Points-to Analysis for Context-Oriented JavaScript Programs

Sergio Cardenas
Universidad de los Andes
Colombia
se.cardenas@uniandes.edu.co

Paul Leger
Universidad Católica del Norte
Chile
pleger@ucn.cl

Hiroaki Fukuda
Shibaura Institute of Technology
Japan
hiroaki@shibaura-it.ac.jp

Nicolás Cardozo
Universidad de los Andes
Colombia
n.cardozo@uniandes.edu.co

ABSTRACT

Static analyses, as points-to analysis, are useful to determine and assure different properties about a program, such as security or type safety. While existing analyses are effective in programs restricted to static features, precision declines in the presence of dynamic language features, and even further when the system behavior changes dynamically. As a consequence, improved points-to sets algorithms taking into account such language features and uses are required. In this paper, we present and extension of the point-to sets analysis to incorporate the language abstractions introduced by context-oriented programming adding the capability for programs to adapt their behavior dynamically to the system's execution context. To do this, we extend WALA to detect the context-oriented language abstractions, and their representation within the system, to capture the dynamic behavior, in the particular case of the Context Traits JavaScript language extension. To prove the effectiveness of our extension, we evaluate the precision of the points-to set analysis with respect to the state of the art, over four context-oriented programs taken from the literature.

CCS CONCEPTS

• **Software and its engineering** → *Context specific languages*; **Software maintenance tools**; *General programming languages.*

KEYWORDS

Static analysis, Points-to analysis, Context-oriented programming

ACM Reference Format:
Sergio Cardenas, Paul Leger, Hiroaki Fukuda, and Nicolás Cardozo. 2023. Points-to Analysis for Context-Oriented JavaScript Programs. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605156.3606451

FTfJP '23, July 18, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0246-4/23/07…$15.00
https://doi.org/10.1145/3605156.3606451

1 INTRODUCTION

Points-to analysis computes the set of possible values that can be referenced by a pointer throughout a program's execution [12]. Effective and precise computation of points-to sets has proven useful in application domains such as security [8] or type checking [4]. There are different techniques for computing precise and scalable points-to analysis for a variety of languages (*e.g.,* Java, C, and C++), but the results depend on the specific language capabilities.

While it is possible to compute the points-to analysis for Object-oriented and imperative languages, analyzing languages with dynamic properties precisely is more challenging (Section 2). Moreover, many of the existing techniques are tailored to object systems, leaving out other modularity types that may be present in a language. JavaScript, for example, has dynamic capabilities and a flexible object model [9], such as run-time object construction, dynamic property access, dynamic script evaluation, and variable parameter lists. Computing points-to analysis in the presence of such dynamic features is challenging. There is little work in points-to analyses for JavaScript programs embracing all its dynamic capabilities [12].

New programming paradigms, like Context-oriented programming (COP) [6, 11], present new modularity features that are not normally conceived in existing points-to analyses. The case of COP is of particular interest given that it enables the capacity to dynamically change the behavior of a program based on its execution context. COP can drive the development of adaptive systems, which are the base of smart environments and Internet of Things (IoT) systems. These two characteristics highlight the need for appropriate techniques to analyze dynamic systems using different modularity approaches. Therefore, we identify two challenges for point-to analysis for COP: (1) imprecision for languages with dynamic capabilities, and (2) lack of support for advanced modularity mechanisms.

The purpose of this work is to improve the results of the points-to set analysis for JavaScript-based systems with dynamic characteristics. In particular, the points-to analysis presented here constitutes the first static analysis for COP (Section 3). To narrow down the problem, we implement an analyzer for a specific implementation of COP for JavaScript: Context Traits [5, 3]. The importance of this work is that the points-to analysis can in turn be used as input for other analyses like type checking [2], or completeness [19].

In Context Traits, adaptations occur in response to contexts (de)activation, triggering dynamic trait composition. Snippet 1 shows a COP program extract managing the behavior of a mobile phone according to the state of its battery (*i.e.,* the context). In

```
1 Phone = Trait ({
2   initialize: function() {
3     this.number = +57 601 2133927;
4     this.gps = ...
5 } });
6
7 BatterySaver = Trait ({
8   this.unavailableFeatures = [];
9   restrictFeatures: function() {
10    delete this.gps;
11    this.unavailableFeatures.push("gps");
12 } })
13
14 LowBattery = new cop.Context();
15 LowBattery.adapt(Phone, BatterySaver);
16
17 if(BatteryManager.EXTRA_LEVEL > threshold) {
18   LowBattery.activate();
19   Phone.restrictFeatures();
20 } else
21   LowBattery.deactivate();
```

Snippet 1: COP mobile phone in Context Traits

this example, depending on the battery level (Line 17), the phone may present different features like the new `unavailableFeatures` and absence of the `gps` properties. The problem that arises from adapting the behavior is that different parts of a program (independent to specific snippets) may be oblivious to property changes.

