Type-Based Call Graph Construction Algorithms for Scala

KARIM ALI, Technische Universität Darmstadt
MARIANNA RAPOPORT and ONDŘEJ LHOTÁK, University of Waterloo
JULIAN DOLBY, IBM T.J. Watson Research Center
FRANK TIP, Samsung Research America

Call graphs have many applications in software engineering. For example, they serve as the basis for code navigation features in integrated development environments and are at the foundation of static analyses performed in verification tools. While many call graph construction algorithms have been presented in the literature, we are not aware of any that handle Scala features such as traits and abstract type members. Applying existing algorithms to the JVM bytecodes generated by the Scala compiler produces very imprecise results because type information is lost during compilation. We adapt existing type-based call graph construction algorithms to Scala and present a formalization based on Featherweight Scala. An experimental evaluation shows that our most precise algorithm generates call graphs with 1.1–3.7 times fewer nodes and 1.5–17.3 times fewer edges than a bytecode-based RTA analysis.

CCS Concepts: • **Software and its engineering → Automated static analysis; Object oriented languages;**

Additional Key Words and Phrases: Call graphs, static analysis, Scala

ACM Reference Format:
Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. 2015. Type-based call graph construction algorithms for Scala. ACM Trans. Softw. Eng. Methodol. 25, 1, Article 9 (November 2015), 43 pages.
DOI: http://dx.doi.org/10.1145/2824234

1. INTRODUCTION

As Scala [Odersky et al. 2012] gains popularity, the need grows for program analysis tools for it that automate tasks such as refactoring, bug-finding, verification, security analysis, and whole-program optimization. Such tools typically need call graphs to approximate the behavior of method calls. Call graph construction has been studied extensively [Ryder 1979; Grove and Chambers 2001], and algorithms vary primarily in how they handle indirect function calls. Several Scala features—such as traits, abstract type members, and closures—affect method call behavior. However, to the best of our knowledge, our research is the first to propose and evaluate call graph construction algorithms for Scala.

In principle, one could construct call graphs of Scala programs by compiling them to JVM bytecode and then applying existing bytecode-based program analysis

This research was supported by the Natural Sciences and Engineering Research Council of Canada and by the Ontario Ministry of Research and Innovation.
Authors' addresses: K. Ali (corresponding author), Secure Software Engineering Group, TU Darmstadt, Rheinstr. 75, 64295 Darmstadt, Germany; email: karim.ali@cased.de; M. Rapoport and O. Lhoták, 200 University Avenue West, Waterloo, Ontario N2L 3G1, Canada; emails: {mrapoport, olhotak}@uwaterloo.ca; J. Dolby, P.O. Box 704, Yorktown Heights, NY 10598; email: dolby@us.ibm.com; F. Tip, 1732 N. First Street, San Jose, CA 95112; email: ftip@samsung.com.

DOI: http://dx.doi.org/10.1145/2824234

frameworks such as WALA [IBM 2013] or SOOT [Vallée-Rai et al. 2000] to those generated bytecodes. However, as we shall demonstrate, this approach is not viable because significant type information is lost during the compilation of Scala programs, causing the resulting call graphs to become extremely imprecise. Furthermore, the Scala compiler translates certain language features using hard-to-analyze reflection. While solutions exist for analyzing programs that use reflection, such approaches tend to be computationally expensive or they make very conservative assumptions that result in a loss of precision.

Therefore, we explore how to adapt existing call graph construction algorithms for Scala, and we evaluate the effectiveness of such algorithms in practice. Our focus is on adapting low-cost algorithms to Scala, in particular Name-Based Resolution (RA) [Srivastava 1992], Class Hierarchy Analysis (CHA) [Dean et al. 1995], and Rapid Type Analysis (RTA) [Bacon and Sweeney 1996]. We consider how key Scala features—such as traits, abstract type members, and closures—can be accommodated and present a family of successively more precise Trait Composition Analysis (TCA) algorithms. We formally define our most precise algorithm for FS_{alg}, the "Featherweight Scala" subset of Scala previously defined by Cremet et al. [2006], and prove its correctness by demonstrating that for each execution of a method call in the operational semantics, a corresponding edge exists in the constructed call graph.

Our new algorithms differ primarily in how they handle the two key challenges of analyzing Scala: *traits*, which encapsulate a group of method and field definitions so that they can be mixed into classes, and *abstract type members*, which provide a flexible mechanism for declaring abstract types that are bound during trait composition. We implement our algorithms in the Scala compiler and compare the number of nodes and edges in call graphs computed for a collection of publicly available Scala programs. In addition, we evaluate the effectiveness of applying the RTA algorithm to the JVM bytecodes generated by the Scala compiler. For each comparison, we investigate which Scala programming idioms result in differences in cost and precision of the algorithms.

Our experimental results indicate that careful handling of complex Scala features greatly improves call graph precision. We also find that call graphs constructed from the JVM bytecodes using the RTA algorithm are much less precise than those constructed using our source-based algorithms, because significant type information is lost due to the transformations and optimizations performed by the Scala compiler.

In summary, this article makes the following contributions.

(1) We present variations on the RA [Srivastava 1992] and RTA [Bacon and Sweeney 1996] algorithms for Scala. To our knowledge, these are the first call graph construction algorithms designed for Scala.
(2) We evaluate these algorithms, comparing their relative cost and precision on a set of publicly available Scala programs.
(3) We evaluate the application of the RTA algorithm to the JVM bytecodes produced by the Scala compiler and show that such an approach is not viable because it produces highly imprecise call graphs.
(4) We formalize our most precise algorithm and prove its correctness.

A previous version of this article appeared in *Proceedings of ECOOP* [Ali et al. 2014]. This article covers a complete formalization of the main algorithm along with detailed proofs, reports on additional experiments, and additional subject programs in experiments reported on previously.

The remainder of this article is organized as follows. Section 2 reviews existing call graph construction algorithms that serve as the inspiration for our work. Section 3 presents a number of motivating examples that illustrate the challenges associated

with constructing call graphs of Scala programs. Section 4 presents our algorithms. Section 5 presents a formalization and correctness proof using the Featherweight Scala (FS_{alg}) formalism. Section 6 presents the implementation in the context of the Scala compiler. An evaluation of our algorithms is presented in Section 7. Lastly, Section 8 concludes and briefly discusses directions for future work.

2. BACKGROUND

Algorithms for call graph construction [Grove and Chambers 2001] have been studied extensively in the context of object-oriented programming languages such as Java [DeFouw et al. 1998; Lhoták and Hendren 2003], C++ [Bacon and Sweeney 1996], and Self [Agesen 1994], of functional programming languages such as Scheme [Shivers 1991] and ML [Heintze 1994], and of scripting languages such as JavaScript [Sridharan et al. 2012]. Roughly speaking, most call graph construction algorithms can be classified as being either *type-based* or *flow-based* [Lhoták and Hendren 2003; Bravenboer and Smaragdakis 2009; Heintze and Tardieu 2001; Lhoták and Hendren 2006; Henglein 1992]. The former class of algorithms uses only local information given by static types to determine possible call targets, whereas the latter analyzes the program's dataflow.

2.1. Type-Based Algorithms

In this section, we briefly review some important type-based call graph construction algorithms for object-oriented languages upon which our work is based. In the exposition of these algorithms, we use a constraint notation that is equivalent to that of Tip and Palsberg [2000] but that explicitly represents call graph edges using a relation '$\mapsto$' between call sites and methods.

Name-Based Resolution (RA). The main challenge in constructing call graphs of object-oriented programs is in approximating the behavior of dynamically dispatched (virtual) method calls. Early work (see, e.g., [Srivastava 1992]) simply assumed that a virtual call $e.m(\cdots)$ can invoke any method with the same name m. This approach can be captured using the following constraints:

$$\frac{}{main \in R}\ \text{RA}_{\text{MAIN}} \qquad \frac{c \mapsto M}{M \in R}\ \text{RA}_{\text{REACHABLE}} \qquad \frac{\begin{array}{c}\text{call } c : e.m(\ldots) \text{ occurs in method } M \\ \text{method } M' \text{ has name } m \\ M \in R\end{array}}{c \mapsto M'}\ \text{RA}_{\text{CALL}}.$$

Intuitively, rule RA_{MAIN} reads "the main method is reachable" by including it in the set R of reachable methods. Rule RA_{CALL} states that "if a method is reachable, and a call site $c : e.m(\ldots)$ occurs in its body, then every method M' with name m is reachable from c." Finally, rule $\text{RA}_{\text{REACHABLE}}$ states that any method M reachable from a call site c is contained in the set R of reachable methods.

Class Hierarchy Analysis (CHA). Obviously, Name-Based Resolution can become very imprecise if a class hierarchy contains unrelated methods that happen to have the same name. Class Hierarchy Analysis [Dean et al. 1995] improves upon name-based resolution by using the static type of the receiver expression of a method call in combination with class hierarchy information to determine what methods may be invoked from a call site. Following the notation of Tip and Palsberg [2000], we use $StaticType(e)$ to denote the static type of an expression e, and $StaticLookup(C, m)$ to denote the method definition that is invoked when method m is invoked on an object

with runtime type C. Using these definitions, CHA is defined as follows:

$$\frac{}{main \in R}\ \text{CHA}_{\text{MAIN}} \qquad \frac{\text{call } c : e.m(\ldots) \text{ occurs in method } M \quad C \in SubTypes(StaticType(e)) \quad StaticLookup(C, m) = M' \quad M \in R}{c \mapsto M'}\ \text{CHA}_{\text{CALL}}.$$

$$\frac{c \mapsto M}{M \in R}\ \text{CHA}_{\text{REACHABLE}}$$

Rules CHA_{MAIN} and $\text{CHA}_{\text{REACHABLE}}$ are the same as their counterparts for RA. Intuitively, rule CHA_{CALL} now reads: "If a method is reachable, and a call site $c : e.m(\ldots)$ occurs in the body of that method, then every method M' with name m that is inherited by a subtype of the static type of e is reachable from c."

Rapid Type Analysis (RTA). Bacon and Sweeney [1996] and Bacon [1997] observe that CHA produces very imprecise results when only a subset of the classes in an application is instantiated. In such cases, CHA loses precision because, effectively, it assumes for a method call $e.m(\cdots)$ that all subtypes of the static type of e may arise at runtime. In order to mitigate this loss of precision, RTA maintains a set of types $\hat{\Sigma}$ that have been instantiated in reachable methods. This set is used to approximate the types that a receiver expression may assume at runtime. The constraint formulation of RTA is as follows:

$$\frac{}{main \in R}\ \text{RTA}_{\text{MAIN}} \qquad \frac{\text{call } e.m(\ldots) \text{ occurs in method } M \quad C \in SubTypes(StaticType(e)) \quad StaticLookup(C, m) = M' \quad M \in R \quad C \in \hat{\Sigma}}{c \mapsto M'}\ \text{RTA}_{\text{CALL}}.$$

$$\frac{\text{“new } C()\text{” occurs in } M \quad M \in R}{C \in \hat{\Sigma}}\ \text{RTA}_{\text{NEW}} \qquad \frac{c \mapsto M}{M \in R}\ \text{RTA}_{\text{REACHABLE}}$$

Rules RTA_{MAIN} and $\text{RTA}_{\text{REACHABLE}}$ are again the same as before. Intuitively, RTA_{CALL} refines CHA_{CALL} by requiring that $C \in \hat{\Sigma}$, and rule RTA_{NEW} reads: "$\hat{\Sigma}$ contains the classes that are instantiated in a reachable method."

Sallenave and Ducourneau [2012] recently present an extension of RTA for the C# language that determines the types with which parameterized classes are instantiated by maintaining sets of type tuples for parameterized classes and methods. They use their analysis to generate efficient CLI code for embedded applications that avoids expensive boxing/unboxing operations on primitive types, while permitting a space-efficient shared representation for reference types.

2.2. Flow-Based Algorithms

The goal of flow-based program analysis is to statically overapproximate the set of all values that a program variable or expression could have at runtime by tracing the possible flow of data or objects through a program. In particular, for a method invocation $e.m(\ldots)$, such an analysis can predict all possible runtime values of e and (by looking up method m on each of these) all possible call targets. Similarly, for a higher-order function call, the analysis can determine all possible functions that may be invoked.

In *propagation-based* analyses, this is achieved by associating sets of abstract values (each representing a set of runtime values) with program expressions, modeling runtime dataflow as inclusion constraints between these sets, and solving the resulting

constraint system. If each expression is associated with a single set of abstract values, the analysis is *monovariant*, whereas *polyvariant* analyses use multiple sets to capture the possible values of an expression in different contexts to improve precision.

Flow-based analyses typically also consider static type information when it is available, but they can also function without static types. They are therefore particularly suited for dynamically typed languages: propagation-based approaches have been used to construct call graphs for Scheme [Shivers 1991], Self [Agesen 1994], and JavaScript [Sridharan et al. 2012]. But of course, they can also be applied in the context of statically typed languages like ML [Heintze 1994] or Java [DeFouw et al. 1998; Lhoták and Hendren 2003].

Propagation-based call graph construction algorithms are more precise and more general than type-based ones, but their effectiveness seems to be very language-dependent. State-of-the-art monomorphic flow analyses [Lhoták and Hendren 2003; Bravenboer and Smaragdakis 2009; Heintze and Tardieu 2001] are able to build call graphs for real-world Java or C programs within minutes; polymorphic flow analyses do not yet scale as well, but at least for Java they do not seem to improve call graph precision by much [Lhoták and Hendren 2006].

For Java, an important remaining obstacle is the fact that most propagation-based analyses require that the entire program is available and hence have to analyze the massive standard libraries. Recent work on application-only call graph construction [Ali and Lhoták 2012, 2013] has shown how to generate a call graph for the application part of a program without analyzing library code.

The scalability of propagation-based call graph construction algorithms can also be improved by coarsening the propagation: instead of computing per-expression information, propagation can be done per-variable [Sundaresan et al. 2000] or per-method [Tip and Palsberg 2000].

A cheaper alternative to propagation-based approaches is *unification-based* analyses, where dataflow edges are undirected [Henglein 1992], yielding less precise analysis results.

3. MOTIVATING EXAMPLES

Before presenting our algorithms in Section 4, we briefly review the Scala features that pose the most significant challenges for call graph construction.

3.1. Traits

Traits are one of the cornerstone features of Scala. They provide a flexible mechanism for distributing the functionality of an object over multiple reusable components. Traits are similar to Java's abstract classes in the sense that they may provide definitions of methods and in that they cannot be instantiated by themselves. However, they resemble Java interfaces in the sense that a class or trait may extend (i.e., "mix-in") multiple super-traits.

