Points-to Analysis for Context-Oriented JavaScript Programs

Sergio Eduardo Cárdenas Landazábal

Department of Systems and Computing Engineering
Universidad de los Andes
Colombia
2022

Contents

Chapter 1

Introduction

Static program analysis is a topic widely researched, as it is useful for a variety of applications. Points-to analysis is a type of static analysis which computes the set of possible values each pointer of the program can point to at any given point of execution [1]. Computing points-to analysis has been demonstrated to be really useful, as it has a multiple applications, such as security [2] or type checking [3], so it is of great interest to compute a precise points-to analysis. There is a vast amount of techniques for computing precise and scalable points-to analysis for a variety of languages such as Java, C and C++, but the result of applying those techniques varies depending on the language and its capabilities.

JavaScript is an extremely popular language for both front-end and back-end development, and is attractive mainly due to its dynamic capabilities and its flexible object model [4]. Run-time object construction, dynamic property access, dynamic script evaluation and variable parameter list are some examples of such capabilities. Due to those dynamic features, computing points-to analysis for JavaScript programs is a challenge on itself. There has been many research on points-to analysis for regular object-oriented languages, but very little for JavaScript that does not ignore its dynamic capabilities [1].

Context-oriented programming (COP) is a relatively new programming paradigm that enables the capacity to dynamically change the behavior of a program based on its execution context [5]. There are multiple implementations of COP libraries that give languages context-oriented capabilities [6] [7] [8]. In this thesis, we will focus in the context-traits library [8]. In the context-traits library, traits are the units that enable adaptation when contexts are activated, so multiple traits can compose a single object at any given point [8].

```
1  _obj1 = {
2      initialize: function() {
3          this.init = true;
4      }
5  };
6  Trait1 = Trait(_obj1);
7  Trait1.obj = _obj1;
8
9  Trait2 = Trait({
10     other: function() {
11         return 0;
12     }
13 })
14
15 LowBattery = new cop.Context();
16 NormalBattery = new cop.Context();
17
18 smth = {
19     initialize: function() {
20         this.init = 0;
21     }
22 };
23 NormalBattery.adapt(smth, Trait1);
24 NormalBattery.adaptation1 = {
25     obj: smth,
26     trait: Trait1
27 };
28 LowBattery.adapt(smth, Trait2);
29
30 if(Math.random() > 0.5) {
31     LowBattery.activate();
32     for(_props1 in LowBattery.adaptation1.trait.obj) {
33         function (p) {
34             LowBattery.adaptation1.obj[p] =
35                 LowBattery.adaptation1.trait.obj[p];
36         }(_props1);
37     }
38 }
39 else {
40     NormalBattery.activate();
41 }
42 smth.initialize()
```

Snippet 1.1: contex-oriented JavaScript program example

Snippet 1.1 shows a small JavaScript program that uses context traits. Depending on the result of Math.random(), half of the times the property smth.init will be true and half of the times it will be 0. The problem that arises in this case is that the property init may be required to be of a specific type as an prerequisite for the program to run successfully. Current static analyzers would not be able to conclude that the initialize function has more than one implementation, leading to not only poor precision but also wrong conclusions about this prerequisite. As COP has been implemented for JavaScript as a library, it is possible to run existing implementations of points-to analysis to analyze programs that use COP in JavaScript, but the precision of the results would be low due to the little to no information about the COP implementation the

analyzer has. An unsound and imprecise analysis can lead to wrong conclusions about bug finding or type checking.

The purpose of this work is to improve the results of analyzers that could use the points-to sets as an input such as type checking by improving the precision of a points-to analysis for context-oriented JavaScript programs. We implement points-to analysis for context-oriented JavaScript programs, that consider both dynamic capabilities of the language and functionalities of COP. To narrow down the problem, we implement the analyzer for a specific implementation of COP for JavaScript. Right now, COP developers do not have tools for analyzing their programs accurately. It has been demonstrated how analyzing a program can be useful for many applications [1], but COP developers do not have access to these applications due to the poor precision in static analyses. This work can be useful for COP developers to analyze their programs more effectively in search for bugs or vulnerabilities thanks to having better precision when statically analyzing their programs. Chapter 2 presents previous works regarding points-to analysis and context-oriented programming. In Chapter 3 you can find our analysis proposal and implementation. Chapter 4 presents the results of the evaluation of the implemented analysis, comparing its precision with standard analyses. Finally, conclusions and future work are presented in Chapter 5.

Chapter 2

Background and Related Work

2.1 Points-to Analysis

Points-to analysis is a family of static analyses, whose goal is to approximate a set of possible targets (called points-to set) for each pointer in the program. There are a lot of possible variations for a points-to analysis implementation and numerous techniques that improve precision in exchange for used resources and vice-versa. Many of these variations are described by Smaragdakis and Balatsouras [1]. We will position our implementation in perspective of the existing points-to analysis categories in the literature. Snippet 2.1 shows a program example in Java-like syntax that will help us illustrate the differences between multiple variations of points-to analysis.

```
1  Object first(Object o1, Object o2) {
2      return o1;
3  }
4
5  Object second(Object o1, Object o2) {
6      return first(o2, o1);
7  }
8
9  void f() {
10     Object o1 = new Object();
11     Object o2 = new Object();
12     Object o3 = first(o1, o2);
13     Object o4 = first(o2, o1);
14     Object o5 = second(o1, o2);
15     Object o6 = second(o2, o1);
16 }
```

Snippet 2.1: Program Example

In the example above, the points-to sets, pts(function/object), with a perfect precision would be the next ones:

```
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)}
```

We are using the labels (instance 1) and (instance 2) to differentiate between different allocations of objects

2.1.1 Subset-based and unification-based

An Andersen-style points-to analysis is subset-based analysis where necessary subset constraints are solved [9]. Subset constraints are inferences about two points-to sets A and B in the form of "A is subset of B". Steengaard-style analyses are unification-based and use equality constraints [10]. Where an Andersen-style analysis would introduce a subset-constraint (e.g., in an assignment instruction), a Steengaard-style analysis would unify both points-to sets, producing more imprecise results against an Andersen-style analysis in exchange for performance. The resulting points-to sets for the example in Snippet 2.1 for a context-insensitive Andersen-style analysis 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, meaning that 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, and o5 containing both instances. Let's compare these results with the ones produced by a context-insensitive Steensgaard-style analysis:

```
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)}
```

This time, the analysis concludes that every variable can point to both instances. This occurs because o1 is unified with o2 through the first method, and then unified with the rest of the variables.

2.1.2 Context Sensitivity

Context sensitivity is another way of improving 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 results against a context-insensitive analysis. There are many ways of implementing context sensitivity, all with different precision and performance, depending on what is selected as context abstraction. We will present some of the types of context sensitivity found in literature later.