Current points-to analyses are not equipped with the capacity to detect the different implementations of a function or the dynamicity of objects' properties, leading to poor precision and wrong conclusions about function prerequisites. Extending point-to analysis to account for the dynamic features of COP can be useful to effectively analyze such programs, detecting bugs or vulnerabilities.

We validate the precision of our extension of point-to analysis through the comparison with the state-of-the-art over different applications gathered from the COP literature (Section 4).

2 BACKGROUND

This section presents an overview of existing techniques used for points-to analysis, as the base of the approach used in our work.

Points-to analysis is a family of static analyses, which is used to approximate a set of possible targets (called points-to set) for each pointer in a program. There are many variations of points-to analysis [12], with additional techniques to improve precision in exchange for used resources and vice-versa. Snippet 2 shows a program example in Java-like syntax used to illustrate the differences between multiple points-to analyses.

```
1 Object first(Object o1,Object o2){ return o1; }
2 Object second(Object o1,Object o2){ return first(o2,o1)
      ;}
3 void f() {
4   Object o1 = new Object();
5   Object o2 = new Object();
6   Object o3 = first(o1, o2);
7   Object o4 = first(o2, o1);
8   Object o5 = second(o1, o2);
9   Object o6 = second(o2, o1);
10 }
```

Snippet 2: Example program to calculate points-to sets

The points-to sets, `pts(function/object)`, with a perfect precision for the program in Snippet 2 are as follows, where the labels `(instance 1)` and `(instance 2)` differentiate object allocations:

```
pts(f/o1) = {new Object() (instance 1)}
pts(f/o2) = {new Object() (instance 2)}
pts(f/o3) = {new Object() (instance 1)}
pts(f/o4) = {new Object() (instance 2)}
pts(f/o5) = {new Object() (instance 2)}
pts(f/o6) = {new Object() (instance 1)}
pts(first/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(first/o2) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o2) = {new Object() (instance 1), new Object() (instance 2)}
```

Subset-based and unification-based analysis. The subset-based points-to analysis solves necessary subset constraints [1]. Subset constraints on two points-to sets `A` and `B` are of the form "`A` is a subset of `B`". Unification-based analyses use equality constraints on points-to sets [14]. The unification-based analysis unifies points-to sets (in an assignment instruction), while subset-based analyses introduce a subset-constraint. The unification approach is more imprecise than the subset approach, in exchange of performance. The context-insensitive subset-based points-to sets are:

```
pts(f/o1) = {new Object() (instance 1)}
pts(f/o2) = {new Object() (instance 2)}
pts(f/o3) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o4) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o5) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o6) = {new Object() (instance 1), new Object() (instance 2)}
pts(first/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(first/o2) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o2) = {new Object() (instance 1), new Object() (instance 2)}
```

Each method is analyzed only once in a context-insensitive analysis, so the analysis concludes that the return of the first function could point to both object instances. This results in the points-to sets for `o3`, `o4`, `o5`, and `o6` containing two instances.

```
pts(f/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o2) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o3) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o4) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o5) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o6) = {new Object() (instance 1), new Object() (instance 2)}
pts(first/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(first/o2) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o2) = {new Object() (instance 1), new Object() (instance 2)}
```

As we can see above, the context-insensitive unification-based points-to sets analysis concludes that every variable can point to both object instances, as `o1` unifies with `o2` through the `first` method, and then unified with the rest of the variables through `f`.

Context Sensitivity. This is used to improve analysis precision by analyzing a method multiple times depending on how it is invoked. Since the same method can have different behavior in each invocation, a context-sensitive analysis can produce different results for each invocation, thus potentially improving the analysis' precision. There are many ways to implement context sensitivity, all with different precision and performance because it depends on the selected context abstraction.