Figure 1 shows an example program that declares a trait A in which a concrete method foo and an abstract method bar are defined. The program also declares a trait B that defines a concrete method bar and an abstract method foo. Lastly, trait C defines a concrete method foo. The program contains a main method that creates an object by composing A and B, and then calls bar on that object. The allocation expression new *A* with *B* is equivalent to a declaration and instantiation of a new empty class with parents *A* with *B*.

Before turning our attention to call graph construction, we need to consider how method calls are resolved in Scala. In Scala, the behavior of method calls depends on the class linearization order of the receiver object [Odersky 2011, Section 5.1.2]. The

```
1  object Traits {
2    trait A {
3      def foo = println("A.foo")
4      def bar
5    }
6    trait B {
7      def foo
8      def bar = this.foo
9    }
10   trait C {
11     def foo = println("C.foo")
12   }
13
14   def main(args: Array[String]) =
15     { (new A with B).bar }
16 }
```

Fig. 1. A Scala program illustrating the use of traits.

linearization of a class C with parents C_1 with $\cdots$ with C_n is defined as

$$\mathcal{L}(C) = C, \mathcal{L}(C_n) \vec{+} \cdots \vec{+} \mathcal{L}(C_1)$$

where $\vec{+}$ denotes concatenation where elements of the right operand replace identical elements of the left one. Scala defines the set of *members* of a class in terms of its linearization. The concrete members of a class C are all concrete members m declared in all classes D in $\mathcal{L}(C)$, except for those overridden by a matching concrete member m' declared in some class E that precedes D in the linearization order.[1] Given this notion of class membership, the resolution of method calls is straightforward: a call $x.m(\cdots)$, where x has type C at runtime, dispatches to the unique concrete member matching m in C.

For the example of Figure 1, the linearization order of type new A with B on line 15 is as follows: X, B, A (here, we use X to denote the anonymous class that is implicitly declared by the allocation expression new A with B). Following the preceding definitions, the set of concrete members of X is { B.bar, A.foo }. Hence, the call to bar on line 15 resolves to B.bar. Using a similar argument, the call to foo on line 8 resolves to A.foo. Therefore, executing the program will print "`A.foo`".

Implications for Call Graph Construction. The presence of traits complicates the construction of call graphs because method calls that occur in a trait typically cannot be resolved by consulting the class hierarchy alone. In the example of Figure 1, B.bar contains a call this.foo on line 8. How should a call graph construction algorithm approximate the behavior of this call, given that there is no inheritance relation between A, B, and C?

To reason about the behavior of method calls in traits, a call graph construction algorithm needs to make certain assumptions about how traits are combined. One very conservative approach would be to assume that a program may combine each trait with any set of other traits in the program in any order,[2] such that the resulting combination is syntactically correct.[3] Then, for each of these combinations, one could compute the members contained in the resulting type and approximate the behavior of calls by determining the method that is selected in each case. For the program of Figure 1, this approach would assume that B is composed with either A or with C. In the former case, the call on line 8 is assumed to be invoked on an object of type A with B (or B with A) and would dispatch to A.foo. In the latter, the call is assumed to be invoked

[1]Roughly speaking, two members are considered as matching when they have the same name and signature. The matching relation is defined fully in Odersky [2011, Section 5.1.3].

[2]Note that an X with Y object may behave differently from a Y with X object in certain situations because these objects have different linearization orders.

[3]If multiple traits that provide concrete definitions of the same method are composed, all but the last of these definitions in the linearization order must have the override modifier in order for the composition to be syntactically correct [Odersky 2011, Section 5.1.4].

```
17 object AbstractTypeMembers {
18   trait HasFoo {
19     def foo: Unit
20   }
21   trait X {
22     class A extends HasFoo {
23       def foo = println("X.A.foo")
24     }
25   }
26   trait Y {
27     class A extends HasFoo {
28       def foo = println("Y.A.foo")
29     }
30     type B = A
31     val o = new A
32   }
33   trait Z {
34     type B <: HasFoo
35     val o: B
36     def bar = o.foo
37   }
38
39   def main(args: Array[String]) =
40     { (new Y with Z {}).bar }
41 }
```

Fig. 2. A Scala program illustrating the use of abstract type members.

on an object of type C with B (or B with C) and would dispatch to C.foo. Hence, a call graph would result in which both A.foo and C.foo are reachable from the call on line 8.

The conservative approach discussed just is likely to be imprecise and inefficient in cases where a program contains many traits that can be composed with each other. For practical purposes, a better approach is to determine the set of combinations of traits that actually occur in the program and to use that set of combinations of traits to resolve method calls. Returning to our example program, we observe that the only combination of traits that occurs in the program is A with B, on line 15. If the call on line 8 is dispatched on an object of this type, it will dispatch to A.foo, as previously discussed. Hence, this approach would create a smaller call graph in which there would be only one outgoing edge for the call on line 8.

This more precise approach requires that the set of all combinations of traits in the program can be determined. The conservative approach could still have merit in cases where such information is not available (e.g., libraries intended to be extended with code that instantiates additional trait combinations).

3.2. Abstract Type Members

Scala supports a flexible mechanism for declaring *abstract type members* in traits and classes. A *type declaration* [Odersky 2011, §4.3] defines a name for an abstract type, along with upper and lower bounds that impose constraints on the concrete types that it could be bound to. An abstract type is bound to a concrete type when its declaring trait is composed with (or extended by) another trait that provides a concrete definition in one of two ways: either it contains a class or trait with the same name as the abstract type, or it declares a *type alias* [Odersky 2011, §4.3] that explicitly binds the abstract type to some specified concrete type.

Figure 2 shows a program that declares traits X, Y, Z, and HasFoo. Traits X and Y each declare a member class A that is a subclass of HasFoo. Trait Z declares an abstract type member B and an abstract field o. Trait Y declares a type alias that instantiates the type member B to A. It also instantiates the field o with the value new A. Observe that the abstract member type B of Z has a bound HasFoo and that o is declared to be of type B. The presence of this bound means that we can call foo on o on line 36.

On line 40, the program creates an object by composing Y with Z and calls bar on it. Following Scala's semantics for method calls, this call will dispatch to Z.bar. To understand how the call o.foo on line 36 is resolved, we must understand how abstract type members are bound to concrete types as a result of trait composition. In this case,

```
42 object Closures {
43   def bar1(y: () => A) = { y() }
44   def bar2(z: () => B) = { z() }
45
46   class A
47   class B
48   def main(args: Array[String]) = {
49     val foo1 = () => { new A }
50     val foo2 = () => { new B }
51     this.bar1(foo1)
52     this.bar2(foo2)
53   }
54 }
```

Fig. 3. A Scala program illustrating the use of closures.

the composition of Y with Z means that the types Y.B and Z.B are unified. Since Y.B is defined to be the same as Y.A, it follows that the abstract type member Z.B is bound to the concrete type Y.A. Thus, executing the call on line 36 dispatches to Y.A.foo, so the program prints "`Y.A.foo`".

Implications for Call Graph Construction. How could a call graph construction algorithm approximate the behavior of calls such as o.foo in Figure 2, where the receiver expression's type is abstract? A conservative solution relies on the bound of the abstract type as follows: For a call $o.f(\cdots)$ where o is of an abstract type T with bound B, one could assume the call to dispatch to definitions of $f(\cdots)$ in any subtype of B. This approach is implemented in our TCA^{bounds} algorithm and identifies both X.A.foo and Y.A.foo as possible targets of the call on line 36.

However, this approach may be imprecise if certain subtypes of the bound are not instantiated. Our TCA^{expand} algorithm implements a more precise approach that considers how abstract type members are bound to concrete types in observed combinations of traits, in the same spirit of the more precise treatment of trait composition discussed above. In Figure 2, the program only creates an object of type Y with Z, and Z.B is bound to Y.A in this particular combination of traits. Therefore, the call on line 36 must dispatch to Y.A.foo.

Scala's parameterized types [Odersky 2011, §3.2.4] resemble abstract type members and are handled similarly. Similar issues arise in other languages with generics [Sallenave and Ducourneau 2012].

3.3. Closures

Scala allows functions to be bound to variables and passed as arguments to other functions. Figure 3 illustrates this feature, commonly known as "closures." On line 49, the program creates a function and assigns it to a variable foo1. The function's declared type is () => A, indicating that it takes no parameters and returns an object of type A. Likewise, line 50 assigns to foo2 a function that takes no arguments and returns a B object.

Next, on line 51, bar1 is called with foo1 as an argument. Method bar1 (line 43) binds this closure to its parameter y, which has declared type () => A, and then calls the function bound to y. Similarly, on line 52, bar2 is called with foo2 as an argument. On line 44, this closure is bound to a parameter z and then invoked. From the simple dataflow in this example, it is easy to see that the call y() on line 43 always calls the function that was bound to foo1 on line 49, and that the call z() on line 44 always calls the function that was bound to foo2 on line 50.

Implications for Call Graph Construction. In principle, one could use the declared types of function-valued expressions and the types of closures that have been created to determine if a given call site could invoke a given function. For example, the type of y is () => A, and line 51 creates a closure that can be bound to a variable of this type. Therefore, a call graph edge needs to be constructed from the call site y() to the closure

```
55 object Closures {
56   def bar1(y: () => A) = { y.apply() }
57   def bar2(z: () => B) = { z.apply() }
58
59   class A
60   class B
61
62   def main(args: Array[String]) = {
63     val foo1: () => A = {
64       class $anonfun extends scala.runtime.AbstractFunction0[A] {
65         def apply(): A = { new A() }
66       };
67       new $anonfun()
68     };
69     val foo2: () => B = {
70       class $anonfun extends scala.runtime.AbstractFunction0[B] {
71         def apply(): B = { new B() }
72       };
73       new $anonfun()
74     };
75     this.bar1(foo1)
76     this.bar2(foo2)
77   }
78 }
```

Fig. 4. Desugared version of the program of Figure 3 (slightly simplified).

```
79 object This {
80   trait A {
81     def foo
82     // can only call B.foo
83     def bar = this.foo
84   }
85
86   class B extends A {
87     def foo = println("B.foo")
88   }
89   class C extends A {
90     def foo = println("C.foo")
91     override def bar = println("C.bar")
92   }
93
94   def main(args: Array[String]) = {
95     (new B).bar
96     (new C).bar
97   }
98 }
```

Fig. 5. A Scala program illustrating a call on this.

on line 51. By the same reasoning, a call graph edge should be constructed from the call site z() to the closure on line 52.

Our implementation takes a different approach to handling closures. Rather than performing the analysis at the source level, we apply it after the Scala compiler has "desugared" the code by transforming closures into anonymous classes that extend the appropriate scala.runtime.AbstractFunction*N*. Each such class has an apply() method containing the closure's original code. Figure 4 shows a desugared version of the program of Figure 3. After this transformation, closures can be treated as ordinary parameterized Scala classes without loss of precision. This has the advantage of keeping our implementation simple and uniform.

3.4. Calls on the Variable this

Figure 5 shows a program that declares a trait A with subclasses B and C. Trait A declares an abstract method foo, which is overridden in B and C, and a concrete method bar, which is overridden in C (but not in B). The program declares a main method that calls bar on objects of type B and C (lines 95–96). Executing the call to bar on line 95

dispatches to A.bar(). Executing the call this.foo() in that method will then dispatch to B.foo(). Finally, executing the call to bar on line 96 dispatches to C.bar, so the program prints "B.foo", then "C.bar."

Consider how a call graph construction algorithm would approximate the behavior of the call this.foo() at line 83. The receiver expression's type is A, so CHA concludes that either B.foo or C.foo could be invoked, since B and C are subtypes of A. However, note that this cannot have type C in A.bar because C provides an overriding definition of bar. Stated informally, this cannot have type C inside A.bar because then execution would not have arrived in A.bar in the first place. The TCA$^{expand\text{-}this}$ algorithm, presented in Section 4, exploits such knowledge. Care must be taken in the presence of super-calls, as we will discuss.

3.5. Bytecode-Based Analysis

The preceding examples show that Scala's traits and abstract type members pose new challenges for call graph construction. Several other Scala features, such as path-dependent types and structural types, introduce further complications and will be discussed in Section 6. At this point, the reader may wonder if all these complications could be avoided by simply analyzing the JVM bytecodes produced by the Scala compiler.

We experimentally determine that such an approach is not viable for two reasons. First, the translation of Scala source code to JVM bytecode involves significant code transformations that result in the loss of type information, causing the computed call graphs to become imprecise. Second, the Scala compiler generates code containing hard-to-analyze reflection for certain Scala idioms.

Loss of Precision. Consider Figure 6, which shows JVM bytecode produced by the Scala compiler for the program of Figure 3. As can be seen in the figure, the closures that are defined on lines 49 and 50 in Figure 3 have been translated into classes Closures$$anonfun$1 (lines 121–131 in Figure 6) and Closures$$anonfun$2 (lines 133–143). These classes extend scala.runtime.AbstractFunction0<T>, which is used for representing closures with no parameters at the bytecode level. Additionally, these classes provide overriding definitions for the apply method inherited by scala.runtime.AbstractFunction0<T> from its super-class scala.Function0<T>. This apply method returns an object of type T. The issue to note here is that Closures$$anonfun$1 and Closures$$anonfun$2 each instantiate the type parameter T with different types, Closures$A and Closures$B, respectively. Therefore, their apply methods return objects of type Closures$A and Closures$B. However, at the bytecode level, all type parameters are erased so that we have the following situation.

—scala.Function0.apply has return type Object.
—Closures$$anonfun$1.apply and Closures$$anonfun$2.apply each override scala.Function0.apply and also have return type Object.
—There are two calls to scala.Function0.apply on lines 111 and 116.

Given this situation, the RTA algorithm creates edges to Closures$$anonfun$1.apply and Closures$$anonfun$2.apply from each of the calls on lines 111 and 116. In other words, a bytecode-based RTA analysis creates four call graph edges for the closure-related calls, whereas the analysis of Section 3.3 only creates two edges. In Section 7, we show that this scenario commonly arises in practice, causing bytecode-based call graphs to become extremely imprecise.