In the field of context-sensitive analysis, there is a lot of work with different context abstractions. Su et al. [11] implemented a parallel solution to computing CFL-reachability-based context-sensitive flow-sensitive points-to analysis. Their solution uses data sharing and query scheduling for parallel graph traversals, and allows a significant improvement over sequential approaches. Li et al. [12] present an approach for improving the performance of context-sensitive points-to analysis in terms of speed without sacrificing too much precision. They achieve this by identifying precision-critical methods and applying context sensitivity only to those methods. Wei and Ryder [13] design a points-to analysis that applies different kinds of context sensitivity to different sections of code for JavaScript programs. 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 sensitivity

call-site context-sensitive analyses use the call site in which a method is called as an abstract context [1]. Different invocations of the method can have the same call site, so it is possible to store in a string not only the call site of the analyzed method, but also the call site of its caller, the call site of the caller method's caller, and so on, with a depth k. This is also known as k-call-site context sensitivity, or k-CFA. For the program in Snippet 2.1, the resulting points-to sets for a 1-CFA analysis would be:

```
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)}
```

This time, o3 and o4 point to a single instance each. For lines 12 and 13 of Snippet 2.1 the method first would be analyzed separately for each line, since the call site is different. However, the points-to sets for o5 and o6 still have both instances due to calling the first method through the second method, which means that two different calls of the first method have the same call site (line 6). This imprecision can be fixed

using a bigger k. In fact, for k = 2, the resulting points-to sets of the analysis are the next ones:

```
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)}
```

Object sensitivity

Object sensitivity is a popular choice when analyzing object-oriented programming languages. The allocation site of the receiver object is used as a context [1]. The use of only the allocation site of the receiver makes the analysis more and more imprecise as more layers of abstraction are added into the program. To solve this, the context can also have information about the allocation site of its caller's receiver, the caller's caller's receiver, with depth k, also called k-object sensitivity

Type sensitivity

Type-sensitive analyses use the type/class of the receiver object as a context instead of the allocation site of the receiver object [1]. This improves scalability at the cost of losing precision. Type sensitivity is only applicable for languages with explicit types such as Java.

2.1.3 Flow Sensitivity

A flow-sensitive analysis is capable of taking into account the order of statements as well as branching when analyzing. In other words, the control flow takes an important role in the analysis. It also computes the results for different points in the program. 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 locations of the program, incurring in greater space complexity [9]. An approach to have flow sensitivity without excessive overhead is partial flow sensitivity. 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 there are non-critical nodes in the original control-flow graph [14].

Sui et al. [15] present a scalable flow-sensitive points-to analysis for multithreaded C programs. To achieve this, 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. Wei and Ryder [16] present a partial

flow-sensitive, context-sensitive points-to algorithm for JavaScript programs that tracks object property updates more accurately. To achieve this, they use what they call "obj-ref state" as the type of a variable at a certain point of the execution, thus using a new type of object sensitivity.

2.1.4 Analyzing dynamic features

Dynamic language features makes programs more difficult to analyze precisely. For instance, JavaScript is one of the language with many dynamic features such as dynamic property access. That is why many analyzers for those languages just take a subset of the language, ignoring dynamic features. Still, there is relevant work made to accurately analyze certain language features. Sridharan et al [17] describe a technique called correlation tracking that analyzes accurately the correlated dynamic property access coding pattern for JavaScript programs, and implemented this technique as an extension to WALA. We will use this technique to improve the precision of our analysis for this coding pattern without sacrificing performance.

Wei and Ryder [16] present a partial flow-sensitive, context-sensitive points-to algorithm for JavaScript programs that tracks object property updates more accurately. To achieve this, they use what they call "obj-ref state" as the type of a variable at a certain point of the execution, thus using a new type of object sensitivity. Smaragdakis and Balatsouras [1] explore the different techniques employed for point-to analysis in a variety of languages. They point out that dynamic features in highly dynamic languages such as JavaScript are often ignored, but there have been certain works that try to statically model those dynamic features.

2.2 Context-Oriented Programming

Context-Oriented Programming (COP) is a programming paradigm that was originally aimed to support pervasive computing environments by incorporating context as a first-class construct of a programming language [18]. COP treats context explicitly, and provides mechanisms to dynamically adapt behavior in reaction to changes in context, even after system deployment at runtime [19]. One of the first implementations of COP is ContextL, an extension to the Common List Object System [6]. After that, there have been multiple implementation for a variety of languages, each one with different approaches about how to implement the dynamic behavior based on context [20].

To mention some recent work related to COP, Appeltauer et al. [7] present an implementation of COP for Java that incorporates layers into the language. Hirschfeld et al. [21] implemented a COP extension to Squeak/Smalltalk that include behavioral variations, layers, activations, and scopes. Elyasaf and Sturm [22] present a framework for analyzing context-oriented programming languages (COPLs) that would allow users to decide which COPL is able to satisfy their needs regarding the building of context-aware systems. González et al. [8] implement COP for JavaScript with traits as behavioral units of adaptation, with mechanisms to solve composition conflicts.

2.3 Discussion

There is no work in statically analyzing COP programs. As there are plenty of implementations of COPL and COP libraries for mainstream languages, it is not possible to write a global analyzer that could work for every possible implementation. In our case, we are aiming towards building an analyzer for the context-traits implementation [8] for JavaScript. Additional to that, analyzing dynamic features in languages like JavaScript is a challenge on itself, and proposed solutions for certain features do not have excellent performance.

COP implementations such as the one in [8] make use of many of those dynamic capabilities, resulting in poor precision or performance for static analyzers. Instead of having to implement an extremely precise analyzer for a highly dynamic language such as JavaScript that would require to model the majority of the dynamic features of the language for the analysis to produce useful results, a better approach would be to model the library itself in a simple way, with little overhead, that could produce good precision results. We extend the WALA framework to make an analyzer that models the behavior of some of the context-traits library instructions to be able to compute points-to sets with good precision and without much overhead.

Chapter 3

Methodology

Our goal is to improve thepoints-to sets precision by extending the basic field-sensitive WALA analysis to consider context-oriented instructions.

3.1 Analysis overview

Our implementation can be described in two steps: first, find adaptation tuples in the form of context-trait-object for each Context.adapt(object,Trait) instruction in the program. We will keep track of context and trait instantiations. The second step will be inserting code for every instruction related to adaptations and context activation. The inserted code will be a basic model of the behavior of these instructions. We describe each step in more detail.