A parallel implementation [15] to compute a reachability-based context-sensitive flow-sensitive points-to analysis uses data sharing and query scheduling for parallel graph traversals; allowing a significant improvement over sequential approaches. Li et al. [7] present an approach for improving the performance of context-sensitive points-to analysis without sacrificing too much precision. The authors achieve this by identifying precision-critical methods and apply context sensitivity only to those methods. Wei et al. [17] design a points-to analysis that applies different kinds of context sensitivity to different sections of a JavaScript program. Their

approach first computes a points-to analysis to identify characteristics of all the present functions, and then decides which context sensitivity technique to apply for each function.

Call-site context-sensitive analyses use the call site in which a method is called as an abstract context [12]. Different invocations of a method can have the same call site, so it is possible to store the call site of the analyzed method, and the call site of the caller, up to an invocation chain of depth k. This is known as k-call-site context sensitivity, or k-CFA. The k-CFA for our example program is:

```
pts(f/o1) = {new Object() (instance 1)}
pts(f/o2) = {new Object() (instance 2)}
pts(f/o3) = {new Object() (instance 1)}
pts(f/o4) = {new Object() (instance 2)}
pts(f/o5) = {new Object() (instance 1), new Object() (instance 2)}
pts(f/o6) = {new Object() (instance 1), new Object() (instance 2)}
pts(first/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(first/o2) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o1) = {new Object() (instance 1), new Object() (instance 2)}
pts(second/o2) = {new Object() (instance 1), new Object() (instance 2)}
```

In this case, `o3` and `o4` point to a single instance each. For Lines 6 and 7 of the example, the method `first` is analyzed separately for each instruction, since the call site is different. However, the points-to sets for `o5` and `o6` still have both object instances due to the call to the `first` method through the `second` method. This result in two different calls to the `first` method with the same call site. This imprecision can be fixed using a bigger k. With $k = 2$, we reach perfect precision of the points-to sets.

Object sensitivity is a popular choice when analyzing object-oriented programming languages, using the allocation site of the receiver object as a context [12]. The use of only the allocation site of the receiver makes the analysis more imprecise as the number of abstraction layers increases in the program. To solve this, the context can also have information about the allocation site of its caller's receiver (with depth k) in k-object sensitivity.

Flow Sensitivity. A flow-sensitive analysis is capable of taking into account statements' order, and branching. A flow-sensitive analysis produces more precise results than a flow-insensitive one, but has an additional overhead, since it needs to store different points-to sets for different program locations, incurring in greater space complexity [1]. Partial flow sensitivity is an alternative to full flow sensitivity that provides the scalability of flow-insensitive analyses while maintaining most of the precision benefits of flow sensitivity by reducing the control-flow graph, given that there are non-critical nodes in the original control-flow graph [10]. Sui et al. [16] present a scalable flow-sensitive points-to analysis for multithreaded C programs. They perform multiple thread inference analysis and sparse analysis. Their solution is highly scalable, presenting a significant improvement in terms of execution speed compared to other analyses. Partial flow-sensitive, context-sensitive points-to algorithms [18] track object property updates more accurately. To achieve this, a *obj-ref state* is used as the variable type at specified execution points, thus a new type of object sensitivity.

3 POINTS-TO ANALYSIS FOR COP

Dynamic language features, such as dynamic property access, make programs more difficult to analyze precisely [12]. Therefore, many analyzers avoid dynamic language features by taking a subset of the language. Still, there is relevant work to accurately analyze certain language features. For example, correlation tracking accurately analyzes the correlated dynamic property access coding pattern for JavaScript programs [13].

We now present our extension of the points-to sets to account for the capabilities of COP. Our work extends the basic field-sensitive correlation tracking WALA[1] analysis to improve precision without sacrificing performance, taking into account COP abstractions and the dynamic properties of JavaScript.

Our implementation first finds adaptation tuples in the form $\langle context - trait - object \rangle$ for each `Context.adapt(object,Trait)` instruction in the program to keep track of context and trait instantiations. Second, we insert code for every instruction related to adaptations and context activation. The inserted code corresponds to a basic model of the behavior for these instructions.

3.1 Adaptation and Activation Finder

First, we gather information about the COP abstractions in the source code –that is, trait and context creation, and calls to the methods `adapt` and `activate`. For each new trait instance, we store its name (if it is assigned to a variable), the caller method, the first argument of the constructor, and the source code position where the instantiation occurs. For context creation, we store the context name, under the assumption that every context instance is stored in a variable. For `adapt` method calls, we store the context, the trait, the object being adapted, the caller method, and the source position of the instruction. Finally, for `activate` method calls, we store the context, the caller method and the source position.