Reflection in Generated Code. We detected several cases where the Scala compiler generates code that invokes methods using java.lang.reflect.Method.invoke(). In particular, the Scala compiler uses reflection to call methods of structural types. Reflection is

```
 99 public final class Closures$ {
100   public void main(java.lang.String []);
101       0: new Closures$$anonfun$1
102          ...
103       8: new Closures$$anonfun$2
104          ...
105      18: invokevirtual Closures$.bar1(scala.Function0) : void
106          ...
107      23: invokevirtual Closures$.bar2(scala.Function0) : void
108      26: return
109   public void bar1(scala.Function0);
110       0: aload_1
111       1: invokeinterface scala.Function0.apply() : java.lang.Object
112       6: pop
113       7: return
114   public void bar2(scala.Function0);
115       0: aload_1
116       1: invokeinterface scala.Function0.apply() : java.lang.Object
117       6: pop
118       7: return
119 }
120
121 public final class Closures$$anonfun$1 extends scala.runtime.AbstractFunction0 {
122     public final Closures$A apply();
123       0: new Closures$A
124       3: dup
125       4: invokespecial Closures$A()
126       7: areturn
127   public final java.lang.Object apply();
128       0: aload_0
129       1: invokevirtual Closures$$anonfun$1.apply() : Closures$A
130       4: areturn
131 }
132
133 public final class Closures$$anonfun$2 extends scala.runtime.AbstractFunction0 {
134     public final Closures$B apply();
135       0: new Closures$B
136       3: dup
137       4: invokespecial Closures$B()
138       7: areturn
139   public final java.lang.Object apply();
140       0: aload_0
141       1: invokevirtual Closures$$anonfun$2.apply() : Closures$B
142       4: areturn
143 }
```

Fig. 6. JVM bytecode produced by the Scala compiler for the program of Figure 3.

necessary so that existing Java objects (whose code the Scala compiler does not change) can be considered instances of structural types [Dubochet and Odersky 2009]. Structural types are common in Scala programs because the Scala compiler infers a structural type whenever a new expression instantiates an anonymous class.

One such example (taken from the ENSIME program, see Section 7) is shown in Figure 7. Figure 8 shows some of the relevant fragments of the JVM bytecodes produced for the program of Figure 7. Here, the reader may observe that reflective calls to Class.getMethod() and Method.invoke() appear on lines 163, 174 and 182, 187, respectively.

```
144 trait ClassHandler
145
146 object LuceneIndex {
147   def buildStaticIndex (): Int = {
148     val handler = new ClassHandler {
149       var classCount = 0
150       var methodCount = 0
151     }
152     handler.classCount + handler.methodCount
153   }
154 }
```

Fig. 7. A Scala program for which the compiler generates code containing reflective method calls (taken from the *ensime* program, see Section 7).

```
155 public final class LuceneIndex$ {
156   public static java.lang.reflect.Method reflMethod$Method1(java.lang.Class);
157     0: getstatic reflPoly$Cache1
158     3: invokevirtual java.lang.ref.SoftReference.get() : java.lang.Object
159     ...
160    49: ldc "classCount"
161    51: getstatic reflParams$Cache1
162    54: invokevirtual java.lang.Class.getMethod(java.lang.String,java.lang.Class[])
163                                                          : java.lang.reflect.Method;
164     ...
165    78: areturn
166
167   public static java.lang.reflect.Method reflMethod$Method2(java.lang.Class);
168     0: getstatic reflPoly$Cache2
169     3: invokevirtual java.lang.ref.SoftReference.get() : java.lang.Object
170     ...
171    49: ldc "methodCount"
172    51: getstatic reflParams$Cache2
173    54: invokevirtual java.lang.Class.getMethod(java.lang.String,java.lang.Class[])
174                                                          : java.lang.reflect.Method;
175     ...
176    78: areturn
177
178   public void buildStaticIndex ();
179   181: invokestatic reflMethod$Method1
180     ...
181   190: invokevirtual java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
182                                                          : java.lang.Object
183     ...
184   214: invokestatic reflMethod$Method2
185     ...
186   223: invokevirtual java.lang.reflect.Method.invoke(java.lang.Object,java.lang.Object[])
187                                                          : java.lang.Object
188     ...
189   260: athrow
190 }
```

Fig. 8. JVM bytecode produced by the Scala compiler for the program of Figure 7.

In general, the use of reflection creates significant problems for static analysis because it must either make very conservative assumptions that have a detrimental effect on precision (e.g., assuming that calls to java.lang.reflect.Method.invoke() may invoke any method in the application) or the analysis will become unsound.

```
191 class A { def foo = "A.foo" }
192 class B extends A { override def foo = "B.foo" }
193 class C { def foo = "C.foo" }
194 class D { def foo = "D.foo" }
195 class CallSiteClass[T <: A](val receiver: T) {
196   def callsite = {
197     /* resolves to:
198      *   TCA^expand: { B.foo }, TCA^bounds: { B.foo, A.foo }
199      *   TCA^names: { B.foo, A.foo, C.foo } , RA: { B.foo, A.foo, C.foo, D.foo }
200      */
201     receiver.foo
202   }
203 }
204 def main(args: Array[String]): Unit = {
205   new A
206   val receiver = new B
207   new C
208   val callSiteClass = new CallSiteClass[B](receiver);
209   callSiteClass.callsite
210 }
```

Fig. 9. A Scala program illustrating the varying precision of the analyses.

4. ALGORITHMS

We present a family of *Trait Composition Analysis* (TCA) call graph construction algorithms using generic inference rules, in the same style that we used in Section 2. The algorithms presented here are as follows: TCA^{names}, a variant of RA that considers only types instantiated in reachable code; TCA^{bounds}, a variant of RTA adapted to deal with Scala's trait composition and abstract type members; TCA^{expand}, which handles abstract type members more precisely; and $\text{TCA}^{expand\text{-}this}$, which is more precise for call sites where the receiver is this.

We will use the example program shown in Figure 9 to illustrate differences between the algorithms. When executed, the call site on line 201 calls method B.foo. As we shall see, our different algorithms resolve this call site to various subsets of the foo methods in classes A, B, C, and D.

4.1. TCA^{names}

The RA algorithm of Section 2 is sound for Scala because it resolves calls based only on method names and makes no use of types. However, it is imprecise because it considers as possible call targets all methods that have the appropriate name, even those in unreachable code. For Figure 9, RA resolves the call site as possibly calling all four foo methods, even though D is never instantiated in code reachable from main. Since RA already computes a set R of reachable methods, we extend it to consider only classes and traits instantiated in reachable methods.

To this end, we add rule RTA_{NEW} from RTA, which computes a set $\hat{\Sigma}$ of types instantiated in reachable methods. The CALL rule is adapted as follows to make use of $\hat{\Sigma}$:

$$\frac{\begin{array}{c}\text{call } c : e.m(\ldots) \text{ occurs in method } M \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{names}_{\text{CALL}}.$$

We use shading to highlight which parts of the rule are modified relative to the preceding rule RTA_{CALL}.

The resulting TCA^{names} analysis consists of the rule RTA_{NEW} and the rules of RA, except that RA_{CALL} is replaced with $\text{TCA}^{names}_{\text{CALL}}$. In $\text{TCA}^{names}_{\text{CALL}}$, a method is considered as a possible call target only if it is a member of some type C that has been instantiated in a reachable method in R. Calls on super require special handling, as will be discussed in Section 6.

For the program of Figure 9, TCA^{names} resolves the call site to A.foo, B.foo, and C.foo, but not D.foo because D is never instantiated in reachable code.

4.2. TCA^{bounds}

To improve precision, analyses such as RTA and CHA use the static type of the receiver e to restrict its possible runtime types. Specifically, the runtime type C of the receiver of the call must be a subtype of the static type of e.

A key difficulty when analyzing a language with traits is enumerating all subtypes of a type, as both CHA and RTA do in the condition $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ in rules CHA_{CALL} and RTA_{CALL} of Section 2. Given a trait T, any composition of traits containing T is a subtype of T. Therefore, enumerating possible subtypes of T requires enumerating all compositions of traits. Since a trait composition is an ordered list of traits, the number of possible compositions is exponential in the number of traits.[4]

In principle, an analysis could make the conservative assumption that all compositions of traits are possible and therefore that any method defined in any trait could override any other method of the same name and signature in any other trait (a concrete method overrides another method with the same name and signature occurring later in the linearization of a trait composition). The resulting analysis would have the same precision as the TCA^{names} algorithm, though it obviously would be much less efficient.

Therefore, we consider only combinations of traits occurring in reachable methods of the program. This set of combinations is used to approximate the behavior of method calls. In essence, this is similar to the closed-world assumption of RTA. Specifically, the TCA^{bounds} analysis includes the rule RTA_{NEW} to collect the set $\hat{\Sigma}$ of trait combinations occurring at reachable allocation sites. The resulting set is used in the following call resolution rule:

$$\frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M \\ C \in \mathit{SubTypes}(\mathit{StaticType}(e)) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{bounds}_{\text{CALL}}.$$

The added check $C \in \mathit{SubTypes}(\mathit{StaticType}(e))$ relies on the subtyping relation defined in the Scala language specification, which correctly handles complexities of the Scala type system such as path-dependent types.

According to Scala's definition of subtyping, abstract types do not have subtypes, so $\text{TCA}^{bounds}_{\text{CALL}}$ does not apply. Such a definition of subtyping is necessary because it cannot be determined locally (just from the abstract type) which actual types will be bound

[4]Although some trait compositions violate the well-formedness rules of Scala, such violations are unlikely to substantially reduce the exponential number of possible compositions. Moreover, the well-formedness rules are defined in terms of the members of a specific composition, so it would be difficult to enumerate only well-formed compositions without first examining all of them.

to it elsewhere in the program. However, every abstract type in Scala has an upper bound (if it is not specified explicitly, scala.Any is assumed), so an abstract type T can be approximated using its upper bound B:

$$\frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M \\ \textit{StaticType}(e) \text{ is an abstract type with upper bound } B \\ C \in \textit{SubTypes}(\,B\,) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{\textit{bounds}}_{\text{ABSTRACT-CALL}}.$$

For the program of Figure 9, TCA^{bounds} resolves the call site to A.foo and B.foo, but not D.foo because D is never instantiated, and not C.foo because C is not a subtype of A, the upper bound of the static type T of the receiver.

4.3. TCA^{expand}

The TCA^{bounds} analysis is particularly imprecise for abstract types that do not have a declared upper bound, since using the default upper bound of scala.Any makes the bound-based analysis as imprecise as the name-based analysis.

It is more precise to consider only concrete types with which each abstract type is instantiated, similar to the approach of Sallenave and Ducourneau [2012]. To this end, we introduce a mapping *expand*(), which maps each abstract type T to those concrete types with which it has been instantiated:

$$\frac{\begin{array}{c}C \in \hat{\Sigma} \\ \text{“type } A = B\text{” is a member of } C \\ D \text{ is a supertype of } C \\ \text{“type } A\text{” is a member of } D\end{array}}{B \in \textit{expand}(D.A)}\ \text{TCA}^{\textit{expand}}_{\text{EXPAND-TYPE}}$$

$$\frac{\begin{array}{c}C \in \hat{\Sigma} \\ \text{“trait } A\,\{\ldots\}\text{” is a member of } C \\ D \text{ is a supertype of } C \\ \text{“type } A\text{” is a member of } D\end{array}}{C.A \in \textit{expand}(D.A)}\ \text{TCA}^{\textit{expand}}_{\text{EXPAND-TRAIT}}$$

$$\frac{\begin{array}{c}R \in \textit{expand}(S) \\ S \in \textit{expand}(T)\end{array}}{R \in \textit{expand}(T)}\ \text{TCA}^{\textit{expand}}_{\text{EXPAND-TRANS}}.$$

Similar rules (not shown) are needed to handle the type parameters of generic types and type-parametric methods. Our implementation fully supports these cases.

The $\text{TCA}^{\textit{bounds}}_{\text{CALL}}$ rule is then updated to use the *expand*() mapping to determine the concrete types bound to the abstract type of a receiver:

$$\frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M \\ \textit{StaticType}(e) \text{ is an abstract type } T \text{ with upper bound } B \\ C \in \textit{SubTypes}(\,\textit{expand}(T) \cap \textit{SubTypes}(B)\,) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{\textit{expand}}_{\text{ABSTRACT-CALL}}.$$

Rule $\text{TCA}^{expand}_{\text{EXPAND-TYPE}}$ handles situations such as the one where a type alias type A = B is a member of some instantiated trait composition C. Now, if a supertype D of C declares an abstract type A, then B is a possible concrete instantiation of the abstract type D.A, and this fact is recorded in the *expand*() mapping by $\text{TCA}^{expand}_{\text{EXPAND-TYPE}}$. Rule $\text{TCA}^{expand}_{\text{EXPAND-TRAIT}}$ handles a similar case, where an abstract type is instantiated by defining a member trait with the same name. The right-hand side of a type alias might be abstract, so it is necessary to compute the transitive closure of the *expand*() mapping (rule $\text{TCA}^{expand}_{\text{EXPAND-TRANS}}$).

Cycles among type aliases may exist. In Scala, cyclic references between type members are a compile-time error. However, recursion in generic types is allowed. For example, the parameter B in a generic type A[B] could be instantiated with A[B] itself, leading to B representing an unbounded sequence of types A[B], A[A[B]], ... This kind of recursion can be detected either by limiting the size of $expand(T)$ for each abstract type to some fixed bound or by checking for occurrences of T in the expansion $expand(T)$. The current version of our implementation limits the size of $expand(T)$ to 1,000 types. This bound is never exceeded in our experimental evaluation, implying that recursive types do not occur in the benchmark programs. The same issue also occurs in Java and C# and has been previously noted by Sallenave and Ducourneau [2012]. Their implementation issues a warning when it detects the situation. Our algorithm resolves the issue soundly: when a recursive type T is detected, the algorithm falls back to using the upper bound of T to resolve calls on receivers of type T.

4.4. $\text{TCA}^{expand\text{-}this}$

In both Java and Scala, calls on the this reference are common. In some cases, it is possible to resolve such calls more precisely by exploiting the knowledge that the caller and the callee must be members of the same object. Care must be taken in the presence of super-calls, as will be discussed in Section 6.1.

For example, at the call this.foo() on line 83 of Figure 5, the static type of the receiver this is A, which has both B and C as subtypes. Since B and C are both instantiated, all of the analyses described so far would resolve the call to both B.foo (line 87) and C.foo (line 90). However, any object that has C.foo as a member also has C.bar as a member, which overrides the method A.bar containing the call site. Therefore, the call site at line 83 can never resolve to method C.foo.

This pattern is handled precisely by the following rule:

$$\frac{\begin{array}{c}\text{call } D.this.m(\ldots) \text{ occurs in method } M \\ D \text{ is the declaring trait of } M \\ C \in SubTypes(D) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ \text{method } M \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{expand\text{-}this}_{\text{THIS-CALL}}.$$

The rule requires not only the callee M' but also the caller M to be members of the same instantiated type C. The rule applies only when the receiver is the special variable this. Because nested classes and traits are common in Scala, it is possible that a particular occurrence of the special variable this is qualified to refer to the enclosing object of some outer trait. Since it would be unsound to apply $\text{TCA}^{expand\text{-}this}_{\text{THIS-CALL}}$ in this case, we require that the receiver be the special variable this of the innermost trait D that declares the caller method M.