3.1.1 Adaptation and activation finder

The first step in the analysis is to find context and trait creation and calls to the methods *adapt* and *activate* in the analyzed source code. For each new trait instance, the information about the trait name (if it is assigned to a variable), the caller method, the first argument of the constructor, and the position in the source code where the instantiation occurs. For context creation, only the context name is stored. We are making the assumption that every context instance is stored in a variable. This is a safe assumption to make, since it is not common to find a context instatiation instruction at the return instruction of a function, or use it as a function parameter. For calls to the method adapt, we store the context, the trait, the object being adapted, the caller method, and the source position of the instruction. Finally, for calls to the method activate, we store the context, the caller method and the source position. All this information is used as input for the next step in the analysis.

3.1.2 Code insertion

This fragment of the analyzer searches for sections of the code related to adaptations and activations, and replaces them with new generated instructions that model the behavior of the original ones, using the information gathered by the adaptation and activation finder. The types of instructions that are replaced are trait instantiations,

Context.adapt function calls, and context activations. We replace nodes in the generated AST (before the analysis) representing the original instructions with new nodes representing the inserted code.

Trait instantiation

Using the code example in Chapter 1 (Snippet 1.1), we have two trait instances, Trait1 and Trait2. The only inserted code for each instance is a property. This is necessary, as we will need to access the object passed as parameter of the trait constructor. This will allow us to have access to the object passed as parameter even if an activation is outside the scope of the call site in which the trait creation occurs. Having this property write will allow us to use a property read when a given context is activated in order to adapt an object. We are assuming there will not be another property write with the same property name anywhere in the code. This is a safe assumption, since a programmer would not be creating new properties for the trait.

Snippet 3.2 illustrates the result of the code insertion for the trait creations in Snippet 3.1. Apart from having a property write for each created trait, we need to declare the parameter object outside the trait creation.

```
1  Trait1 = Trait({
2      initialize: function() {
3          this.init = true;
4      }
5  });
6
7  Trait2 = Trait({
8      other: function() {
9          return 0;
10     }
11 });
```

Snippet 3.1: Trait creation

```
1  o1 = {
2      initialize: function() {
3          this.init = true;
4      }
5  };
6  Trait1 = Trait(o1);
7  Trait1.obj = o1;
8
9  o2 = {
10     other: function() {
11         return 0;
12     }
13 };
14 Trait2 = Trait(o2);
15 Trait2.obj = o2;
```

Snippet 3.2: Code insertion for trait creation

A trait instantiaton could be the return statement of a function and not stored in a variable at all. For those cases, the trait would be stored in a temporal variable in order

to perform the property write, and then returned. Snippets 3.3 and 3.4 show the trait instantiation instruction in the return statement and the result of the code insertion respectively.

```
function makeTrait(msg) {
    return Trait({
        getMsg: function() {
            return msg;
        }
    });
}
```

Snippet 3.3: Trait creation in return statement

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

Snippet 3.4: Code insertion for trait creation in return statement

The precision of this approach depends on the type of context sensitivity used. In Snippet 3.3; 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 (if the adequate type of context sensitivity is used).

Adaptations

The second type of instruction we insert code into is invocations of the adapt function for a context. We will be inserting another property write for adaptations. The reason is the same as before, we will be accessing these properties when a context is activated. This time, as a single context can have multiple adaptations, the property name we are writing into is numbered. The object we are writing into a property is a new object containing both the adapted object and the trait. Snippet 3.6 illustrates the result of the code insertion performed for snippet 3.5

```
NormalBattery.adapt(smth, Trait1);
NormalBattery.adapt(another, Trait1);
LowBattery.adapt(smth, Trait2);
```

Snippet 3.5: Adaptations

```
NormalBattery.adaptation1 = {
    obj: smth,
    trait: Trait1
};
NormalBattery.adaptation2 = {
    obj: another,
    trait: Trait1
```

```
};

LowBattery.adaptation1 = {
    obj: smth,
    trait: Trait2
}
```

Snippet 3.6: Code insertion for adaptations

Activations

The code insertion we will be performing on every invocation of the method activate of a context is described next: every time a context is activated, the properties of the object passed as a parameter of the trait constructor are written into the adapted object. This is just a basic model of what happens when a context is activated. Note that this model is imprecise when multiple contexts can be active at once.

Snippet 3.14 shows the result of the code insertion performed in snippet 3.7. Note that we are using a for-in loop for each adaptation. This could be a source of imprecision, so in order to solve this we wrap the correlated pairs inside a function that will be called for each property [17]. Snippet 3.9 shows the resulting code after the code insertion taking into account correlated pairs of property accesses.

```
NormalBattery.activate();
...
LowBattery.activate();
```

Snippet 3.7: Activations

```
for(var prop1 in NormalBattery.adaptation1.trait.obj) {
    NormalBattery.adaptation1.obj[prop1] = NormalBattery.adaptation1.
       trait.obj[prop1]
}

for(var prop2 in NormalBattery.adaptation2.trait.obj) {
    NormalBattery.adaptation2.obj[prop2] = NormalBattery.adaptation2.
       trait.obj[prop2];
}
...
for(var prop3 in LowBattery.adaptation3.trait.obj) {
    LowBattery.adaptation3.obj[prop3] = LowBattery.adaptation3.trait.
       obj[prop3];
}
```

Snippet 3.8: Code insertion for activations

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

for(var prop2 in NormalBattery.adaptation2.trait.obj) {
    (function (prop) {
```

```
9           NormalBattery.adaptation2.obj[prop] = NormalBattery.
                adaptation2.trait.obj[prop]
10      })(prop2);
11  }
12  ...
13  for(var prop3 in LowBattery.adaptation3.trait.obj) {
14      (function (prop) {
15          LowBattery.adaptation3.obj[prop] = LowBattery.adaptation3.
                trait.obj[prop]
16      })(prop3);
17  }
```

Snippet 3.9: Code insertion for activations with correlated property accesses

3.2 Implementation

As mentioned earlier, we are using the Watson Libraries for Analysis (WALA) and its base field-sensitive analysis for JavaScript to implement ours. WALA is a Java framework that allows users to statically analyze Java bytecode and JavaScript files [23]. The core of WALA was initially implemented to analyze Java, but then extended with a Front-end for JavaScript. 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.

3.2.1 Adaptation and activation finder

We do not analyze the source code line by line, but convert it to an intermediate representation (IR) in SSA form. With the IR of each methods, we find trait and context instantiations, adaptations, and activations.

```
1  ...
2  IRFactory<IMethod> factory = AstIRFactory.makeDefaultFactory();
3  SSAOptions ssaOptions;
4  ...
5  Map<IMethod,IR> methodsToAnalyze = HashMapFactory.make();
6  for (IClass klass : cha) {
7    if (!klass.getName().toString().startsWith("Lprologue.js")) {
8      for (IMethod method : klass.getAllMethods()) {
9        methodsToAnalyze.put(method,factory.makeIR(method, Everywhere.
             EVERYWHERE, ssaOptions));
10     }
11   }
12 }
```

Snippet 3.10: Converting methods to IR

We know a given instruction is an adaptation when it is an instruction of the type JavaScriptInvoke representing a method call, has two arguments, and the method's object is a context. After identifying a given instruction is an adaptation, we store the trait, the adapted object, and the method into an Adaptation object. As for activations,

the process is similar: first, find an instruction that is instance of JavaScriptInvoke, check if it is a method call, if its name is "activate", if it has 0 arguments, and if the method's object is a context. After checking that, the activation is stored in an Activation object.