3.2 Code Insertion

Having identified the COP abstractions, we now search for code fragments including adaptations and activations, and replace them with new generated instruction sets that model the behavior of the original instructions. To do this, we replace nodes in the generated AST (before the analysis) representing the original instructions with new nodes representing the inserted code.

Trait instantiation. Consider the phone example in Snippet 1, which contains two trait instances, `BaseBehavior` and `BatterySaver`. For each trait instance, we modified its AST node by adding a new `obj` property. This is necessary to access the parameter object passed to the trait constructor, giving us access to the object even if an activation is outside the scope of the call site in which the trait creation occurs. The new property opens up access whenever a context is activated to adapt an object. For the trait object modification, we must ensure that no other modifications use the same property name. Snippet 3 shows the result of the code insertion creating the `obj` parameter for each trait (Lines 3 and 6).

```
1 o1 = {  initialize: function() { ... }  };
2 Phone = Trait(o1);
3 Phone.obj = o1;
4 o2 = {  restrictFeatures: function() { ... }  };
5 BatterySaver = Trait(o2);
6 BatterySaver.obj = o2;
```

Snippet 3: Code insertion for trait creation

Trait instantiations may also occur in the return statement of a function (not being stored in a variable). For such cases, the trait is

[1] WAtson Libraries for Analysis (WALA): http://wala.sourceforge.net

stored in a temporary variable in order to write the `obj` property, to then return the complete trait. Snippet 4 shows the resulting code for trait instantiation in the return statement. Note the precision of this approach depends on the type of context sensitivity used. In Snippet 4 the fields of the object depend on the function parameter. If context sensitivity is used, the function will be analyzed for each possible parameter, resulting in the points-to sets of the object fields being of size one.

```
function makeTrait(msg) {
  var _obj1 = { getMsg: function() { return msg; } };
  var _trait1 = Trait(_obj1);
  _trait1.obj = _obj1;
  return _trait1;
}
```

Snippet 4: Code insertion for trait creation in return statement

Adaptations. The second type of instruction inserts code into the `adapt` function invocations for a context. In this case, we enhance adaptations with two additional properties, following the same idea as before. These properties are used to access the behavior upon context activation. Given that a context can have multiple adaptations, the property name that we insert is numbered for each adaptation. For example, the `Phone` object can be extended with two behavioral variations: `BatterySaver`, and `Emergency`. The adaptations `LowBattery.adapt(Phone, BatterySaver)` and `LowBattery.adapt(Phone, Emergency)` result in the objects in Snippet 5, containing both, the adapted object (`obj`), and the trait (`trait`) properties.

```
LowBattery.adaptation1 = {
  obj: Phone,
  trait: BatterySaver
}
LowBattery.adaptation2 = {
  obj: Phone,
  trait: Emergency
}
```

Snippet 5: Code insertion for adaptations

Activations. For context activation method calls, we insert the properties of the object passed as a parameter into the adapted object. This is just a basic model of what happens when a context is activated. Note that this model is still imprecise when multiple contexts are active at once, but still effective in capturing the different function implementations. Consider the context activations `HighBattery.activate()`, and `LowBattery.activate()`. The resulting inserted code is in Snippet 6. In this case we wrap correlated pairs (`obj`, `trait`) inside a function called for each property [13].

```
for(var prop1 in HighBattery.adaptation1.trait.obj) {
  (function (prop) {
    HighBattery.adaptation1.obj[prop] = HighBattery.
        adaptation1.trait.obj[prop] })(prop1); }

for(var prop1 in LowBattery.adaptation1.trait.obj) {
  (function (prop) {
    LowBattery.adaptation1.obj[prop] = LowBattery.
        adaptation1.trait.obj[prop] })(prop1); }
for(var prop2 in LowBattery.adaptation2.trait.obj) { ...
    }
```

Snippet 6: Code insertion for activations with correlated property accesses

3.3 Implementation

We extend the WALA field-sensitive analysis for JavaScript. WALA is a Java framework for the statically analysis of Java bytecode and JavaScript files. The core of WALA was initially implemented to analyze Java, but then extended with a JavaScript front-end. An advantage of using WALA is that we can reuse the model for certain JavaScript constructs. In order to extend the existing analysis, we create a `CAstRewriter` that receives the results of the adaptation and activation finder to rewrite sections of the generated AST prior to the analysis. We use Rhino to parse JavaScript, as it is the recommended parser for JavaScript in WALA. The analysis takes place at the Intermediate Representation (IR) level, in SSA form.