After adding rule $\text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}$, we adapt rule $\text{TCA}^{\textit{bounds}}_{\text{CALL}}$ by adding a precondition so that it does not apply when $\text{TCA}^{\textit{expand-this}}_{\text{THIS-CALL}}$ should, that is, when the receiver is the special variable this of the declaring trait D of the caller method M. We designate the adapted rule $\text{TCA}^{\textit{expand-this}}_{\text{CALL}}$:

$$\frac{\begin{array}{c}\text{call } e.m(\ldots) \text{ occurs in method } M \\ e \text{ is not } D.\textit{this}, \text{ where } D \text{ is the declaring trait of } M \\ C \in \textit{SubTypes}(\textit{StaticType}(e)) \\ \text{method } M' \text{ has name } m \\ \text{method } M' \text{ is a member of type } C \\ M \in R \qquad C \in \hat{\Sigma}\end{array}}{c \mapsto M'}\ \text{TCA}^{\textit{expand-this}}_{\text{CALL}}.$$

5. FORMALIZATION

In this section, we provide a formalization of the inference rules for $\text{TCA}^{\textit{expand-this}}$ based on the FS_{alg} (Featherweight Scala) representation of Cremet et al. [2006]. We also prove the $\text{TCA}^{\textit{expand-this}}$ analysis correct with respect to the operational semantics of FS_{alg} by demonstrating the following.

(1) For any FS_{alg} program P, the set of methods called in an execution trace of P is a subset of the set R of reachable methods computed for P by $\text{TCA}^{\textit{expand-this}}$.
(2) For any FS_{alg} program P, if the execution trace of P contains a call from call site c to a target method M, then $\text{TCA}^{\textit{expand-this}}$ applied to P derives $c \mapsto M$.

5.1. Extensions to Featherweight Scala

Cremet et al. [2006] present an operational semantics and type system for a core Scala subset FS_{alg}, which is commonly referred to as "Featherweight Scala." FS_{alg} includes traits, abstract type members, singleton types, path-dependent types, and dynamic dispatch.

We make one minor extension to FS_{alg}. In FS_{alg}, a program that calls a method on a receiver whose static type is an abstract type member is never well-formed, because abstract type members do not themselves have any members and therefore no methods. However, we feel that disallowing such calls would be unrealistic because they are quite common in full Scala and because resolving them requires reasoning about the way in which traits are composed, as we discussed in Section 3.2. Fortunately, such calls can be supported in a straightforward manner, by requiring abstract type members to have upper bounds, just like in full Scala. In full Scala, abstract type members must also have lower bounds, which we do not add here because they are not used by $\text{TCA}^{\textit{expand-this}}$. In particular, in full Scala, each abstract type member has an upper bound with scala.Any as the default if none is explicitly specified. If e is an expression whose declared type is an abstract type member T, then calling methods on e that are declared in the upper bound of T is legal. This allows us to model situations such as the one in Figure 2, where type Z.B is declared to have a bound of HasFoo on line 34, thus permitting the call o.foo on line 36.

In the remainder of this section, we review the syntax, operational semantics, and typing rules of FS_{alg}. In the process, we also formalize the small extension that we have just described. The extension to FS_{alg} is identified by shading.

x, y, z, φ	Variable
a	Value label
A	Type label
$P ::=$	Program
$\quad \{x \mid \overline{M}\ t\}$	
$M, N ::=$	Member decl
$\quad \texttt{val}_n a : T\ (= t)^?$	Field decl
$\quad \texttt{def}_n a(\overline{y : S}) : T (= t)^?$	Method decl
$\quad \texttt{type}_n A = T$	Type decl
$\quad \texttt{type}_n A <: U$	Bounded type decl
$\quad \texttt{trait}_n A\ \texttt{extends}(\overline{T})\ \{\varphi \mid \overline{M}\}$	Class decl
$s, t, u ::=$	Term
$\quad x$	Variable
$\quad t.a$	Field selection
$\quad s.a(\overline{t})$	Method call
$\quad \texttt{val}\ x = \texttt{new}\ T;\ t$	Object creation
$p ::=$	Path
$\quad x$	Variable
$\quad p.a$	Field Selection
$S, T, U ::=$	Type
$\quad p.A$	Type selection
$\quad p.\mathbf{type}$	Singleton type
$\quad (\overline{T})\ \{\varphi \mid \overline{M}\}$	Type signature

Fig. 10. Featherweight Scala syntax.

Figure 10 shows the syntax of FS_{alg}. An FS_{alg} *program* is a term which is usually of the form

$$\texttt{val}\ z = \texttt{new}\ \{\varphi \mid \overline{M}\};\ t.$$

Here, z is a universe object consisting of a list of member declarations $\overline{M}$, t is some term to be evaluated in the context of z, and φ is the name of the self-reference. The only change to the original definition by Cremet et al. [2006] is the addition of an upper bound U to abstract type member declarations in order to allow calls on variables whose type is an abstract type member, as illustrated in Figure 2.

Figure 11 presents an operational semantics for FS_{alg} in the form of a relation $\rightarrow$. Here, $\Sigma; t \rightarrow \Sigma'; t'$ means that a term t in the context of an *evaluation environment* Σ can be rewritten to a term t' in the context of an environment Σ'. These rules are unchanged from Cremet et al. [2006].

Figure 12 presents a set of lookup judgments that defines the set of members of a given object based on the type signature that it is instantiated with. These judgments take the form $\Sigma \vdash T \prec_{\varphi} \overline{M}$, indicating that in the context of a runtime environment Σ, the type T contains a set of members $\overline{M}$. Here, we added a judgment to handle the case for a type of the form $y.A$, where A is a type with a bound T. In this case, the set of members in $y.A$ is the same as the set of members present in T.

The $\uplus$ operator used in Figure 12 deserves some additional discussion. In FS_{alg} [Cremet et al. 2006], the $\uplus$ operator is defined as concatenation with rewriting of common members: $\overline{M} \uplus \overline{N} = \overline{M}|_{\mathrm{dom}(\overline{M})\setminus\mathrm{dom}(\overline{N})}, \overline{N}$, where $\mathrm{dom}(\overline{M})$ is the set of labels

$$\frac{\texttt{val}_n a : T = t \in \Sigma(x)}{\Sigma;\, x.a \to \Sigma\,;\, t} \quad (\textsc{red-value})$$

$$\frac{\Sigma \vdash T \prec_x \overline{M}}{\Sigma\,;\, \texttt{val}\, x = \texttt{new}\, T; t \to \Sigma, x : \overline{M}\,;\, t} \quad (\textsc{red-new})$$

$$\frac{\texttt{def}_n a(\overline{z : S}) : T = t \in \Sigma(x)}{\Sigma\,;\, x.a(\overline{y}) \to \Sigma\,;\, [\overline{y}/\overline{z}]t} \quad (\textsc{red-method})$$

$$\frac{\Sigma\,;\, t \to \Sigma'\,;\, t'}{\Sigma\,;\, e[t] \to \Sigma'\,;\, e[t']} \quad (\textsc{red-context})$$

where

$e ::=$ **(term evaluation context)**

$\langle\rangle$

$e.a$

$e.a(t)$

$x.a(\overline{s},\, e,\, \overline{u})$

$\texttt{val}\, x = \texttt{new}\, E;\, t$

$E ::=$ **(type evaluation context)**

$e.a$

$(\overline{T},\, E,\, \overline{U})\, \{\varphi \mid \overline{M}\}$

Fig. 11. Reduction.

$$\frac{\forall i,\ \Sigma \vdash T_i \prec_\varphi \overline{N_i}}{\Sigma \vdash (\overline{T})\, \{\varphi \mid \overline{M}\} \prec_\varphi (\biguplus_i \overline{N_i}) \uplus \overline{M}} \quad (\textsc{lookup-sig})$$

$$\frac{\texttt{trait}_n A\, \texttt{extends}\, (\overline{T})\, \{\varphi \mid \overline{M}\} \in \Sigma(y) \quad \Sigma \vdash (\overline{T})\, \{\varphi \mid \overline{M}\} \prec_\varphi \overline{N}}{\Sigma \vdash y.A \prec_\varphi \overline{N}} \quad (\textsc{lookup-class})$$

$$\frac{\texttt{type}_n A = T \in \Sigma(y) \quad \Sigma \vdash T \prec_\varphi \overline{M}}{\Sigma \vdash y.A \prec_\varphi \overline{M}} \quad (\textsc{lookup-alias})$$

$$\frac{\texttt{type}_n A <: U \in \Sigma(y) \quad \Sigma \vdash U \prec_\varphi \overline{M}}{\Sigma \vdash y.A \prec_\varphi \overline{M}} \quad (\textsc{lookup-bounded-type})$$

Fig. 12. Lookup.

$$\frac{x : T \in \Gamma}{\mathcal{S},\ \Gamma \vdash_{path} x : T} \quad \text{(PATH-VAR)}$$

$$\frac{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{val}_n a : T(= t)^?}{\mathcal{S},\ \Gamma \vdash_{path} p.a : T} \quad \text{(PATH-SELECT)}$$

Fig. 13. Path typing.

$$\frac{\mathcal{S},\ \Gamma \vdash_{path} p : T}{\mathcal{S},\ \Gamma \vdash p : p.\mathbf{type}} \quad \text{(PATH)}$$

$$\frac{\begin{array}{c}\mathcal{S},\ \Gamma \vdash s : S \\ \mathcal{S},\ \Gamma \vdash \bar{t} : \overline{T'} \qquad \mathcal{S},\ \Gamma \vdash \overline{T'} <: \overline{T} \\ \mathcal{S},\ \Gamma \vdash S \ni \mathtt{def}_n a(\overline{x : T}) : U(= u)^?\end{array}}{\mathcal{S},\ \Gamma \vdash s.a(\bar{t}) : U} \quad \text{(METHOD)}$$

$$\frac{\begin{array}{c}\mathcal{S},\ \Gamma \vdash t : S \qquad t \text{ is not a path} \\ \mathcal{S},\ \Gamma \vdash S \ni \mathtt{val}_n a : T(= u)^?\end{array}}{\mathcal{S},\ \Gamma \vdash t.a : T} \quad \text{(SELECT)}$$

$$\frac{\begin{array}{c}\mathcal{S},\ \Gamma,\ x : T \vdash t : S \qquad x \notin \mathrm{fn}(S) \\ \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M_c} \qquad \mathcal{S},\ \Gamma \vdash T\ \mathrm{WF}\end{array}}{\mathcal{S},\ \Gamma \vdash \mathtt{val}\ \ x = \mathtt{new}\ \ T;\ t : S} \quad \text{(NEW)}$$

Fig. 14. Type assignment.

defined in $\overline{M}$ and $\overline{M}|_{\mathcal{L}}$ consists of all declarations in $\overline{M}$ that define a label in $\mathcal{L}$. This definition does not distinguish between abstract and concrete members and, in particular, specifies that an abstract member in $\overline{N}$ can override a concrete member in $\overline{M}$. Here, a member declaration is *concrete* if it is a Field decl or Method decl with the optional $= t$, or if it is a Type decl or Class decl, and *abstract* otherwise. In particular, a Bounded type decl is considered abstract. In contrast, the Scala specification [Odersky 2011, Section 5.1] specifies that a concrete member always hides an abstract member regardless of their relative order. To be consistent with Scala, we therefore redefine the $\uplus$ operator as follows: $\overline{M} \uplus \overline{N} = \overline{M}|_{(\mathrm{cdom}(\overline{M}) \setminus \mathrm{cdom}(\overline{N})) \cup (\mathrm{adom}(\overline{M}) \setminus \mathrm{dom}(\overline{N}))}, \overline{N}|_{\mathrm{cdom}(\overline{N}) \cup (\mathrm{adom}(\overline{N}) \setminus \mathrm{cdom}(\overline{M}))}$, where $\mathrm{cdom}(\overline{M})$ is the set of labels defined in concrete declarations in $\overline{M}$ and $\mathrm{adom}(\overline{M})$ is the set of labels defined in abstract declarations in $\overline{M}$.

Figures 13 and 14 define the type assignment relations that determine a static type T for each form of term t. The rules are parameterized with a set of bindings Γ and a set of locked declarations $\mathcal{S}$ used to prevent infinite recursion in types and in type checking. When t is a path, two types are defined: the path-specific type assignment relation $\mathcal{S},\ \Gamma \vdash_{path} \cdot : \cdot$ identifies the general declared type T specified for t by the current environment, whereas the general type assignment relation $\mathcal{S},\ \Gamma \vdash \cdot : \cdot$ assigns t the specific singleton type $t.\mathbf{type}$.

Figures 15 and 16 define the membership relations that define the set of members of a static type T. The expansion judgment $\mathcal{S},\ \Gamma \vdash \cdot \prec_\varphi \cdot$ is the static analogue of the runtime lookup judgment: using a static type environment Γ, it looks up the members of each form of type T. In the original FS_{alg}, an abstract type has no members because

$$\frac{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \texttt{trait}_n A\ \texttt{extends}(\overline{T})\ \{\varphi \mid \overline{M}\} \quad \{n\} \cup \mathcal{S},\ \Gamma \vdash (\overline{T})\ \{\varphi \mid \overline{M}\} \prec_\varphi \overline{N} \qquad n \notin \mathcal{S}}{\mathcal{S},\ \Gamma \vdash p.A \prec_\varphi \overline{N}} \quad (\prec\text{-CLASS})$$

$$\frac{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \underline{\texttt{type}}_n A = T \quad \{n\} \cup \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M} \qquad n \notin \mathcal{S}}{\mathcal{S},\ \Gamma \vdash p.A \prec_\varphi \overline{M}} \quad (\prec\text{-TYPE})$$

$$\frac{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \underline{\texttt{type}}_n A <: U \quad \{n\} \cup \mathcal{S},\ \Gamma \vdash U \prec_\varphi \overline{M} \qquad n \notin \mathcal{S}}{\mathcal{S},\ \Gamma \vdash p.A \prec_\varphi \overline{M}} \quad (\prec\text{-BOUNDED-TYPE})$$

$$\frac{\forall i,\ \mathcal{S},\ \Gamma \vdash T_i \prec_\varphi \overline{N_i}}{\mathcal{S},\ \Gamma \vdash (\overline{T})\ \{\varphi \mid \overline{M}\} \prec_\varphi \left(\biguplus_i \overline{N_i}\right) \uplus \overline{M}} \quad (\prec\text{-SIGNATURE})$$

Fig. 15. Expansion.

$$\frac{\mathcal{S},\ \Gamma \vdash p \simeq q \qquad \mathcal{S},\ \Gamma \vdash_{path} q : T \quad \psi(p) \cup \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M} \qquad \psi(p) \not\subseteq \mathcal{S}}{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni [p/\varphi]M_i} \quad (\ni\text{-SINGLETON})$$

$$\frac{T \text{ is not a singleton type} \quad \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M} \qquad \varphi \notin \text{fn}(M_i)}{\mathcal{S},\ \Gamma \vdash T \ni M_i} \quad (\ni\text{-OTHER})$$

Fig. 16. Membership.

the expansion judgment is not defined for abstract type members of the form $\texttt{type}_n A$.[5] Therefore, in the original FS_{alg}, it was impossible to call a method on a receiver whose type is abstract, since such a type has no member methods. In our extension, we add rule $\prec$-BOUNDED-TYPE that gives bounded abstract types of the form $\texttt{type}_n A <: U$ the members of their bound U. This makes a method call on a (bounded) abstract type possible, as shown in Figure 2. The membership judgment $\mathcal{S},\ \Gamma \vdash \cdot \ni \cdot$ adjusts the members found by the expansion judgment to account for singleton types. When the type of a member involves the self-reference `this`, represented as φ in FS_{alg}, φ is replaced by the actual path p when it occurs in a singleton type $p.\mathbf{type}$, and the member is removed completely when it occurs in a non-singleton type in which no specific value for the self-reference φ is available.