```
1  public AdaptationSummary findAdaptations(IMethod method, IR ir, Set<
       Trait> traits, Set<ContextCop> contexts) {
2    OrdinalSetMapping<SSAInstruction> instrIndices = new
       ObjectArrayMapping<>(ir.getInstructions());
3    AdaptationSummary summary = new AdaptationSummary(ir.getMethod(),
       instrIndices);
4    for(SSAInstruction ssa : ir.getInstructions()) {
5      if(ssa != null) {
6        if(ssa instanceof JavaScriptInvoke) {
7          JavaScriptInvoke jsInvoke = (JavaScriptInvoke) ssa;
8          if(ir.getSymbolTable().isConstant(jsInvoke.getFunction())
9              && ir.getSymbolTable().getConstantValue(jsInvoke.
                   getFunction()).equals("adapt")
10             && jsInvoke.getCallSite().getDeclaredTarget().equals(
                   JavaScriptMethods.dispatchReference)
11             && jsInvoke.getNumberOfUses() == 4) {
12           int contextVar = jsInvoke.getUse(1);
13           int objVar = jsInvoke.getUse(2);
14           int traitVar = jsInvoke.getUse(3);
15           ContextCop contextFound = searchContext(ir.getInstructions()
                 , contexts, contextVar);
16           if(contextFound == null) continue;
17           Trait traitFound = searchTrait(ir.getInstructions(), traits,
                  traitVar);
18           String obj = searchGlobalRead(ir.getInstructions(), objVar);

19           summary.addAdaptation(AdaptationFactory.make(jsInvoke,
                 contextFound, traitFound, obj, ir.getMethod()));
20         }
21       }
22     }
23   }
24
25   return summary;
26 }
```

Snippet 3.11: Converting methods to IR

```
1  public ActivationSummary findActivations(IMethod method, IR ir,
       Collection<AdaptationSummary> adaptations) {
2    OrdinalSetMapping<SSAInstruction> instrIndices = new
       ObjectArrayMapping<>(ir.getInstructions());
3    ActivationSummary summary = new ActivationSummary(ir.getMethod(),
       instrIndices);
4    for(SSAInstruction ssa : ir.getInstructions()) {
5      if(ssa instanceof JavaScriptInvoke) {
6        JavaScriptInvoke jsInvoke = (JavaScriptInvoke) ssa;
7        if(ir.getSymbolTable().isConstant(jsInvoke.getFunction())
8            && jsInvoke.getCallSite().getDeclaredTarget().equals(
                 JavaScriptMethods.dispatchReference)
9            && jsInvoke.getNumberOfUses() == 2
```

```
10            && ir.getSymbolTable().isStringConstant(jsInvoke.getFunction
                  ())) {
11          String contextName = searchGlobalRead(ir.getInstructions(),
               jsInvoke.getUse(1));
12          String methodName = ir.getSymbolTable().getStringValue(
               jsInvoke.getFunction());
13          if(contextName != null && methodName.equals("activate")) {
14            ContextCop ctx = searchContextInAdaptations(contextName,
                 adaptations);
15            if(ctx != null) {
16              summary.addActivation(new Activation(ctx, jsInvoke, true,
                   method));
17            }
18          }
19
20        }
21      }
22    }
23
24    return summary;
25  }
```

Snippet 3.12: Converting methods to IR

3.2.2 Code insertion

To insert pieces of code in certain positions with the WALA framework, implementing a CAST Rewriter is a must. We implemented a CAST Rewriter that 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, the method checks if a given node corresponds to a trait instantiation, an adaptation, or an activation, and constructs a new node that will replace the current one depending on the type of instruction.

To find which nodes will be replaced by the CAST Rewriter, we use the source position of each instruction and pattern matching. For example, we know a given node corresponds to a certain adaptation instruction if the source position of the node is equals to the source position of the adaptation instruction and if the node has a certain structure (i.e. the node is of type CALL and has 5 sub-nodes). The source position alone is not enough to identify if the node corresponds to a trait instantiation, an adaptation, or an activation, since the father node could also have the same source position. We save the relations between a node and the type of instruction it represents in hash maps.

```
1  private static AdaptedObjectInsertionPolicy addAnalysis(CAstEntity
       entity,
2       Map<Position, AdaptationSummary> adaptationSumms, Map<Position,
           ActivationSummary> activationSumms,
3       Map<Position, TraitSummary> traitSumms,
           AdaptedObjectInsertionPolicy policy) {
4    CAstNode root = entity.getAST();
5    if(root != null) {
6      if (adaptationSumms.containsKey(entity.getPosition())) {
```

```
7         AdaptationSummary adaptations = adaptationSumms.get(entity.
              getPosition());
8         for(Adaptation adapt : adaptations.getAdaptations()) {
9           policy.addAdaptation(entity, adapt, adaptations);
10        }
11      }
12      if (activationSumms.containsKey(entity.getPosition())) {
13        ActivationSummary activations = activationSumms.get(entity.
              getPosition());
14        for(Activation activ : activations.getActivations()) {
15          policy.addActivation(entity, activ, activations);
16        }
17      }
18      if (traitSumms.containsKey(entity.getPosition())) {
19        TraitSummary traits = traitSumms.get(entity.getPosition());
20        for(Trait trait : traits.getTraits()) {
21          policy.addTrait(entity, trait, traits);
22        }
23      }
24    }
25    return policy;
26  }
```

Snippet 3.13: Converting methods to IR

Trait instantiation

Figure 3.1 shows the AST pattern we are matching with a given node to know if it represents a trait instantiation, and Figure 3.2 shows the AST node we are replacing the original node with.

```
CALL
  VAR
    "Trait"
  "do"
  VAR
    "__WALA__int3rnal__global"
  <obj>
```

Figure 3.1: Original AST for trait instantiation

Note that we are just replacing 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 name of those variables will be different for each trait instantiation to ensure each of those variables can only point to a single object. Finally the node <obj> is stored and copied afterwards in the new AST. The <obj> node can be a variable or a new object; either way the results would be unaffected.