3.3.1 Adaptation and activation finder. In the IR, adaptations correspond to method calls represented by `JavaScriptInvoke` instructions with two arguments where the method's object is a context. After identifying adaptation instructions, we store the trait, the adapted object, and the method as an `Adaptation` object. Activations are identified as `JavaScriptInvoke` instructions named `"activate"` with no arguments, and where the method's object is a context. Activations are stored in an `Activation` object.

3.3.2 Code Insertion. To insert code in certain AST nodes within the WALA framework, we implement a CAST Rewriter. Our implementation takes the information given by the adaptation and activation finder to find which AST nodes need to be replaced. When copying nodes from one AST to another, trait instantiation, adaptation, or activation nodes are replaced for new nodes with the appropriate information. To find which nodes will be replaced by the CAST Rewriter, we pattern match over the source position of each instruction. For example, adaptation nodes correspond to instructions in which the source position of the node is equal to the source position of the adaptation instruction, and the node follows a structure with type `CALL` and 5 sub-nodes (Figure 1b).

Nodes' source position alone is not enough to identify the correct node type (*i.e.,* trait instantiation, adaptation, or activation instructions), since the father node could also have the same source position. Therefore, we use hash maps to save the relations between a node and the instruction type it represents.

Trait instantiation. Figure 1a shows the AST node to represent traits, and Figure 1d the corresponding new node with the inserted trait reference. Note that we just replace the call to the Trait constructor and not the whole assignment. The reason for this is that trait creation can occur without an assignment (*e.g.,* it could be the return of a function). For that reason, it is necessary to create variables to store both the created trait and the adapter object. The names for the new variables must differ for each trait instantiation, to ensure each variable points to a single object. Finally, the node `<obj>` is stored and copied in the new AST. The `<obj>` node can be a variable or a new object; either way the result is unaffected.

Adaptation. The AST for adaptation message calls is shown in Figure 1b. This node is replaced by the node in Figure 1e. For adaptations, we match the node pattern in Figure 1b and check if the position of the AST node and the adaptation call are the same. The method name is verified by the adaptation finder, while calls to other functions in the same source code line would have different

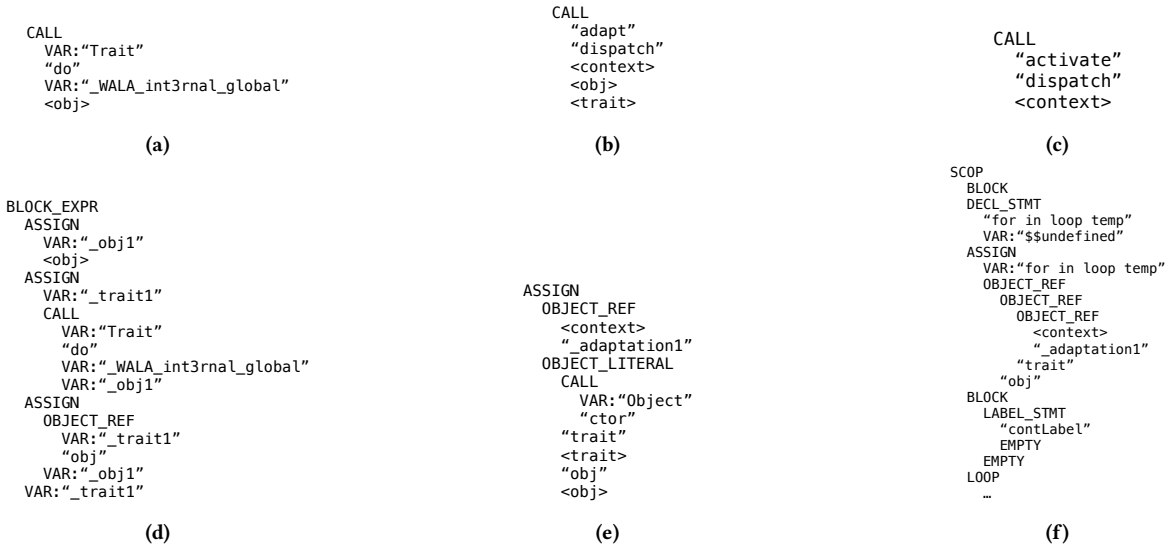

Figure 1: IR AST replacement for the trait instantiation, adaptation, and activation COP instructions

positions (same line but different offsets). The new node contains a copy of the <context>, <obj>, and <trait> nodes, with the additional references to the corresponding objects.