Figure 17 defines the subtyping relation $<:$ between types. TCA$^{expand\text{-}this}$ relies on this relation to decide when a term of a given type S can reduce to an object of some other given type T. The subtyping relation relies on alias expansion relations for types and paths, shown in Figures 18 and 19. The type alias expansion expands type aliases of the form $\texttt{type}_n A = T$ so that subtypes of T can also be considered subtypes of A

[5]In every member declaration such as $\texttt{type}_n A$, the subscript n is a unique label that is used in FS_{alg} to give an identity to that particular declaration.

$$\frac{\mathcal{S},\ \Gamma \vdash T \simeq T' \qquad \mathcal{S},\ \Gamma \vdash U \simeq U' \qquad \mathcal{S},\ \Gamma \vdash_* T' <: U'}{\mathcal{S},\ \Gamma \vdash T <: U} \quad (<: -\textsc{unalias})$$

$$\frac{A \neq A' \qquad \{n\} \cup \mathcal{S},\ \Gamma \vdash T_i <: p'.A' \qquad n \notin \mathcal{S} \qquad \mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{trait}_n A\, \mathtt{extends}(\overline{T})\, \{\varphi \mid \overline{M}\}}{\mathcal{S},\ \Gamma \vdash_* p.A <: p'.A'} \quad (<: -\textsc{class})$$

$$\frac{\mathcal{S},\ \Gamma \vdash p \simeq p' \qquad \mathcal{S},\ \Gamma \vdash q \simeq p'}{\mathcal{S},\ \Gamma \vdash_* p.\mathbf{type} <: q.\mathbf{type}} \quad (<: -\textsc{singleton-right})$$

$$\frac{\exists i,\ \mathcal{S},\ \Gamma \vdash T_i <: p.A}{\mathcal{S},\ \Gamma \vdash_* (\overline{T})\, \{\varphi \mid \overline{M}\} <: p.A} \quad (<: -\textsc{sig-left})$$

$$\frac{U \text{ is not a singleton type} \qquad \mathcal{S},\ \Gamma \vdash p \simeq q \qquad \mathcal{S},\ \Gamma \vdash_{path} q : T \qquad \mathcal{S},\ \Gamma \vdash T <: U}{\mathcal{S},\ \Gamma \vdash_* p.\mathbf{type} <: U} \quad (<: -\textsc{singleton-left})$$

$$\frac{\mathcal{S},\ \Gamma \vdash p \simeq p' \qquad \mathcal{S},\ \Gamma \vdash q \simeq p'}{\mathcal{S},\ \Gamma \vdash_* p.A <: q.A} \quad (<: -\textsc{paths})$$

$$\frac{T \text{ is not a singleton type} \qquad \forall i,\ \mathcal{S},\ \Gamma \vdash T <: T_i \qquad \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{N} \qquad \mathrm{dom}(\overline{M}) \subseteq \mathrm{dom}(\overline{N}) \qquad \mathcal{S},\ \Gamma \vdash \overline{N} \ll \overline{M}}{\mathcal{S},\ \Gamma \vdash_* T <: (\overline{T})\, \{\varphi \mid \overline{M}\}} \quad (<: -\textsc{sig-right})$$

Fig. 17. Algorithmic subtyping.

$$\frac{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{type}_n A = T \qquad \{n\} \cup \mathcal{S},\ \Gamma \vdash T \simeq U \qquad n \notin \mathcal{S}}{\mathcal{S},\ \Gamma \vdash p.A \simeq U} \quad (\simeq -\textsc{type})$$

$$\frac{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{trait}_n A\, \mathtt{extends}(\overline{T})\, \{\varphi \mid \overline{M}\}}{\mathcal{S},\ \Gamma \vdash p.A \simeq p.A} \quad (\simeq -\textsc{class})$$

$$\mathcal{S},\ \Gamma \vdash (\overline{T})\, \{\varphi \mid \overline{M}\} \simeq (\overline{T})\, \{\varphi \mid \overline{M}\} \quad (\simeq -\textsc{signature})$$

$$\frac{\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{type}_n A <: U}{\mathcal{S},\ \Gamma \vdash p.A \simeq p.A} \quad (\simeq -\textsc{bounded-type})$$

$$\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \simeq p.\mathbf{type} \quad (\simeq -\textsc{singleton})$$

Fig. 18. Type alias expansion.

$$\frac{\mathcal{S},\ \Gamma \vdash_{path} p : q.\mathbf{type} \qquad \psi(p) \cup \mathcal{S},\ \Gamma \vdash q \simeq q' \qquad \psi(p) \not\subseteq S}{\mathcal{S},\ \Gamma \vdash p \simeq q'} \quad (\simeq -\textsc{step})$$

$$\frac{\mathcal{S},\ \Gamma \vdash_{path} p : T \qquad T \text{ is not a singleton type}}{\mathcal{S},\ \Gamma \vdash p \simeq p} \quad (\simeq -\textsc{refl})$$

Fig. 19. Path alias expansion.

$$\frac{\mathcal{S},\ \Gamma \vdash T <: T'}{\mathcal{S},\ \Gamma \vdash \mathtt{val}_n a : T(=t)^? <: \mathtt{val}_m a : T'(=t')^?} \quad (<: -\text{MEMBER-FIELD})$$

$$\mathcal{S},\ \Gamma \vdash \mathtt{type}_n A = T <: \mathtt{type}_n A = T \quad (<: -\text{MEMBER-TYPE})$$

$$\frac{\mathcal{S},\ \Gamma \vdash T <: U}{\mathcal{S},\ \Gamma \vdash \mathtt{type}_n A = T <: \mathtt{type}_n A <: U} \quad (<: -\text{BOUNDED-TYPE})$$

$$\mathcal{S},\ \Gamma \vdash \mathtt{trait}_n A\,\mathtt{extends}(\overline{T})\,\{\varphi \mid \overline{M}\} <: \mathtt{trait}_n A\,\mathtt{extends}(\overline{T})\,\{\varphi \mid \overline{M}\} \quad (<: -\text{MEMBER-CLASS})$$

$$\frac{\mathcal{S},\ \Gamma \vdash \overline{S'} <: \overline{S} \qquad \mathcal{S},\ \Gamma \vdash T <: T'}{\mathcal{S},\ \Gamma \vdash \mathtt{def}_n a(\overline{x : S}) : T(=t)^? <: \mathtt{def}_m a(\overline{x : S'}) : T'(=t')^?} \quad (<: -\text{MEMBER-METHOD})$$

Fig. 20. Member subtyping.

within the context of the object that defines the type alias $\mathtt{type}_n A = T$. The path alias expansion unifies the singleton types p.**type** and q.**type** when the type of p is q.**type**. The subtyping relation also uses the member subtyping relation, shown in Figure 20, to enforce that a type is allowed to extend another type only when their respective members have compatible types.

5.2. Formalization of TCA$^{expand\text{-}this}$ for FS_{alg}

In this section, we present a formalization of the TCA$^{expand\text{-}this}$ algorithm on FS_{alg}. The full TCA$^{expand\text{-}this}$ algorithm formalized for FS_{alg} is presented in Figure 21.

In our formalization of TCA$^{expand\text{-}this}$, we drop the $\Gamma, \mathcal{S}$ for brevity because it is uniquely determined by the identity of the particular term t and its position within the overall term representing the whole program.

FS_{alg} defines paths as a subset of terms that consist of a variable followed by zero or more field dereferences. The type of a path p is defined to be the singleton type p.**type** containing only the object designated by p. A separate typing judgment of the form $\Gamma, \mathcal{S} \vdash_{path} p : T$ assigns p the wider type T determined by the declared types of the variable and fields referenced in p. In our definition of TCA$^{expand\text{-}this}$, it is this latter type that we need, because it determines the possible methods that could be members of objects pointed to by p. Therefore, in the formalization of TCA$^{expand\text{-}this}$, whenever we write $t : T$, we mean $\Gamma, \mathcal{S} \vdash_{path} t : T$ when t is a path, and $\Gamma, \mathcal{S} \vdash t : T$ when t is not a path.

Rules $\text{TCA}^{expand\text{-}this}_{\text{MAIN}}$ and $\text{TCA}^{expand\text{-}this}_{\text{REACHABLE}}$ assert that the *main* method is reachable and that if any method M is called from some call site c, then that M is reachable.

Rule $\text{TCA}^{expand\text{-}this}_{\text{NEW}}$ accumulates the allocation sites that occur in reachable methods into a set $\hat{\Sigma}$. In FS_{alg}, an allocation site is a term of the form $\mathtt{val}\ x = \mathtt{new}\ T; u$. The subterm u is evaluated in a context in which the variable x is bound to the newly-allocated object of type T. It is at allocation sites that traits are finally composed into the types of actual runtime objects. Therefore, the set $\hat{\Sigma}$ contains all compositions of traits that occur in reachable parts of the program. In addition to the composed type T, $\hat{\Sigma}$ also collects the corresponding variable x that holds the instantiated object because it will be required by the rule $\text{TCA}^{expand\text{-}this}_{\text{EXPAND-TRAIT}}$.

The informal rules $\text{TCA}^{expand\text{-}this}_{\text{CALL}}$ and $\text{TCA}^{expand\text{-}this}_{\text{CALL-THIS}}$ from Section 4 are merged into a single rule $\text{TCA}^{expand\text{-}this}_{\text{CALL}}$ in the formalization on FS_{alg}. To determine the possible subtypes of the receiver type S, the rule first considers all types T that occur in the set of reachable compositions $\hat{\Sigma}$, then selects only those that are subtypes of the receiver type ($T <: S$), and subsequently selects the method M' that is a member of

$$\frac{}{\mathit{main} \quad\quad \in R}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{MAIN}}$$

$$\frac{\begin{array}{c}\text{call } c : s.a(\overline{t}) \text{ occurs in method } M,\ s : S \\ (x, T) \in \hat{\Sigma} \\ T \ni M',\ M' \equiv \mathtt{def}_n a(\overline{y : S'}) : T'(= t')^? \\ T <: S, (S \ni M \wedge s \equiv \varphi) \Rightarrow (T \ni M)\end{array}}{(M \in R)\ \Rightarrow\ c \mapsto M'}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{CALL}}$$

$$\frac{\mathtt{val}\ x = \mathtt{new}\ T; u \text{ occurs in } M}{(M \in R)\ \Rightarrow\ ((x, T) \in \hat{\Sigma})}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{NEW}}$$

$$\frac{c \mapsto M}{M \in R}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{REACHABLE}}$$

$$\frac{\begin{array}{c}\text{call } c : s.a(\overline{t}) \text{ occurs in method } M,\ s : p.A \\ p.\mathbf{type} \ni \mathtt{type}_n A <: U \\ (x, T) \in \hat{\Sigma} \\ T \ni M',\ M' \equiv \mathtt{def}_n a(\overline{y : S'}) : T'(= t')^? \\ T <: S,\ S \in \mathit{expand}(\mathtt{type}_n A), S <: U\end{array}}{(M \in R)\ \Rightarrow\ c \mapsto M'}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{ABSTRACT\text{-}CALL}}$$

$$\frac{\begin{array}{c}(x, T) \in \hat{\Sigma} \\ T \ni^{\cup} \mathtt{type}_n A <: U,\ T \ni \mathtt{type}_{n'} A = T'\end{array}}{T' \in \mathit{expand}(\mathtt{type}_n A)}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{EXPAND\text{-}TYPE}}$$

$$\frac{\begin{array}{c}(x, T) \in \hat{\Sigma} \\ T \ni^{\cup} \mathtt{type}_n A <: U,\ T \ni \mathtt{trait}_{n'} A\ \mathtt{extends}\ (\overline{T'})\{\varphi \mid \overline{M}\}\end{array}}{x.A \in \mathit{expand}(\mathtt{type}_n A)}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{EXPAND\text{-}TRAIT}}$$

$$\frac{\begin{array}{c}p.A' \in \mathit{expand}(\mathtt{type}_n A) \\ p.\mathbf{type} \ni \mathtt{type}_{n'} A' <: U' \\ T \in \mathit{expand}(\mathtt{type}_{n'} A')\end{array}}{T \in \mathit{expand}(\mathtt{type}_n A)}\ \mathrm{TCA}^{\mathit{expand\text{-}this}}_{\mathrm{EXPAND\text{-}TRANS}}$$

Fig. 21. $\mathrm{TCA}^{\mathit{expand\text{-}this}}$ formalized for FS_{alg}.