```
BLOCK_EXPR
  ASSIGN
    VAR
      "_obj1"
    <obj>
  ASSIGN
    VAR
      "_trait1"
    CALL
      VAR
        "Trait"
      "do"
      VAR
        "__WALA__int3rnal__global"
      VAR
        "_obj1"
  ASSIGN
    OBJECT_REF
      VAR
        "_trait1"
      "obj"
    VAR
      "_obj1"
  VAR
    "_trait1"
```

Figure 3.2: rewritten AST for Trait instantiation

Adaptation

Figure 3.3 show the the AST pattern we are expected to find for each call to the adapt method in a context, that will be replaced by the AST in Figure 3.4.

```
CALL
  VAR
    adapt"
  "dispatch"
  <context>
  <obj>
  <trait>
```

Figure 3.3: AST for adaptations

We do not check if the method that is being called is the adapt method. We just need to match the pattern in Figure 3.3 and check if the position of the AST node and the adaptation are the same. The check for the method's name was done in the adaptation finder, and calls to other functions in the same line of the source code would have different positions (same line but different offsets). This time, we are copying the <context>, <obj>, and <trait> nodes to the new AST shown before.

```
ASSIGN
  OBJECT_REF
    <context>
    "_adaptation1"
  OBJECT_LITERAL
    CALL
      VAR
        "Object"
      "ctor"
    "trait"
    <trait>
    "obj"
    <obj>
```

Figure 3.4: rewritten AST for adaptations

Activation

The nodes that will be replaced must match the pattern shown in Figure 3.5. Replacing activation nodes requires more work than trait instantiation or adaptations. For each adaptation the activated context has, a node that matches what is shown in Figure 3.6 will be inserted. The information about what adaptations to trigger is compiled by the adaptation and activation finder. We use a block statement to group all of the adaptations that have to be applied.

```
CALL
  "activate"
  "dispatch"
  <context>
```

Figure 3.5: Original AST for activations

The only information copied from the original node to the new one is the context variable name. As mentioned before, we can assume the context being activated is stored in a variable. Otherwise, there will not be any adaptation to apply.

Copying nodes

For each node to be rewritten, we check if the node corresponds to a trait instantiation, an adaptation, or an activation, based on the node position and its pattern.

```
1 private CAstNode copyNode(CAstNode root, CAstControlFlowMap cfg,
     NonCopyingContext context,
2      Map<Pair<CAstNode, NoKey>, CAstNode> nodeMap) {
3   AdaptedObjectInsertionPolicy policy = policies.peek();
4   if(policy == null) throw new IllegalStateException("impossible to
       copy nodes without an insertion policy");
5   Adaptation adaptation = policy.getAdaptationFromNode(root);
6   if(adaptation != null) {
7     return buildAdaptationAst(root, cfg, context, nodeMap, adaptation,
          policy);
8   }
```

```
SCOPE
  BLOCK
    DECL_STMT
      "for in loop temp"
      VAR
        "$$undefined"
    ASSIGN
      VAR
        "for in loop temp"
      OBJECT_REF
        OBJECT_REF
          OBJECT_REF
            <context>
            "_adaptation1"
          "trait"
        "obj"
    BLOCK
      LABEL_STMT
        "contLabel"
        EMPTY
      EMPTY
    LOOP
      ...
    BLOCK
      LABEL_STMT
        "breakLabel"
        EMPTY
      EMPTY
```

Figure 3.6: AST for each active adaptation

```
9   Activation activation = policy.getActivationFromNode(root);
10  if(activation != null) {
11    return buildActivationAst(root, cfg, context, nodeMap, activation,
          policy);
12  }
13  Trait trait = policy.getTraitFromNode(root);
14  if(trait != null) {
15    return buildTraitAst(root, cfg, context, nodeMap, trait, policy);
16  }
17  return copySubtreesIntoNewNode(root, cfg, context, nodeMap);
18 }
```

Snippet 3.14: Converting methods to IR

Control flow graph

Since we are adding new nodes 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 nodes. We also add edges for the anonymous functions that are called for each adaptation when a context is activated.

Chapter 4

Evaluation

In this section, we will describe how we evaluate our solution against existing points-to analysis for regular JavaScript programs.

4.1 Experiments

We compare our analysis with the standard field-sensitive WALA analysis for JavaScript. We evaluate our implementation on the test programs found in Snippets A.1, A.4, A.3, and A.2. These programs use different functionalities of the context-traits library. The measured quantities are the number of nodes and edges of the callgraph, the preprocessing time (Adaptation and activation finder, and AST rewriter), and the analysis time (callgraph and points-to sets computation). As the analyzed programs are small, it is necessary to show the points-to sets for certain variables and fields. We will use our knowledge about the analyzed programs to determine the precision of the computed points-to sets.

4.2 Experiment results

4.2.1 Empirical evaluation

Now we proceed to evaluate the precision of our implementation. For that, we take a look at different points-to sets across the tested programs.

greetings.js

The first test program (Snippet A.1) has an object Person declared between Lines 3 and 7 and has a base implementation for the method greetings. There are two adaptations for the object Person that modify the implementation of the greetings function. For this program, we take a look at the variable result from Line 27. These are the resulting points-to sets for each evaluated analysis:

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

ours:
```

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

whole code:
pts(result) = {"Hello!"}
```

The first test program is the simplest of all, but it allow us to verify which analyses are able to take into account the possible implementations of the method greetings. Only our implementation was able to compute the correct points-to set for the variable result. Now, see the points-to sets for the field greetings of the object in the variable Person in Line 3:

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

ours:
pts(Person.greetings) = {<JS Function greetings.js@61:greetings>,
    <JS Function greetings.js@167:greetings>,
    <JS Function greetings.js@273:greetings>}

whole code:
pts(Person.greetings) = {<JS Function greetings.js@61:greetings>}
```

We can confirm that our implementation includes all 3 possible implementations of the greetings method of Person. This is consistent with what we obtained in the points-to sets for the variable result. If we were to check the possible types of the variable result based on the generated points-to sets for each analysis, we see that the variable result can only point to strings. If it was required for the variable result to only point at strings, then the conclusion would be the same for every analysis due to all implementations of the method greetings returning a string.

shapes.js

The second test program is shown in Snippet A.2. In this program, we have a variable TShape that has base implementations of the methods getType, area, perimeter, and numberOfSides. There are two contexts representing a type of shape and a trait defining the behavior of the methods for each context. For this program we show the computed points-to sets for the variable area in Line 63:

```
baseline:
pts(area) = {"Calling an abstract method area"}

ours:
pts(area) = {"calling an abstract method area", Number}

whole code:
pts(area) = {"Calling an abstract method area"}
```

Just like in the previous test program, the points-to sets for the baseline analysis and the whole code analysis do not include the possible implementations of the area method.

