa}(s)$	$\mathbf{R}_{type}$	$\mathbf{R}_{\sqsubseteq}$	$\mathbf{R}_{\xrightarrow{f}}$
compress	0.02	0.11	0.02	87	202	24
crypto	0.01	0.13	0.03	94	226	18
bootstrap	23.97	34.59	0.03	191	453	17
commons-codec	0.008	0.12	0.13	306	3324	49
junit	24.56	34.01	0.18	1075	5772	241
commons-httpclient	0.008	0.11	0.36	2423	8511	521
serializer	22.95	32.72	1.71	3006	17726	331
xerces	22.49	31.18	3.74	12590	72503	2779
eclipse	22.80	39.74	1.68	7933	37435	1620
derby	22.71	49.05	18.09	20698	191854	5386
xalan	79.57	163.68	42.11	32971	162249	3696
antlr	43.08	92.18	3.96	16117	76741	3879
batik	48.85	94.20	6.38	29409	122534	6039

2.3 RQ2:Accurancy

We answer the second question with considering generated callsite as the aspect of accurancy. In type flow analysis, a method call will be resolve depending on the type which the caller can access through the relation we generated and that type must have the privage to access the method. Table 2 shows the callsite generated by different analysis. We calculate the origin callsite, the callsite directly written in source code, as the baseline. Our approach reduces a satisfying number of callsite, yielding corresponding accurancy. The result is comparable to what can be achieved by the class hierarchy analysis and the points-to analysis.

Table 2. Callsite generated with different analysis

bench	$\mathbf{CS}_{origin}$	$\mathbf{CS}_{cha}$	$\mathbf{CS}_{pta}$	$\mathbf{CS}_{tfa}$
compress	153	160	18	73
crypto	302	307	62	121
bootstrap	657	801	891	328
commons-codec	1162	1372	270	554
junit	3196	17532	11176	1197
commons-httpclient	6817	17118	567	2928
serializer	4782	9533	1248	1756
xerces	24579	56252	10631	8120
eclipse	23607	95073	70016	9379
derby	69537	180428	85160	16381
xalan	57430	155866	164805	18669
antlr	62007	147014	105861	17177
batik	56877	235085	145126	20901

2.4 RQ3:Optimization

We use a split algorithm to merge the node that have the same behavior. Thus we can reduce the space consumption. In the third reseach question, we calculated the number of relation nodes before and after optimization. Besides, time consumption is another factor that we consider. The results are showed in Tabel 3. We evaluated our optimization algorithm on all benchmarks 10 times and sucessfully exceuted our approach on 7 benchmarks. It shows that we can reduce the space by about 45% on average, with a reasonable time consuming. Another 6 benchmarks can not be execute because the same reason metioned above with PTA. We put the label "N/A" on Table 3.

Table 3. Optimalization result

bench	**Node**$_{origin}$	**Node**$_{opt}$	**Reduce**	**Time**(s^{-1})
compress	205	130	36.59%	0.008
crypto	312	158	49.36%	0.010
bootstrap	514	279	45.72%	0.019
commons-codec	1742	886	49.14%	0.202
junit	5859	3243	44.65%	2.269
commons-httpclient	9708	5164	46.81%	4.651
serializer	9600	6668	30.54%	3.198
xerces	41634	N/A	N/A	N/A
eclipse	34631	N/A	N/A	N/A
derby	112265	N/A	N/A	N/A
xalan	103697	N/A	N/A	N/A
antlr	56589	N/A	N/A	N/A
batik	89336	N/A	N/A	N/A