Activation. Activation nodes' pattern is shown in Figure 1c. For each adaptation, the activated context node is replaced by the structure in Figure 1f. Replacing activation nodes requires more work than trait instantiation or adaptations.

The information about the adaptations to trigger is gathered by the adaptation and activation finder. We use a block statement to group all the adaptations to be applied. In this case we copy only the context variable name to the new node. As mentioned before, we can assume the context to activate is stored in a variable. Otherwise, there would not be any adaptation to apply.

Note that beyond single node modifications to the AST, we need to add new control flow edges to the graph. The first thing we need to do is add edges to the node EXCEPTION_TO_EXIT (*i.e.,* a node that represents ending the execution due to an exception, specific to WALA) for each VAR, CALL, OBJECT_REF, and EACH_ELEMENT_GET node. We also add edges for the anonymous functions that are called for each adaptation when a context is activated.

4 EVALUATION

This section evaluates our points-to analysis for dynamic language features together with COP language abstractions. In particular, we evaluate the precision of our approach with respect to existing points-to analysis for JavaScript programs and the algorithms' performance.

4.1 Experimental Design

We compare our analysis with the standard field-sensitive WALA analysis for JavaScript. We evaluate our implementation on four COP programs[2] gathered from the literature (representative of COP interactions): (**A.1**) a basic multi-language hello-world program, (**A.2**) a shape area and perimeter calculator, (**A.3**) a video encoder, and (**A.4**) a course management system. These programs are used as examples of the different Context Traits functionalities. As system metrics, we take into account the number of nodes and edges of the call-graph, the preprocessing time (adaptation and activation finder, and AST rewriter), and the analysis time (call-graph and points-to sets computation). We choose small examples for the evaluation to assess the precision of our approach, manually inspecting the generated points-to sets for given variables and fields.

[2]Code examples are available at: https://github.com/FLAGlab/AdaptiveSystemAnalysis

4.2 Precision Analysis

We now proceed to evaluate the precision of our implementation. Due to space restrictions, we only discuss in detail the results for the greetings.js (**A.1**) program in Snippet 7, but similar analysis and results apply to the other programs.

```
1  Person = { greetings : function(){ return "Hello!"; }};
2
3  Spanish = new cop.Context();
4  SpanishSpeaking = Trait({
5    greetings: function() { return "Hola!"; }
6  });
7
8  French = new cop.Context();
9  FrenchSpeaking = Trait({
10   greetings: function() { return "Bonjour!";  }
11 });
12
13 Spanish.adapt(Person, SpanishSpeaking);
14 French.adapt(Person, FrenchSpeaking);
15
16 var result = Person.greetings();
17 Spanish.activate();
18 result = Person.greetings();
19 French.activate();
20 result = Person.greetings();
```

Snippet 7: greetings.js example program

The greetings.js program has a Person object (Line 1) with a base implementation of the greetings function. There are two adaptations for Person that modify the implementation of the greetings function. Let us calculate the points-to set for the result variable in Line 16.

```
pts(result) = {"Hello!"}

pts(Person.greetings) =
{<JSFunction
greetings.js@61:greetings>}
```

(a)

```
pts(result) = {"Hello!", "Hola!", "Bonjour!}

pts(Person.greetings) = {<JSFunction
greetings.js@61:greetings>,
    <JSFunction greetings.js@167:greetings>,
    <JSFunction greetings.js@273:greetings>}
```

(b)

Figure 2: Results of the points-to sets for the `greetings` example

Figure 2 shows how our implementation (b) is able to compute the complete points-to set for all possible values of the `result` variable. A similar situation occurs calculating the points-to set for the `greetings` field of the `Person` variable, where only one call site is detected by the baseline (WALA) (a) analysis, while all three call sites are detected using our approach.

Given that the baseline analysis can miss specific call sites when calculating the points-to sets for a variable, we calculate the recall of the points-to set algorithms, for different variables in each of the programs. Table 1 shows the baseline's and our approach's results. Note that our algorithm is effective in identifying all possible (context) values for variables in most cases. For the cases with low precision, it is as good as the baseline in identifying the call sites.