For our implementation, we start to see imprecisions due to the flow insensitivity of the analysis. Even though exactly one of the contexts Circle and Triangle is active when the area method is activated, the generated points-to set in our implementation still include the return of the base implementation of the method. Note that both our implementation and the whole code analysis have a Number element on their points-to sets, but not any particular number value. This is another imprecision, but it is due to the model for mathematical operations in WALA. The analysis is not able to compute those operations, so it assumes the result could be any number. In this instance, if we check the possible types of the variable area, we would conclude that it can only point to a string if we use the baseline analysis results or the whole code analysis results. With our implementation, the conclusion would be that the variable can point to a string or a number. In reality, the variable area can only point to a number, and if we wanted to check this with any of the three analyses, we would get a wrong conclusion. We still can use a control flow analysis to make sure any of the two context is active when the area method is activated. That way, we can ignore the string value in the points-to set for area and arrive to a correct conclusion. On the other hand, the points-to sets for the sides variable in Line 64 are:

```
baseline:
pts(sides) = {0}

ours:
pts(sides) = {0, 3, "Number of sides is not defined in a circle"}

whole code:
pts(sides) = {0}
```

As always, only our implementation can take into account all possible implementations. The most relevant part about these results is that the inferred possible types of the variable sides with our implementation are integer and string, while the inferred type using any of the two alternative is only integer. It is only natural for a variable storing the number of sides to be of type integer, but we see that it is not the case for this program. Checking the possible types of the variable sides with the baseline and the whole code analysis would give a wrong conclusion, and would not give any warnings if the variable is required to be an integer.

video-encoder.js

The code for the video encoder program can be found in Snippet A.3. This program uses the proceed function to work with multiple adaptations at once. There are two contexts representing an operation applied to an incoming message and two traits defining the implementations for the method send. The variable obj is adapted to both traits in a context each. If one of the encryption or compression contexts is activated, the send method applies an operation to the message sent by the previous implementation (with the proceed method), meaning that it is possible to activate multiple implementations of the same method at once, and the order of activation determines the behavior of the method. The points-to sets for the variable received_msg in Line 38 are the next ones:

```
baseline:
pts(received_msg) = {"message"}

ours:
pts(received_msg) = {Unknown}

whole code:
pts(received_msg) = {"message"}
```

All three analyses struggle to get a points-to set with good precision for the received_msg variable. The only element that the variable can point to is "<C><E>message<E><C>". Our implementation did analyze all possible implementation, but was not able to resolve the call to the proceed method. This time, the inferred type for the variable received_msg using the results of our implementation would be unknown.

course.js

The final test program is the one in A.4. This time, we have a function makeStudentTrait in Line 12 that creates traits dynamically depending on the marks parameter. There are traits representing the behavior of a student and a professor, and two contexts representing two courses. For the rop context, the object thomas is adapted with the trait defining a professor, and the rest of the variables are adapted with new student traits. Then, while the context rop is active, thomas can give marks to their students, modifying the array containing their marks. For this program, we take a look at the marks variable in Line 85:

```
baseline:
pts(marks) = {Unknown}

ours:
pts(marks) = {Array (instance 4)}

whole code:
pts(marks) = {Unknown}
```

The baseline analysis and the whole code analysis were not able to find the correct implementation for the method marks, so the results are unknown. For our implementation, the variable points to a single instance of an array. We will call this array i4. The points-to set for the first position of this array is:

```
ours:
pts(i4[0]) = {Object (instance 1),
    Object (instance 2),
    Object (instance 3)}
```

Even though the array should have only one element, and can only point to a single object instance, our implementation concludes that the first position of the array can point to three different instances. This imprecision can be explained by the context

sensitivity applied. The method giveMark is called 3 times, but a new element of the array is added at the same callsite, causing all three students to have all three created marks. Taking the first instance as an example (we will call it o1), we take a look at the points-to sets for each field:

```
ours:
pts(o1.course) = {"ROP", "SE"}
pts(o1.mark) = {1}
```

We see that the type of the course field is always a string, and the type of the mark field is always an integer. If those fields were require to be of a specific type at any point of execution, a type checker would conclude that this is the case using our analysis. This would not be possible with any of the two presented alternatives.

4.2.2 Performace evaluation

Table 4.1 shows the measurements made for all 4 test programs for the standard JavaScript WALA analysis, which will be our baseline, Table 4.2 shows the measurements made for our analysis implementation, and Table 4.3 show the measurements for the standard JavaScript WALA analysis including the code of the library. The experiments were performed in a PC with an AMD Ryzen 5 3500U with a base frequency of 2.10 GHz and a 8 GB 2400 MHz RAM.

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 activated. To illustrate this, see the code in Snippet A.1. The method greetings can have 3 possible implementations depending on which context is activated. Our implementation 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 implementation compared to the baseline due to the overhead of computing possible adaptations and activations, but the analysis time is lower, making the total time about the same for the baseline and our implementation. Finally, if we compare both the baseline and our implementation with the analysis of the whole code (analyzed file + context-traits library), we see that every metric is significantly higher for the whole code analyzer.

Table 4.1: Analysis results baseline

Analysis	Baseline			
Metrics	No. of nodes	No. of edges	Preproc. time [ms]	Analysis time [ms]
A.1	116	115	665 ± 25.4	398 ± 10.4
A.2	125	124	688 ± 10.7	391 ± 34.1
A.3	115	114	647 ± 20.4	372 ± 26.5
A.4	150	149	688 ± 13.6	436 ± 26.7

Table 4.2: Analysis results implementation

Analysis	Ours			
Metrics	No. of nodes	No. of edges	Preproc. time [ms]	Analysis time [ms]
A.1	120	119	727 ± 29.9	321 ± 25.3
A.2	131	130	791 ± 37.2	367 ± 17.4
A.3	119	118	751 ± 35.8	316 ± 18.4
A.4	159	158	857 ± 35.8	383 ± 12.6

Table 4.3: Analysis results whole code

Analysis	Whole code analysis			
Metrics	No. of nodes	No. of edges	Preproc. time [ms]	Analysis time [ms]
A.1	211	214	858 ± 25.5	538 ± 18.6
A.2	220	223	897 ± 28.6	567 ± 28
A.3	213	218	907 ± 39.9	602 ± 24.3
A.4	248	253	969 ± 53.8	602 ± 27

In general, we see that our implementation gives over-approximations to the points-to sets, but with better precision than the compared analyses and with little overhead. We also see how this improvement in precision can improve the results of type checking. Our implementation still has sources of imprecision due to not taking into account some of the capabilities of the context-traits library. We see that the implemented analysis can be useful for COP developers, as shown by the results of the type checking using each analysis. Our implementation produced better type checking results thanks to the improvement in the points-to sets precision, and we showed how this improvement can be useful to detect possible errors in the code.