T with a name and signature that are consistent with the call site c. The precondition $(S \ni M \wedge s \equiv \varphi) \Rightarrow (T \ni M)$ implements the special handling of calls on the special variable this. In words, the condition states that if the receiver s of a call is the special variable this (which is denoted φ in FS_{alg}), then the runtime type T of the receiver must contain the caller M as a member. Because it is common in Scala to nest traits within other traits, it is a nontrivial detail to recognize whether the receiver is the particular variable this on which the caller method was called; in general, the variable this could refer to any of the outer objects that encloses the current trait. The precondition $(S \ni M \wedge s \equiv \varphi) \Rightarrow (T \ni M)$ applies only if the static type of the receiver this contains the caller method $(S \ni M)$. If it does not (i.e., the receiver is a different variable this than the receiver of the caller), then the left-hand side of the implication is false, so the precondition is trivially true.

$$\frac{\mathcal{S},\ \Gamma \vdash T \ni M}{\mathcal{S},\ \Gamma \vdash T \ni^{\cup} M} \quad (\ni^{\cup}\text{-BASE})$$

$$\frac{\begin{array}{c}\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{trait}_n A\ \mathtt{extends}(\overline{T})\ \{\varphi \mid \overline{M}\} \\ \mathcal{S},\ \Gamma \vdash T_i \ni^{\cup} N\end{array}}{\mathcal{S},\ \Gamma \vdash p.A \ni^{\cup} N} \quad (\ni^{\cup}\text{-CLASS})$$

$$\frac{\begin{array}{c}\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{type}_n A = T \\ \mathcal{S},\ \Gamma \vdash T \ni^{\cup} N\end{array}}{\mathcal{S},\ \Gamma \vdash p.A \ni^{\cup} N} \quad (\ni^{\cup}\text{-ALIAS})$$

$$\frac{\begin{array}{c}\mathcal{S},\ \Gamma \vdash p.\mathbf{type} \ni \mathtt{type}_n A <: U \\ \mathcal{S},\ \Gamma \vdash U \ni^{\cup} N\end{array}}{\mathcal{S},\ \Gamma \vdash p.A \ni^{\cup} N} \quad (\ni^{\cup}\text{-BOUNDED})$$

$$\frac{\mathcal{S},\ \Gamma \vdash T_i \ni^{\cup} N}{\mathcal{S},\ \Gamma \vdash (\overline{T})\ \{\varphi \mid \overline{M}\} \ni^{\cup} N} \quad (\ni^{\cup}\text{-SIGNATURE})$$

Fig. 22. Ancestor membership.

In rule $\text{TCA}^{\textit{expand-this}}_{\text{CALL-ABSTRACT}}$, the static type of the receiver is a path-dependent type $(s : p.A)$, and the member A of the object p is an abstract type member $(p.\mathbf{type} \ni \mathtt{type}_n A)$. Here, the difficulty is that the subtyping rules in Scala and FS_{alg} do not define any subtypes for an abstract type member, so there is no direct way to determine which types of objects may be assigned to the receiver. Therefore, in order to determine which types T of objects may be bound to the receiver s, the analysis must first determine which concrete types may be bound to the abstract type member $p.A$. This is accomplished using rules $\text{TCA}^{\textit{expand-this}}_{\text{EXPAND-TYPE}}$, $\text{TCA}^{\textit{expand-this}}_{\text{EXPAND-TRAIT}}$, and $\text{TCA}^{\textit{expand-this}}_{\text{EXPAND-TRANS}}$, which generate the set $\textit{expand}(\mathtt{type}_n A)$ of all concrete types that are possibly assigned to the abstract type member A. The key complication here is that the call site $s.a(\overline{t})$ may occur in one trait, and the actual static type of s may be assigned to $p.A$ in a different trait, so the actual static type of s is not known until the traits are composed at an allocation site of the enclosing object.

In rule $\text{TCA}^{\textit{expand-this}}_{\text{EXPAND-TYPE}}$, we examine the set of all reachable trait compositions $\hat{\Sigma}$ for the possible type T of the enclosing object. If some supertype of T declares an abstract type member $\mathtt{type}_n A <: U$, and T contains a member $\mathtt{type}_{n'} A = T'$, then the abstract type member A represents the actual type T' in T, so we want to include T' in $\textit{expand}(\mathtt{type}_n A)$ as one of the possible actual types represented by the abstract type $\mathtt{type}_n A$. In order to determine whether some supertype of T declares an abstract type member $\mathtt{type}_n A <: U$, we define a new relation $T \ni^{\cup} \mathtt{type}_n A <: U$ to hold whenever there exists a supertype S of T such that $S \ni \mathtt{type}_n A <: U$. This ancestor membership relation is formally defined in Figure 22. The definition follows the form of the FS_{alg} membership relation from Figure 16 but extends the members of a type to also include members of all of its supertypes.

In addition to an explicit type alias declaration $\mathtt{type}_{n'} A = T'$, it is also possible in full Scala (but not FS_{alg}) to define the actual type assigned to an abstract type $\mathtt{type}_n A <: U$ implicitly, by mixing in the trait that defines $\mathtt{type}_n A <: U$ with another trait that contains a member trait $\mathtt{trait}_{n'} A$ of the same name A. This case is handled

by rule $\text{TCA}^{expand\text{-}this}_{\text{EXPAND-TRAIT}}$. Although this rule is not necessary for FS_{alg}, which does not support this mechanism for binding abstract types, we present it here for completeness to show how it appears in our implementation for full Scala.

Finally, the type that is assigned to an abstract type member may itself be abstract. Therefore, to determine the actual types represented by an abstract type member, the *expand*() relation must be transitively closed. The transitive closure of *expand*() is ensured by the inference rule $\text{TCA}^{expand\text{-}this}_{\text{EXPAND-TRANS}}$.

5.3. Correctness Proof

The definition of FS_{alg} [Cremet et al. 2006] defines both an operational semantics and a type system, but it does not state or prove a theorem of type soundness relating the two, although soundness is clearly intended. Although the correctness of $\text{TCA}^{expand\text{-}this}$ relies on the soundness of the FS_{alg} type system, we consider it outside the scope of this article to prove type soundness for FS_{alg}. Instead, we prove that $\text{TCA}^{expand\text{-}this}$ is correct for all executions that respect the FS_{alg} type system. If the FS_{alg} type system is indeed sound, then this covers all possible executions.

The correctness of $\text{TCA}^{expand\text{-}this}$ depends on the type soundness of FS_{alg} in two places. The obvious dependence is in the computation of the possible target methods of a call site. The targets depend on the runtime types of the possible receivers, and $\text{TCA}^{expand\text{-}this}$ approximates those using the declared type of the receiver. For correctness, it is necessary that the runtime types of the receiver truly are subtypes of the declared type. This requirement is exhibited as a dependence of Lemma 5.4 on Assumption 1. A second, more subtle dependence on type soundness is in the computation of the *expand*($\cdot$) sets. Since types in Scala are path-dependent, to determine the possible concrete types of an abstract type $p.T$ with path p, it is necessary to correctly determine the possible objects that p could be instantiated to. The $\text{TCA}^{expand\text{-}this}$ analysis does this using the declared type of p, and its correctness therefore depends on the declared type of p soundly abstracting the actual objects that p could be instantiated to. This requirement is exhibited as a dependence of Lemma 5.3 on Assumption 1.

It is typical for a definition of type soundness to assert that if a term s has type S and reduces to value x with type T, then $T <: S$ (preservation). Such a definition fails for FS_{alg} when S is an abstract type $p.A$. In such cases, the concrete type of S is unknown until execution time when the trait declaring abstract type member A is instantiated with a type alias that binds A to a concrete type. The FS_{alg} subtype relation therefore defines an abstract type to have no subtypes. Thus, under the common definition of type soundness, terms whose type is abstract could not be reduced, and abstract types would become useless. To get around this extreme restriction, we define an extended subtype relation $<<:$ that considers abstract types. We can then define an analogue of type preservation as follows: if term s with type S reduces to value x with type T, then $T <<: S$. However, unlike the normal subtype relation $<:$, our extended subtype relation $<<:$ must depend on some specific execution trace that specifies how the abstract type member A has been instantiated.

Definition 5.1. In the context of a given dynamic execution trace τ, $T <<:_\tau S$ if either

(1) $T <: S$, or
(2) (a) the type S has the form $p.A$,
    (b) A is an abstract type declaration in p,
    (c) in the execution trace τ, p reduces to some variable x,

(d) there exist types S' and T' such that $p : S'$ and $x : T'$,
(e) if $S' \ni \texttt{type}_n A <: U$, then $T' \ni^{\cup} \texttt{type}_n A <: U$ and $T' \ni \texttt{type}_{n'} A = S''$, for some type S'', and
(f) $T <<:_\tau S''$.

In addition, we extend $<<:_\tau$ to be reflexively and transitively closed. We omit the subscript τ when it is clear from the context.

We now make precise the notion of type soundness that our correctness proof of TCA$^{expand\text{-}this}$ depends on. Informally stated, we assume that executions preserve types consistently with the extended subtype relation $<<:$, and that an execution does not get stuck because a method call invokes a method that is not a member of the receiver.

Assumption 1. Our soundness proof applies to FS_{alg} programs whose execution is type-sound in the following specific ways.

(1) If a term $s : S$ reduces to a variable $x : T$ in an execution τ, then $T <<:_\tau S$.
(2) If reduction of a valid FS_{alg} program reaches a method call term $\Sigma; x.a(\overline{y})$, then the set $\Sigma(x)$ of members of x contains a method M with name a, and if $x : T$, then this same method is also a member of the type T (that is, $T \ni M$).

Our proof also depends on some syntactic restrictions on the types that can be instantiated. Full Scala has these same restrictions. Although FS_{alg} does not make these restrictions syntactically, it is not obvious what it would mean in FS_{alg} to instantiate the types that we prohibit, and any FS_{alg} program instantiating them would immediately get stuck, because the lookup relation is not defined for them. The restrictions are specified by the following assumption:

Assumption 2. We assume that in every term that instantiates an object, which is of the form $\texttt{val}\, x = \texttt{new}\ T; t$, that the type T cannot be a singleton type of the form $p.\textbf{type}$ or an abstract type of the form $p.A$, where the member named A in p is a type declaration and the type member A is not an alias for some concrete type.

Definition 5.2. Let τ be a trace $\Sigma, t \rightarrow^* \Sigma', t'$ of the execution of an FS_{alg} program according to the operational semantics of Figure 11. We define calls(τ) and allocs(τ) to be the sets of methods called and types instantiated in the trace τ, respectively. Precisely, each use of the reduction rule RED-METHOD refers to a method label n in the precondition of the rule; calls(τ) is defined to be the set of all such method labels defined by all uses of the RED-METHOD reduction rule in the trace τ. Similarly, each use of the reduction rule RED-NEW refers to a type T in the conclusion of the rule, and the new object of this type is assigned to some variable x. The set allocs(τ) is defined to be the set of all such pairs x, T defined by all uses of the RED-NEW reduction rule in the trace τ.

With these definitions and assumptions, we can now proceed with the soundness proof of TCA$^{expand\text{-}this}$. We begin with a lemma that states, informally, that the *expand*$(\cdot)$ set soundly overapproximates the set of concrete types that any abstract type member $p.A$ can be instantiated with at runtime.

LEMMA 5.3. *Suppose that when* TCA$^{expand\text{-}this}$ *is applied to some initial term t, it computes the set of reachable methods R, the set of possibly instantiated types* $\hat{\Sigma}$*, and the expand relation expand*$(\cdot)$*. Let* τ *be an execution trace starting with the initial term t. Further suppose that s is a subterm of t that is reduced to x in the trace* τ*. If*

(1) $calls(\tau) \subseteq R$,
(2) $allocs(\tau) \subseteq \hat{\Sigma}$,
(3) $s : p.A$, *where* A *is an abstract type member in* p,
(4) $x : T$, *and*
(5) $p.\textbf{\textit{type}} \ni type_n A <: U$,

there exists a type S *such that* $T <: S$ *and* $S \in expand(type_n A)$.

PROOF. By Assumption 1, if p reduces to some variable y and if $y : T'$ and $p : U'$, then $T' <<: U'$. We first show that this reduction from p to y must actually occur within the execution trace τ: The reduction rule RED-METHOD looks up the members of x in the environment Σ, so the variable x must have been added to the environment in the trace τ by an application of RED-NEW. This rule looks up the members of the type of the variable being inserted, so the lookup relation must be applicable to $p.A$. In all of the types of the form $p.A$ appearing in the conclusions of the lookup relation in Figure 12, the type is of the form $y.A$. Therefore, a reduction from p to y must have occurred within the execution trace τ.

The lookup relations that apply to $y.A$ require y to be in the environment, so the instantiation of $\texttt{val}\, y = \texttt{new}\, T'; u$ must also occur in the trace τ. Therefore, by the assumptions of the lemma, $(y, T') \in \hat{\Sigma}$. By Definition 5.1, $T' \ni^{\cup} \texttt{type}_n A$, and $T' \ni \texttt{type}_{n'} A = S$ for some S such that $T <<: S$. Therefore, by analysis rule $\text{TCA}^{expand\text{-}this}_{\text{EXPAND-TYPE}}$, $S \in expand(\texttt{type}_n A)$. If S is a concrete type, then $T <<: S$ implies that $T <: S$ due to Definition 5.1, and we are done proving the lemma. If S is an abstract type $p'.A'$, then we can repeat the same argument as previously to show that a reduction of p' to some y' occurs in τ, that y' contains a type alias $\texttt{type}_{n''} A' = S'$, that $S' \in expand(\texttt{type}_{n'''} A')$, where $\texttt{type}_{n'''} A'$ is the declaration of A' in p', and that $T <<: S'$. Rule $\text{TCA}^{expand\text{-}this}_{\text{EXPAND-TRANS}}$ can then be applied since

(1) $p'.A' \in expand(\texttt{type}_n A)$,
(2) $p'.A' \ni \texttt{type}_{n'''} A' <: U'$, and
(3) $S' \in expand(\texttt{type}_{n'''} A')$,

it follows that $S' \in expand(\texttt{type}_n A)$. If S' is a concrete type, then $T <<: S'$ implies that $T <: S'$, and we are done proving the lemma. Otherwise, the same argument can be repeated to find additional types in the sequence $S, S', S'', \ldots$ until one of them is concrete. For each such type S^*, $S^* \in expand(\texttt{type}_n A)$, and $T <<: S^*$, so when S^* is concrete, then $T <: S^*$, and we are done proving the lemma. The type checking rules of FS_{alg} use a lock set $\mathcal{S}$ to explicitly check for and reject loops in type declarations, so they guarantee that for a valid FS_{alg} program, a concrete type S^* will eventually be encountered. □

Having shown the soundness of the $expand(\cdot)$ relation, we now prove a step lemma that states, informally, that if $\text{TCA}^{expand\text{-}this}$ soundly overapproximates a partial execution trace τ', then τ' can be extended with an additional execution step and $\text{TCA}^{expand\text{-}this}$ will still soundly overapproximate it.

