A Relational Static Semantics for Call Graph Resolution

June 18, 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 Optimazation

Algorithm 1 split

Require: valueToSplit, include=(), exclude=(), usedSplitter=()
  isSplit=false
  while true **do**
    aaa
  end while

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.

Efficiency: 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 0.029 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, so 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	0.12	0.013	87	202	24
crypto	0.01	0.12	0.028	94	226	18
bootstrap	23.74	34.13	0.029	191	453	17
commons-codec	0.009	0.12	0.131	306	3324	49
junit	24.45	34.24	0.153	1075	5772	245
commons-httpclient	0.009	0.12	0.362	2423	8511	597
serializer	22.60	32.68	1.704	3006	17726	368
xerces	21.69	34.83	3.434	12590	72503	3660
eclipse	21.95	42.64	1.671	7933	37435	2078
derby	22.77	48.82	17.781	20698	191854	6236
xalan	78.40	N/A	28.901	32971	162249	4438
antlr	44.20	N/A	3.73	16117	76741	6014
batik	45.84	N/A	5.031	29409	122534	9569

Accurancy: We consider generated callsite as the aspect of accurancy. Table 3 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 points-to analysis.

Optimazation: We use a split algorithm to merge the node that have the same behavior. Thus we can reduce the space. The results are showed in Tabel 2. We evaluated our optimazation algorithm on all benchmarks and sucessfully exceute 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 2.

Table 2. 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%	0.023
commons-codec	1742	886	49.14%	0.189
junit	5859	3243	44.65%	2.301
commons-httpclient	9708	5168	46.77%	4.873
serializer	9600	6671	30.51%	3.416
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

Table 3. Callsite generated with different analysis

bench	**CS**$_{origin}$	**CS**$_{cha}$	**CS**$_{pta}$	**CS**$_{tfa}$
compress	153	160	18	73
crypto	302	307	62	121
bootstrap	657	801	891	328
commons-codec	1162	1372	270	442
junit	3196	17532	11176	1358
commons-httpclient	6817	17118	567	2927
serializer	4782	9533	1248	1756
xerces	24579	56252	10631	8111
eclipse	23607	95073	70016	9379
derby	69537	180428	85212	16381
xalan	57430	155866	N/A	18669
antlr	62007	147014	N/A	17177
batik	56877	235071	N/A	20901