4.3 Threads to validity

As the used implementation of COP for JavaScript [8] is quite big and complex, our implementation will not cover all the functionalities of the library. This means that the obtained results can only be generalized to context-oriented programs with a limited set of functionalities of the COP library. More specifically, we do not take into account the deactivation of a context, policies that determine what happens when multiple contexts are activated, or the proceed function. We evaluate how the lack of model for those functionalities can affect the precision for the analysis.

As we provide the programs that will be tested, there could be a bias on the precision obtained given our interest in obtaining good precision results, so it cannot be concluded with 100% certainty that our precision results can be generalized to all context-oriented programs. To mitigate this effect, we performed the experiments with a set of programs that cover multiple functionalities of the context traits library.

We could get to different conclusions when comparing the precision results of our implementation with existing ones if we use analyzers that would have considerably higher or lower precision than our analyzer would have for regular JavaScript programs. To

avoid that, all evaluated analyses were filed-sensitive, 1-CFA, and flow-insensitive.

It could be difficult to measure the precision of the points-to sets, and the average size of the points-to sets, that is generally used to measure precision, is not useful in this instance. This is because unsound analyses would produce smaller points-to sets in context-oriented programs while being imprecise. In general, an analysis with lower average points-to sets size is considered more precise than another with higher average size, but this is not true when the analyses are not sound. We will have prior knowledge about the ideal points-to sets based on our understanding of the context-oriented paradigm; that is why we will perform qualitative analysis when possible.

Chapter 5

Conclusion and Future Work

This thesis has described the problems that arise when analyzing context-oriented programs. We proposed a basic analysis that could handle certain operations in context-oriented programs using a specific JavaScript library. The evaluation of the implementation resulted in better precision results than possible alternatives.

5.1 Conclusion

The objective of this thesis was to improve the results of analyses that could use the points-to sets as an input by improving the points-to sets precision. When statically analyzing programs that make use of complex libraries, the obtained results can be quite imprecise even with a good context-sensitive flow-sensitive analysis, not to mention performance will be affected if precise results are needed. The imprecise results and unsoundness of said analyses can be a problem when using those results for other analyses such as type checking and bug finding. This thesis proposed a points-to analysis that is built on top of the standard field-sensitive WALA analysis for JavaScript, that statically model some of the context-oriented capabilities provided by the context-traits library.

The proposed analysis had two phases: the adaptation and activation finder, where the source code is analyzed in search for trait instantiations, adaptations, and context activations, and a code insertion part, that uses the information gathered by the adaptation and activation finder to rewrite certain nodes in the AST with the proposed model for each replaced instruction. The resulting AST will be the one analyzed by the standard JavaScript WALA analysis. The implemented analysis resulted in better precision results for basic programs, and with good performance in all cases. However, the precision results for more complex programs did not improve much due to the use of context-traits instructions that were omitted in the analysis. We showed that the improvement in precision can have benefits in type checking, improving results in some cases, but with poor results for more complex programs. In general, our analysis demonstrated to be better than current alternatives.

5.2 Future Work

Our research can be continued through the following directions.

5.2.1 Context-traits library coverage

Our analysis only takes into account a subset of the instructions available in the context-traits library, and the precision results can only be extrapolated to programs that only use that subset of instructions. To improve our analysis, it is necessary to statically model more instructions from the library. The challenge here is that the models have to allow the analysis below to get good precision results, so it is necessary that the chosen models are simple enough based on the context sensitivity and flow sensitivity used.

5.2.2 Flow sensitivity

The models used for our analysis are specifically designed for flow-insensitive analyses for the models to be simple. Allowing for flow sensitivity requires to change the way each instruction is modelled and to take into account the order of context activations.

5.2.3 Other libraries

Every COP library is different from the others. The way a library is implemented defines the behavior of its instructions. It is possible to follow the same approach for other COP implementations, and even for other languages. The challenges may vary depending on the library and the language.

5.2.4 Other approaches

Instead of replacing instructions with a simple model and analyzed the resulting AST with an existing analysis, another approach would be to implement an analysis from scratch, with context-oriented constructs included in the analysis.

Appendix A

Analyzed Context-Oriented Programs

```
1  var cop = require('context-traits');
2
3  Person = {
4    greetings: function() {
5      return "Hello!";
6    }
7  };
8
9  Spanish = new cop.Context();
10 SpanishSpeaking = Trait({
11   greetings: function() {
12     return "Hola!";
13   }
14 });
15
16 French = new cop.Context();
17 FrenchSpeaking = Trait({
18   greetings: function() {
19     return "Bonjour!";
20   }
21 });
22
23
24 Spanish.adapt(Person, SpanishSpeaking);
25 French.adapt(Person, FrenchSpeaking);
26
27 var result = Person.greetings();
28 Spanish.activate();
29 result = Person.greetings();
30 French.activate();
31 result = Person.greetings();
```

Snippet A.1: Test program greetings.js

```
var cop = require('context-traits');

TShape = {
  b: 3,
  h: 4,
  type: "shape",
  getType: function() {
    return this.type;
  },
  area: function() {
    return 'Calling an abstract method area';
  },
  perimeter: function() {
    return 'Calling an abstract method perimeter';
  },
  numberOfSides: function() {
      return 0;
  }
};

Triangle = new cop.Context();
TriangleBehavior = cop.Trait({
  getType: function() {
    return "triangle";
  },
  area: function() {
    return this.b * this.h/2;
  },
  perimeter: function() {
    return this.b + 2*Math.sqrt(Math.pow(this.h,2) + Math.pow(this.b
       /2, 2));
  },
  numberOfSides: function() {
      return 3;
  }
});
Triangle.adapt(TShape, TriangleBehavior);



Circle = new cop.Context();
CircleBehavior = cop.Trait({
  getType: function() {
    return "circle";
  },
  area: function() {
    return Math.pow(this.b, 2) * Math.PI;
  },
  perimeter: function() {
    return this.b * 2* Math.PI;
  },
  numberOfSides: function() {
      return "Number of sides is not defined in a circle";
  }
});
Circle.adapt(TShape, CircleBehavior);
```

```
56
57 if(Math.random() >= 0.5) {
58     Circle.activate();
59 } else {
60     Triangle.activate();
61 }
62
63 var area = TShape.area();
64 var sides = TShape.numberOfSides();
65
66 if(Circle.isActive()) {
67     Circle.deactivate();
68 } else {
69     Triangle.deactivate();
70 }
```

Snippet A.2: Test program shapes.js

```
1  var cop = require('context-traits');
2
3  CoreObject = Trait({
4    send: function(msg) {
5      return msg;
6    }
7  });
8
9  Encryption = new cop.Context({
10   name: "Encryption"
11 });
12
13
14 Compression = new cop.Context({
15   name: "Compression"
16 });
17
18 EncryptionRole = Trait({
19   send: function(msg) {
20     return "<E>" + this.proceed(msg) + "<E>";
21   }
22 });
23
24 CompressionRole = Trait({
25   send: function(msg) {
26     return "<C>" + this.proceed(msg) + "<C>";
27   }
28 });
29
30 obj = Object.create(Object.prototype, CoreObject);
31
32 Compression.adapt(obj, CompressionRole);
33 Encryption.adapt(obj, EncryptionRole);
34
35
36 Encryption.activate();
37 Compression.activate();
38 received_msg = obj.send("message");
39 Compression.deactivate();
```

```
40 Encryption.deactivate();
```