LEMMA 5.4. *Suppose that when* $\text{TCA}^{expand\text{-}this}$ *is applied to some initial term* t, *it computes the set of reachable methods* R *and the set of possibly instantiated types* $\hat{\Sigma}$. *Let* τ' *be an execution trace from* Σ, t *to* Σ', t', *where* Σ *is the empty environment, and let* τ'' *be the extension of* τ' *with one additional execution step from* Σ', t' *to* Σ'', t''. *If*

(1) $calls(\tau') \subseteq R$, *and*
(2) $allocs(\tau') \subseteq \hat{\Sigma}$,

then

(*1*) $calls(\tau'') \subseteq R$, *and*
(*2*) $allocs(\tau'') \subseteq \hat{\Sigma}$.

PROOF. Assume that $\text{calls}(\tau') \subseteq R$ and $\text{allocs}(\tau') \subseteq \hat{\Sigma}$. There are three cases to consider depending on the reduction rule used in the transition $\Sigma', t' \rightarrow \Sigma'', t''$.

(1) *The reduction rule may be* RED-NEW. In this case, t' is of the form $\texttt{val}\, x = \texttt{new}\, T; t'''$, and $\text{allocs}(\tau'') = \text{allocs}(\tau') \cup \{T\}$. We must therefore prove that $(x, T) \in \hat{\Sigma}$. The conclusion of analysis rule $\text{TCA}_{\text{NEW}}^{\textit{expand-this}}$ is $(M \in R) \Rightarrow ((x, T) \in \hat{\Sigma})$, where M is the method containing the allocation site, that is, our term t'. M is in R because it is in $\text{calls}(\Sigma, t \rightarrow^* \Sigma', t')$. Therefore, by the conclusion of rule $\text{TCA}_{\text{NEW}}^{\textit{expand-this}}$, the pair (x, T) is in $\hat{\Sigma}$ as required.
(2) *The reduction rule may be* RED-METHOD. In this case, t' is a call site of the form $x.a(\overline{y})$ and $\text{calls}(\tau'') = \text{calls}(\tau') \cup \{M'\}$, where M' is the method that is invoked in the transition from Σ', t' to Σ'', t''. We must therefore prove that $M' \in R$. Since t' is of the form $x.a(\overline{y})$, the original program t must have contained a call site c of the form $s.a(\overline{t})$ such that s reduced to x and $\overline{t}$ reduced to $\overline{y}$ in the trace τ'. The overall plan is to show that either rule $\text{TCA}_{\text{CALL}}^{\textit{expand-this}}$ or $\text{TCA}_{\text{CALL-ABSTRACT}}^{\textit{expand-this}}$ applies to assert that $c \mapsto M'$, and then rule $\text{TCA}_{\text{REACHABLE}}^{\textit{expand-this}}$ concludes that $M' \in R$. We examine each of the preconditions of rules $\text{TCA}_{\text{CALL}}^{\textit{expand-this}}$ and $\text{TCA}_{\text{CALL-ABSTRACT}}^{\textit{expand-this}}$ in turn.
   (a) $M \in R$, where M is the method containing call site c.
   The execution trace τ' must contain a call to M, or else the execution would not be reducing the call site c, which is contained in M. By the premise of the lemma, $M \in R$.
   (b) $(x, T) \in \hat{\Sigma}$.
   In the rule RED-METHOD, x is looked up in the current runtime environment Σ'. Since the only reduction rule that adds bindings to the runtime environment is RED-NEW, such a reduction reducing an instantiation $\texttt{val}\, x = \texttt{new}\, T; t'''$, for some T and t''', must have occurred in the trace τ'. By the premise of the lemma, the pair (x, T) is therefore present in $\hat{\Sigma}$.
   (c) $T \ni M'$, $M' \equiv \texttt{def}_n a(\overline{y : S'}) : T'(= t')^?$.
   This is given by Assumption 1.
   (d) (for $\text{TCA}_{\text{CALL}}^{\textit{expand-this}}$) $T <: S$, where $s : S$.
   When the type S of s is a concrete type, this is given by Assumption 1. Otherwise, see case (f).
   (e) (for $\text{TCA}_{\text{CALL}}^{\textit{expand-this}}$) $(S \ni M \wedge s \equiv \varphi) \Rightarrow (T \ni M)$.
   The left-hand side of the implication ensures that this condition is relevant only when the receiver s of the call is the "this" variable φ, and in the case of nested traits, only when it is the φ of the innermost trait containing the method M that contains the call site. The latter is guaranteed by the predicate $S \ni M$, since only the innermost enclosing trait contains M as a member. In this case, the receiver φ has the same value that the receiver had when M itself was invoked. This value must contain M as a member because M was the method invoked. Therefore, the type T of the actual receiver x at the current call site c contains M as a member.
   (f) (for $\text{TCA}_{\text{ABSTRACT-CALL}}^{\textit{expand-this}}$) $T <: S$, $S \in \textit{expand}(\texttt{type}_n A)$, $p.\textbf{type} \ni \texttt{type}_n A <: U$, $s : p.A$.
   We need to ensure that these conditions are satisfied when the type $p.A$ of s is an abstract type. Lemma 5.3 applies to trace τ' and shows that the analysis infers a type $S \in \textit{expand}(\texttt{type}_n A)$, where $\texttt{type}_n A <: U$ is the declaration of A that is a member of $p.\textbf{type}$, such that $T <: S$.

Since all of the premises of either analysis rule $\text{TCA}^{\textit{expand-this}}_{\text{CALL}}$ or $\text{TCA}^{\textit{expand-this}}_{\text{CALL-ABSTRACT}}$ are satisfied, its conclusion must hold, so $c \mapsto M'$. By analysis rule $\text{TCA}^{\textit{expand-this}}_{\text{REACHABLE}}$, $M' \in R$, so the conclusion of the lemma is satisfied.

(3) *The reduction rule may be a rule other than* RED-NEW *and* RED-METHOD. In this case, $\text{calls}(\tau') = \text{calls}(\tau'')$ and $\text{allocs}(\tau') = \text{allocs}(\tau'')$, so the conclusion of the lemma is satisfied. □

Finally, we apply induction to the step lemma to prove that $\text{TCA}^{\textit{expand-this}}$ soundly overapproximates all execution traces.

THEOREM 5.5. *Let t be any initial FS_{alg} program, and let $\tau = \Sigma, t \rightarrow^* \Sigma', t'$ be its execution trace, where Σ is the empty environment. If* $\text{TCA}^{\textit{expand-this}}$ *computes the set of reachable methods R when applied to t, then $\text{calls}(\tau) \subseteq R$.*

PROOF. The proof is by induction on the length of the trace τ. The induction hypothesis is defined as follows: for a given trace τ, (1) $\text{calls}(\tau) \subseteq R$, and (2) $\text{allocs}(\tau) \subseteq \hat{\Sigma}$.

When τ is an empty trace, then the sets $\text{calls}(\tau)$ and $\text{allocs}(\tau)$ are empty, and therefore the conclusion is immediate. When τ is nonempty, let τ' be the subtrace of τ consisting of all except the last reduction step. If the induction hypothesis holds for τ', then by Lemma 5.4, the induction hypothesis also holds for τ. Therefore, by induction, the theorem holds for every execution trace τ. □

As a corollary, we also show that in addition to the sets R and $\hat{\Sigma}$, the computed call relation $\cdot \mapsto \cdot$ also soundly overapproximates runtime behaviour.

COROLLARY 5.6. *Let t be any initial FS_{alg} program, and let $\tau = \Sigma, t \rightarrow^* \Sigma', t'$ be its execution trace, where Σ is the empty environment. If* $\text{TCA}^{\textit{expand-this}}$ *computes the set of calls $\cdot \mapsto \cdot$ when applied to t, then each pair of call site c and method target n that occurs in a use of the reduction rule* RED-METHOD *in τ also occurs as a pair $c \mapsto n$ in the analysis result.*

PROOF. The corollary follows from the soundness of $\hat{\Sigma}$ and R (shown by Theorem 5.5) and following the same reasoning as in Case 2 (for rule RED-METHOD) of the proof of Lemma 5.4. □

6. IMPLEMENTATION

We implemented RA, $\text{TCA}^{\textit{names}}$, $\text{TCA}^{\textit{bounds}}$, $\text{TCA}^{\textit{expand}}$, and $\text{TCA}^{\textit{expand-this}}$ as a plugin for version 2.10.2 of the Scala compiler and tested the implementation on a suite of programs exhibiting a wide range of Scala features. To the best of our knowledge, our analyses soundly handle the entire Scala language under the assumption that all code to be analyzed is available and that reflection and dynamic code generation can be ignored. We also used the implementation of RTA in the WALA framework to construct call graphs from JVM bytecode. The implementation of all our analyses and scripts required to replicate our experiments are available at `http://karimali.ca/scalacg/`.

The analysis runs after the `uncurry` phase, which is the 12th of 30 phases in the Scala compiler. At this stage, most of the convenience features in Scala that are specified as syntactic sugar have been desugared. However, the compiler has not yet transformed the program to be closer to JVM bytecode and has not yet erased any significant type information. In particular, closures have been turned into function objects with `apply` methods, pattern matching has been desugared into explicit tests and comparisons, and implicit calls and parameters have been made explicit, so our analysis does not have to deal with these features explicitly.

Some Scala idioms (e.g., path-dependent types, structural types, singletons, and generics) make the subtype testing in Scala complicated [Odersky 2011, §3.5]. Fortunately, we can rely on the Scala compiler infrastructure to answer subtype queries. Two issues, however, require special handling in the implementation: super calls and incomplete programs.

6.1. Super Calls

Normally, when a method is called on some receiver object, the method is a member of that object. Super calls violate this general rule: a call on super invokes a method in a supertype of the receiver's type. This method is typically not a member of the receiver object because some other method overrides it.

At a call on super, the analysis must determine the method actually invoked. When the call site is in a class (not a trait), the call is resolved statically as in Java. When the call site is in a trait, however, the target method is selected using a dynamic dispatch mechanism depending on the runtime type of the receiver [Odersky 2011, §6.5]. Our analysis resolves such calls using a similar procedure as for ordinary dynamically dispatched calls. For each possible runtime type of the receiver, the specified procedure is followed to find the actual call target.

The $\text{TCA}^{\textit{expand-this}}$ analysis requires that within any method M, the this variable refers to an object of which M is a member. This premise is violated when M is invoked using a super call. To restore soundness, we blacklist the signatures of the targets of all reachable super calls. Within a method whose signature is blacklisted, we fall back to the $\text{TCA}^{\textit{expand}}$ analysis instead of $\text{TCA}^{\textit{expand-this}}$.

6.2. Incomplete Programs

Our analyses are defined for complete programs, but a practical implementation must deal with incomplete programs. A typical example of an incomplete program is a situation where user code calls unanalyzed libraries.

Our implementation analyzes Scala source files presented to the compiler, but not referenced classes provided only as bytecode such as the Scala and Java standard libraries. The analysis soundly analyzes call sites occurring in the provided Scala source files using a Scala analogue of the Separate Compilation Assumption [Ali and Lhoták 2012, 2013; Ali 2014], which asserts that unanalyzed "library" classes do not directly reference analyzed "application" classes. If application code passes the name of one of its classes to the library and the library instantiates it by reflection, then our analysis faces the same challenges as any Java analysis, and the same solutions would apply.

If the declaring class of the static target of a call site is available for analysis, then so are all its subtypes. In such cases, the analysis can soundly determine all possible actual call targets. On the other hand, if the declaring class of the static target of a call is in an unanalyzed class, it is impossible to determine all possible actual target methods, because some targets may be in unanalyzed code or in trait compositions that are only created in unanalyzed code. The implementation records the existence of such call sites but does not attempt to resolve them soundly. However, the methods invoked by such call sites, as well as those that occur in unanalyzed code, may call methods in analyzed code via call-backs. For soundness, the analysis must treat such target methods as reachable. This is achieved by considering a method reachable if it occurs in an instantiated type and if it overrides a method declared in unanalyzed code. This is sound because in both cases (a call whose static target is in unanalyzed code, or a call in unanalyzed code), the actual runtime target method must override the static target of the call.

Determining the method overriding relationship is more difficult than in Java. Two methods declared in two independent traits do not override each other unless these traits are composed in the instantiation of some object. Therefore, the overriding relation must be updated as new trait compositions are discovered.

7. EVALUATION

We evaluated our implementation on publicly available Scala programs covering a range of different application areas and programming styles.[6] Table I shows, for each program, the number of lines of Scala source code (excluding library code), classes, objects, traits, trait compositions, methods, closures, call sites, call sites on abstract types, and call sites on the variable this. ARGOT is a command-line argument parser for Scala. CASBAH is a Scala toolkit for the MongoDB[7] document database. ENSIME is an Emacs plugin that provides an enhanced Scala interactive mode, including a read-eval-print loop (REPL) and many features commonly found in IDEs, such as live error-checking, package/type browsing, and basic refactorings. FACTORIE is a toolkit for probabilistic modeling. It provides its users with a language for creating relational factor graphs, estimating parameters, and performing inference. FIMPP is an interpreter for an imperative, dynamically-typed language that supports integer arithmetic, console output, dynamically growing arrays, and subroutines. KIAMA is a library for language processing used to compile and execute several small languages. PHANTM is a tool that uses a flow-sensitive static analysis to detect type errors in PHP code [Kneuss et al. 2010]. SCALAP is Scala class-file decoder. SCALARIFORM is a code formatter written in Scala. SCALAXB is an XML data-binding tool for Scala. SCALISP is a LISP interpreter written in Scala. SEE is a simple engine for evaluating arithmetic expressions. SQUERYL is a Scala library that provides object-relational mapping for SQL databases. TICTACTOE is an implementation of the classic "tic-tac-toe" game with a text-based user interface. The programs SCALAP, SCALARIFORM, KIAMA, and SCALAXB are part of the DaCapo Scala Benchmarking project [Sewe et al. 2011]. We did not use the other DaCapo Scala benchmarks as they are not compatible with the version of Scala that we used for our experiments.

We ran all of our experiments on a machine with eight dual-core AMD Opteron 1.4 GHz CPUs (running in 64-bit mode) and capped the available memory for the experiments to 16GB of RAM.

Assumptions. For all our experiments, we consider main methods and constructors of Scala modules to be entry points of the call graph. For benchmarks that represent libraries (e.g., CASBAH, KIAMA, and SQUERYL), we also include the test suites that are offered by the library designers as part of the analyzed program. This allows us to properly evaluate those benchmarks that represent libraries, as they normally do not have main methods.

7.1. Research Questions

Our evaluation aims to answer the following research questions.