Table 1: Results of the points-to sets for the `greetings` example

Program	variable	instances	TP		FN		Recall	
			baseline	ours	baseline	ours	baseline	ours
A.1	result	3	1	3	2	0	0.33	**1**
	greetings	3	1	3	2	0	0.33	**1**
A.2	area	3	1	2	1	0	0.50	**1**
	sides	3	1	3	2	0	0.33	**1**
A.3	received_msg	3	1	1	2	2	**0.33**	**0.33**
A.4	marks	1	0	1	1	0	0	**1**
	i4[0]	1	0	0	1	1	**0**	**0**
	o1.course	2	1	2	1	0	0.50	**1**

The imprecision of our approach for programs A.3 and A.4 is due to the specific implementation of the variables. In the first case, we fail to identify the `proceed` calls, and therefore, miss possible values. In the second case, our algorithm detects different possible values coming from adaptations, as different possible instances of a value.

4.3 Performance Analysis

We now evaluate the performance of our solution with respect to existing point-to analyses. The performance experiments were executed using a PC with an AMD Ryzen 5 3500U, with base frequency of 2.10 GHz, and 8 GB 2400 MHz RAM.

The top part of Table 2 shows the measurements of all four programs using the baseline WALA analysis. The bottom part shows the measurements using our analysis.

Comparing the baseline analysis with our implementation, the first thing to note is that the size of the callgraphs are always slightly bigger. One reason for this is that the callgraphs of our implementation include the functions that could be adapted into an object when a context is active. To illustrate this, the method `greetings` in Snippet 7 can have 3 possible implementations depending on which context is activated. Our approach can conclude that this method has those 3 implementations, resulting in 4 more nodes and edges in the callgraph (2 for the extra implementations and 2 for the functions for correlated pairs inserted when a context is activated). Note that the preprocessing time is higher for our approach compared to the baseline, due to the overhead of computing possible adaptations and activations. Nonetheless, the analysis time is lower, making the total time about the same for the baseline and our implementation.

Table 2: Performance results (in ms) of the four programs

	Program	No. of nodes	No. of edges	Preprocessing	Analysis	Total time
baseline	A.1	116	115	**665 ± 25.4**	398 ± 10.4	1063
	A.2	125	124	**688 ± 10.7**	391 ± 34.1	**1079**
	A.3	115	114	**647 ± 20.4**	372 ± 26.5	**1019**
	A.4	150	149	**688 ± 13.6**	436 ± 26.7	**1124**
ours	A.1	120	119	727 ± 29.9	**321 ± 25.3**	**1048**
	A.2	131	130	791 ± 37.2	**367 ± 17.4**	1158
	A.3	119	118	751 ± 35.8	**316 ± 18.4**	1067
	A.4	159	158	857 ± 35.8	**383 ± 12.6**	1240

While our implementation over-approximates the points-to sets, it has a better precision than the other analyses, at a comparable performance. We observe how this improvement in precision can be beneficial for further analyses (*e.g.,* type checking). Our implementation still has sources of imprecision as not all COP capabilities are included. Still, the implemented analysis can be useful for COP developers, as for example when reasoning about the types or interfaces of behavior variations.

5 CONCLUSION AND FUTURE WORK

This work is motivated by the imprecision and bad performance of state-of-the-art analyses (*e.g.,* context-sensitive flow-sensitive) in the presence of dynamic language features, or new language abstractions, as in the case of COP. Moreover, imprecise and unsound results can lead to further errors in type checking and bug detection analyses. Our work presents an extension to the points-to analysis to improve the points-to sets precision. Our solution extends the standard field-sensitive WALA analysis for JavaScript, that statically models some of the COP capabilities, realized in the specific case of Context Traits.

The proposed analysis is defined in two phases. First, we implement the adaptation and activation finder, which analyzes the source code in search of trait instantiations, adaptations, and context activations. Second we use a code insertion algorithm to use the information gathered by the adaptation and activation finder to rewrite nodes representing COP features in the program's AST with a new proposed model for each feature. The resulting AST is then analyzed using the standard JavaScript WALA analysis. Our extension gives us better precision for basic programs, and a good performance in all cases. However, the precision results for more complex programs did not improve significantly due to the use of context-traits instructions not yet covered by the analysis. While the evaluation can extend to more complex programs, the examples used represent many of the interactions existing in COP. Therefore, we can conclude our results show the promise and impact this type of analysis can have for COP programs (*e.g.,* in type checking).