Snippet A.3: Test program video-encoder.js

```
1  var cop = require('context-traits');
2
3  //--- Base behavior definition
4  Person = Trait({
5    name: "",
6    role: function() {
7      console.log("base person method");
8    }
9  });
10
11 //--- Adaptations definition
12 function makeStudentTrait(marks) {
13    return Trait({
14      role: function() {
15        console.log("Student method");
16      },
17      setMarks: function(newMarks) {
18        marks = newMarks;
19      },
20      marks: function() {
21        return marks;
22      },
23      printMarks: function() {
24        console.log("Marks for the student " + this.name);
25        for(var i=0; i<marks.length; i++) {
26          console.log(marks[i].course + ": " + marks[i].mark);
27        }
28      }
29   });
30 }
31
32 ProfessorOps = Trait({
33   role: function() {
34     console.log("Professor method");
35   },
36   giveMark: function(mark, student) {
37     var newMarks = student.marks();
38     newMarks[newMarks.length] = {
39       "course": rop.isActive() ? "ROP" : "SE",
40       "mark": mark
41     };
42     console.log(newMarks);
43     student.setMarks(newMarks);
44   }
45 });
46
47 //context instances
48 rop = new cop.Context({
49   name: "rop"
50 });
51 se = new cop.Context({
52   name: "se"
53 });
```

```
54
55
56 //--- Object instances definition
57 var nicolas = Object.create(Object.prototype, Person);
58 nicolas.name = "Nicolas";
59 var thomas = Object.create(Object.prototype, Person);
60 thomas.name = "Thomas";
61 var ly = Object.create(Object.prototype, Person);
62 ly.name = "Ly";
63 var martin = Object.create(Object.prototype, Person);
64 martin.name = "Martin";
65 var markus = Object.create(Object.prototype, Person);
66 markus.name = "Markus";
67
68 //--- Context-object-behaviora adaptations association
69
70 rop.adapt(nicolas, makeStudentTrait([]));
71 rop.adapt(thomas, ProfessorOps);
72 rop.adapt(ly, makeStudentTrait([]));
73 rop.adapt(martin, makeStudentTrait([]));
74 rop.adapt(markus, makeStudentTrait([]));
75
76 se.adapt(nicolas, ProfessorOps);
77 se.adapt(thomas, makeStudentTrait([]));
78
79 //--- Main
80 rop.activate();
81 thomas.giveMark(1, nicolas);
82 thomas.giveMark(1, ly);
83 thomas.giveMark(1, markus);
84
85 var marks = markus.marks();
```

Snippet A.4: Test program course.js

Bibliography

[1] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Found. Trends Program. Lang.*, vol. 2, no. 1, p. 1–69, apr 2015. [Online]. Available: https://doi.org/10.1561/2500000014

[2] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. USA: USENIX Association, 2005, p. 18.

[3] S. Chandra and T. Reps, "Physical type checking for c," *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 5, p. 66–75, sep 1999. [Online]. Available: https://doi.org/10.1145/381788.316183

[4] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 1–12. [Online]. Available: https://doi.org/10.1145/1806596.1806598

[5] P. Costanza and O. Nierstrasz, "Context-oriented programming," *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, Mar. 2008. [Online]. Available: http://www.jot.fm/contents/issue_2008_03/article4.html

[6] R. Keays and A. Rakotonirainy, "Context-oriented programming," in *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, ser. MobiDe '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 9–16. [Online]. Available: https://doi.org/10.1145/940923.940926

[7] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara, "Contextj: Context-oriented programming with java," *Information and Media Technologies*, vol. 6, no. 2, pp. 399–419, 2011.

[8] S. González, K. Mens, M. Colacioiu, and W. Cazzola, "Context traits: Dynamic behaviour adaptation through run-time trait recomposition," in *Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development*, ser. AOSD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 209–220. [Online]. Available: https://doi.org/10.1145/2451436.2451461

[9] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Copenhage, 1994.

[10] B. Steensgaard, "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 32–41. [Online]. Available: https://doi.org/10.1145/237721.237727

[11] Y. Su, D. Ye, and J. Xue, "Parallel pointer analysis with cfl-reachability," *Proceedings of the International Conference on Parallel Processing*, vol. 2014, pp. 451–460, 11 2014.

[12] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "A principled approach to selective context sensitivity for pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 42, no. 2, may 2020. [Online]. Available: https://doi.org/10.1145/3381915

[13] S. Wei and B. G. Ryder, "Adaptive context-sensitive analysis for javascript," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. T. Boyland, Ed., vol. 37. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 712–734. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/5244

[14] S. Roy and Y. N. Srikant, "Partial flow sensitivity," in *High Performance Computing – HiPC 2007*, S. Aluru, M. Parashar, R. Badrinath, and V. K. Prasanna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 245–256.

[15] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 160–170. [Online]. Available: https://doi.org/10.1145/2854038.2854043

[16] S. Wei and B. Ryder, "State-sensitive points-to analysis for the dynamic behavior of javascript objects," 07 2014, pp. 1–26.

[17] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," in *ECOOP 2012 – Object-Oriented Programming*, J. Noble, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 435–458.

[18] R. Keays and A. Rakotonirainy, "Context-oriented programming," in *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, ser. MobiDe '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 9–16. [Online]. Available: https://doi.org/10.1145/940923.940926

[19] R. Hirschfeld, P. Costanza, and O. M. Nierstrasz, "Context-oriented programming," *Journal of Object technology*, vol. 7, no. 3, pp. 125–151, 2008.

[20] G. Salvaneschi, C. Ghezzi, and M. Pradella, "Context-oriented programming: A software engineering perspective," *Journal of Systems and Software*, vol. 85, no. 8, pp. 1801–1817, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412121200074X

[21] R. Hirschfeld, P. Costanza, and M. Haupt, "An introduction to context-oriented programming with contexts," in *International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 2007, pp. 396–407.

[22] A. Elyasaf and A. Sturm, "Towards a framework for analyzing context-oriented programming languages," in *Proceedings of the 13th ACM International Workshop on Context-Oriented Programming and Advanced Modularity*, ser. COP 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 16–23. [Online]. Available: https://doi.org/10.1145/3464970.3468414

[23] "T.j. watson libraries for analysis (wala)." [Online]. Available: http://wala.sourceforge.net