The work on analysis of COP systems can be extended to lift our proposal as an analysis framework for modular and adaptive software systems in the following three directions. (1) We can extend the coverage of COP abstractions as to improve the precision of more complex programs. (2) Use flow-sensitivity to take into account the order in which context activations/deactivations take place. (3) Extend the analysis further to offer results about the completeness of execution traces upon context changes.

REFERENCES

[1] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis. University of Copenhagen, 1994.

[2] Tomoyuki Aotani, Tetsuo Kamina, and Hidehiko Masuhara. *Type-Safe Layer-Introduced Base Functions with Imperative Layer Activation*. Proc. of the Intl. Workshop on Context-Oriented Programming. COP'15. ACM, 2015, 8:1–8:7. ISBN: 978-1-4503-3654-3.

[3] Nicolás Cardozo and Kim Mens. *Programming language implementations for context-oriented self-adaptive systems*. Information and Soft. Technology 143 (2022), p. 106789. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2021.106789.

[4] Satish Chandra and Thomas Reps. *Physical type checking for C*. Proc. of the Workshop on Program analysis for software tools and engineering. 1999, pp. 66–75.

[5] Sebastián González et al. *Context Traits: dynamic behaviour adaptation through run-time trait recomposition*. Proc. of Intl. Conf. on Aspect-Oriented Software Development. AOSD'13. ACM, 2013, pp. 209–220. ISBN: 978-1-4503-1766-5. DOI: 10.1145/2451436.2451461.

[6] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. *Context-Oriented Programming*. Jour. of Object Technology 7.3 (Mar. 2008), pp. 125–151.

[7] Yue Li et al. *A principled approach to selective context sensitivity for pointer analysis*. ACM Trans. on Programming Languages and Systems 42.2 (2020), pp. 1–40.

[8] V Benjamin Livshits and Monica S Lam. *Finding Security Vulnerabilities in Java Applications with Static Analysis*. USENIX security Symp. 2005, pp. 271–286.

[9] Gregor Richards et al. *An analysis of the dynamic behavior of JavaScript programs*. Proc. of the Conf. on Programming Language Design and Implementation. 2010, pp. 1–12.

[10] Subhajit Roy and YN Srikant. *Partial flow sensitivity*. Intl. Conf. on High Performance Computing. HiPC'07. Springer. 2007, pp. 245–256.

[11] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. *Context-Oriented Programming: A Software Engineering Perspective*. Jour. of Systems and Soft. 85.8 (Aug. 2012), pp. 1801–1817. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.03.024.

[12] Yannis Smaragdakis, George Balatsouras, et al. *Pointer analysis*. Foundations and Trends in Programming Languages 2.1 (2015), pp. 1–69.

[13] Manu Sridharan et al. *Correlation tracking for points-to analysis of javascript*. Proc. of the European Conf. on Object-Oriented Programming. ECOOP'12. 2012, pp. 435–458.

[14] Bjarne Steensgaard. *Points-to analysis in almost linear time*. Proc. of the ACM Symp. on Principles of programming languages. 1996, pp. 32–41.

[15] Yu Su, Ding Ye, and Jingling Xue. *Parallel pointer analysis with CFL-reachability*. Intl. Conf. on Parallel Processing. IEEE. 2014, pp. 451–460.

[16] Yulei Sui, Peng Di, and Jingling Xue. *Sparse flow-sensitive pointer analysis for multithreaded programs*. Proc. of the Intl. Symp. on Code Generation and Optimization. 2016, pp. 160–170.

[17] Shiyi Wei and Barbara G Ryder. *Adaptive context-sensitive analysis for JavaScript*. European Conf. on Object-Oriented Programming. ECOOP'15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2015.

[18] Shiyi Wei and Barbara G Ryder. *State-sensitive points-to analysis for the dynamic behavior of JavaScript objects*. Proceedigns of the European Conf. on Object-oriented Programming. ECOOP'14. Springer. 2014, pp. 1–26.

[19] Jingling Xue and Phung Hua Nguyen. *Completeness analysis for incomplete object-oriented programs*. Intl. Conf. on Compiler Construction. CC'05. Springer. 2005, pp. 271–286.

Received 2023-05-26; accepted 2023-06-23