—*RQ1.* How precise are call graphs constructed from JVM bytecode produced by the Scala compiler compared to call graphs constructed from Scala source code using our new algorithms?

[6]The benchmark source code is available from http://github.com/bmc/argot, http://github.com/mongodb/casbah, http://github.com/aemoncannon/ensime, http://github.com/factorie/factorie, http://github.com/KarolS/fimpp, http://code.google.com/p/kiama, http://github.com/colder/phantm, http://scala-lang.org/, http://github.com/mdr/scalariform, http://github.com/eed3si9n/scalaxb, http://github.com/Mononofu/Scalisp, http://scee.sourceforge.net, http://github.com/max-l/Squeryl, and http://github.com/nickknw/arbitrarily-sized-tic-tac-toe.

[7]http://www.mongodb.org.

Table I. Various Characteristics of Our Benchmark Programs

	LOC	# classes	# objects	# traits	# trait compositions	# methods	# closures	# call sites	# call sites on abstract types	# call sites on this
ARGOT	1,074	18	4	6	185	485	168	2,543	2	276
CASBAH	1,944	39	47	11	152	1,309	130	4,864	8	912
ENSIME	7,832	223	172	36	1,051	4,878	532	19,555	23	3,195
FACTORIE	35,428	1,173	816	420	5,652	22,292	3,892	99,933	1,380	16,876
FIMPP	1,089	42	53	5	673	2,060	549	5,880	4	1,159
KIAMA	17,914	801	664	162	5,340	19,172	3,963	69,352	401	16,256
PHANTM	9,319	317	358	13	1,491	7,208	561	36,276	15	6,643
SCALAP	2,348	91	83	35	661	2,746	519	7,888	24	2,059
SCALARIFORM	7,750	147	191	46	1,362	5,776	966	23,142	26	7,012
SCALAXB	10,290	324	259	222	3,047	10,503	2,204	47,382	35	7,305
SCALISP	795	20	14	0	125	428	115	2,313	23	293
SEE	4,311	130	151	17	415	2,280	262	9,566	11	1,449
SQUERYL	7,432	255	55	110	1,043	3,793	826	13,585	173	2,540
TICTACTOE	247	2	7	0	32	112	24	603	0	41

—*RQ2.* What is the impact on call graph precision of adopting subtype-based call resolution instead of name-based call resolution?
—*RQ3.* What is the impact on call graph precision of determining the set of concrete types that may be bound to abstract type members instead of using a bounds-based approximation?
—*RQ4.* What is the impact of the special treatment of calls on `this`?
—*RQ5.* How does the running time of the analyses compare?
—*RQ6.* For how many call sites can the algorithms find a single outgoing edge?

7.2. Results

Table II summarizes the precision of the call graphs computed by our analyses. For each benchmark and analysis combination, the table shows the number of reachable methods and call edges in the call graph. All call graphs presented in this section include only the analyzed code of the benchmark itself, excluding any library code. For RTA^{wala}, such "summarized call graphs" were obtained by collapsing the parts of the call graph in the library into a single node.

RQ1. To answer this question, we compare the call graphs from the TCA^{bounds} and RTA^{wala} analyses. The call graphs constructed from bytecode have on average 1.7x as many reachable methods and 4.5x as many call edges as the call graphs constructed by analyzing Scala source. In other words, analyzing generated bytecode incurs a very large loss in precision.

Investigating further, we found that the most significant cause of precision loss is due to `apply` methods, which are generated from closures. These account for, on average, 25% of the spurious call edges computed by RTA^{wala} but not by TCA^{bounds}. The second-most significant cause of precision loss are `toString` methods, which account for, on average, 13% of the spurious call edges.

The ENSIME program is an interesting special case because it uses Scala constructs that are translated into code that uses reflection (see Section 3.5). As a result, the RTA^{wala} analysis makes conservative approximations that cause the call graph to become extremely large and imprecise. The summarized call graph computed by RTA^{wala} shown in Table II has 4,525 nodes and 61,803 edges. However, the size of the call graph originally computed by RTA^{wala} (before summarizing the library code) has 78,901 nodes

Table II. Number of Nodes and Edges in the Summarized Version of Call Graphs Computed using the RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, $TCA^{expand\text{-}this}$, and RTA^{wala}

		RA	TCA^{names}	TCA^{bounds}	TCA^{expand}	$TCA^{expand\text{-}this}$	RTA^{wala}
ARGOT	nodes	265	184	161	161	161	236
	edges	3,516	1,538	442	442	440	648
CASBAH	nodes	772	592	398	398	398	581
	edges	15,159	10,540	1,271	1,271	1,269	2,275
ENSIME	nodes	3,491	3,018	2,967	2,966	2,965	4,525
	edges	191,435	150,974	8,025	8,023	8,017	61,803
FACTORIE	nodes	16,373	11,589	9,656	9,530	9,493	19,889
	edges	6,187,646	3,835,924	63,768	45,790	45,414	370,668
FIMPP	nodes	870	773	771	771	771	1,381
	edges	12,716	10,900	2,404	2,404	2,404	8,327
KIAMA	nodes	11,959	8,684	7,609	7,600	7,200	13,597
	edges	1,555,533	845,120	35,288	34,041	32,475	609,255
PHANTM	nodes	5,945	5,207	4,798	4,587	4,587	5,157
	edges	376,065	296,252	14,727	13,899	13,870	213,264
SCALAP	nodes	1,650	1,449	1,314	1,306	1,306	1,856
	edges	48,089	33,744	3,563	3,551	3,551	19,016
SCALARIFORM	nodes	4,277	3,548	2,852	2,852	2,852	4,629
	edges	210,352	157,589	9,117	9,042	9,042	81,512
SCALAXB	nodes	6,795	2,263	1,196	1,196	1,196	3,866
	edges	1,832,473	322,499	5,819	5,819	5,818	48,966
SCALISP	nodes	283	196	186	186	186	307
	edges	3,807	2,380	526	526	526	908
SEE	nodes	1,869	1,711	1,645	1,572	1,572	2,016
	edges	77,303	63,706	8,349	7,466	7,418	14,520
SQUERYL	nodes	2,484	1,488	408	408	408	1,507
	edges	91,342	46,160	1,677	1,677	1,676	8,669
TICTACTOE	nodes	79	78	78	78	78	112
	edges	524	523	170	170	170	327

and 7,835,170 edges. We experimentally confirmed that nearly half of these edges are in parts of the libraries related to the reflection API. This further reaffirms that a bytecode-based approach to call graph construction is highly problematic.

RQ2. To answer this question, we compare TCA^{names} and TCA^{bounds} and find that name-based analysis incurs a very significant precision loss: The call graphs generated by TCA^{names} have, on average, 12.34x as many call edges as those generated by TCA^{bounds}. Investigating further, we find that, on average, 66% of the spurious call edges computed by the name-based analysis were to apply methods, which are used to implement closures.

RQ3. To answer this question, we compare TCA^{bounds} and TCA^{expand}. On the smaller benchmark programs that make little use of abstract types, the two produce identical results. Since FACTORIE, KIAMA, PHANTM, and SEE contain some call sites on receivers with abstract types, TCA^{expand} computes more precise call graphs for them. For SCALAXB,

```
211 trait org.kiama.output.ParenPrettyPrinter {
212   def toParenDoc = { this.bracket }
213   def bracket = { /* calls method noparens() that instantiates type Postfix */ }
214 }
215
216 object A4 extends ... with L0.source.PrettyPrinter with ... {
217   // toParenDoc() inherited from L0.source.PrettyPrinter
218   // no definition of bracket()
219  this.toParenDoc()
220 }
221
222 object A2b extends ... with L1.source.PrettyPrinter with ... {
223    // no definition of toParenDoc()
224    // bracket() inherited from L1.source.PrettyPrinter
225   this.toParenDoc()
226 }
```

Fig. 23. A Scala program for which TCA^{expand} finds new instantiated types in methods found unreachable by TCA$^{expand\text{-}this}$ (taken from the KIAMA program).

SCALISP, and SQUERYL, call graph precision is not improved despite the presence of abstract types because the call sites on abstract receivers occur in unreachable code.

RQ4. To answer the fourth research question, we compare the TCA^{expand} and TCA$^{expand\text{-}this}$ analyses. In general, we found that the precision benefit of the special handling of this calls is small and limited to specific programs. In particular, we found that the number of call edges is reduced by 5% on KIAMA and by 1% on SEE but that there is no significant difference on the other benchmarks. The situation for KIAMA is interesting in that TCA^{expand} finds 3.7% more instantiated types than TCA$^{expand\text{-}this}$. Those types are instantiated in methods identified as unreachable by TCA$^{expand\text{-}this}$.

Figure 23 shows an excerpt from KIAMA that illustrates why TCA$^{expand\text{-}this}$ finds fewer instantiated types. The figure shows a trait org.kiama.output.ParenPrettyPrinter that defines a method toParenDoc, which contains a call this.bracket. This method, in turn, calls another method that instantiates type PostFix. org.kiama.output.ParenPrettyPrinter also defines a method bracket(), so the TCA^{expand} analysis would create an edge from the call on line 212 to method org.kiama.output.ParenPrettyPrinter.bracket(), thus causing type PostFix to be instantiated. The figure also shows two objects, A4 and A2b, that mix in traits L0.source.PrettyPrinter and L1.source.PrettyPrinter, respectively, which inherit from org.kiama.output.ParenPrettyPrinter. Since A4 inherits a definition of toParenDoc() from L0.source.PrettyPrinter, the TCA$^{expand\text{-}this}$ analysis concludes that the type of this on line 212 cannot be A4. The TCA$^{expand\text{-}this}$ analysis also rules out type A2b as the type of this on line 212 because it inherits a different definition of bracket from L1.source.PrettyPrinter. Consequently, the TCA$^{expand\text{-}this}$ analysis finds that method org.kiama.output.ParenPrettyPrinter.bracket() is unreachable and that type PostFix is not instantiated.

The two most common reasons why the more precise rule TCA$^{expand\text{-}this}_{\text{THIS-CALL}}$ may fail to rule out a given call graph edge are that the caller M is inherited into the runtime receiver type C, so the call can occur, or that the caller M can be called through super, so using the rule would be unsound, as explained in Section 6.1. Across all the benchmark programs, the rule failed to eliminate a call edge at 81% of call sites on this due to the caller M being inherited into C, and at 17% of call sites on this due to the caller M being called through super.

RQ5. The running times of the analyses are presented in Table III. For comparison, the last column of the table also shows the time required to compile each benchmark

Table III. Time (in Seconds) Taken by RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, TCA$^{expand\text{-}this}$, and RTA^{wala} to Compute the Call Graphs

	RA	TCA^{names}	TCA^{bounds}	TCA^{expand}	TCA$^{expand\text{-}this}$	RTA^{wala}	scalac
ARGOT	3.8	3.4	3.5	3.7	4	11.7	25.7
CASBAH	3.8	3.4	3.9	4.4	4.2	16.4	35.3
ENSIME	31.5	25.2	28.9	29	28.3	528.3	61.5
FACTORIE	873.9	523.8	317.2	419.2	405.2	594.8	126.7
FIMPP	5.2	4.8	7.8	8.1	8.1	14.3	37
KIAMA	279.7	88	133	138.2	123	65.9	107.6
PHANTM	55.1	42.9	56.1	57	58.7	27.8	70.2
SCALAP	8.2	7.1	7.2	7.9	8.1	15.3	42.2
SCALARIFORM	25.8	19.4	17.6	20.2	17.4	21.6	61.4
SCALAXB	110.3	16.4	12.1	12.6	12.7	21.9	87.1
SCALISP	3.1	3	3.1	3.4	3.4	12.8	26.1
SEE	7.2	6.4	8.6	9	9.3	13.8	40.4
SQUERYL	20.6	11.6	6.4	6.9	7	20	59.4
TICTACTOE	1.6	1.8	1.8	2	2	10	16.4

using the unmodified Scala compiler. Although our implementation has not been heavily tuned for performance, across all the benchmarks, the analysis times of TCA$^{expand\text{-}this}$ are an average 28% of the compilation time taken by `scalac`. The high imprecision of the RA analysis generally makes it significantly slower than the other, more complicated but more precise analyses. The TCA^{names} analysis is sometimes significantly faster and sometimes significantly slower than the TCA^{bounds} analysis, since it avoids many expensive subtype tests, but is significantly less precise. The TCA^{expand} and TCA$^{expand\text{-}this}$ analyses have generally similar execution times as the TCA^{bounds} analysis because abstract types and this calls are a relatively small fraction of all call sites in the benchmark programs.

The long running time of nearly 500 seconds of RTA^{wala} on ENSIME is because the computed call graph becomes extremely large (see discussion of RQ1).

RQ6. Certain applications of call graphs require call sites to have a unique outgoing edge. For example, whole-program optimization tools [Tip et al. 2002] may inline such "monomorphic" call sites. It is therefore useful to measure the ability of the different algorithms to resolve call sites to a unique target method. In principle, the RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, and TCA$^{expand\text{-}this}$ algorithms are progressively more precise, so we could see fewer nodes and edges as we move to the next algorithm in this sequence. Therefore, it is particularly interesting to determine if an increase in precision occurs in practice.

Tables IV–VII show, for each benchmark program, the number of monomorphic and polymorphic call sites, as determined by the analyses RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, and TCA$^{expand\text{-}this}$, respectively. For each pair of algorithms (X, Y), where X is the less precise algorithm and Y is the more precise algorithm, these tables show:

—for call sites that are identified as monomorphic by X, whether they are (i) identified as unreachable by Y, or (ii) identified as monomorphic by Y, and
—for call sites that are identified as polymorphic by X, whether they are (i) identified as unreachable by Y, (ii) identified as monomorphic by Y, or (iii) identified as polymorphic by Y.

Table IV. Number of Monomorphic and Polymorphic Reachable Call Sites in the Summarized Version of Call Graphs Computed using RA and How Many of Them Became Unreachable, Monomorphic, or Polymorphic in TCA^{names}

			TCA^{names}		
	RA		Unreachable	Mono	Poly
ARGOT	**Mono**	1,200	425	775	–
	Poly	1,296	536	4	756
CASBAH	**Mono**	1,898	193	1,705	–
	Poly	2,228	279	48	1,901
ENSIME	**Mono**	10,901	362	10,539	–
	Poly	8,433	368	53	8,012
FACTORIE	**Mono**	35,980	5,301	30,679	–
	Poly	61,737	8,689	683	52,365
FIMPP	**Mono**	4,058	56	4,002	–
	Poly	1,636	7	11	1,618
KIAMA	**Mono**	40,974	13,460	27,514	–
	Poly	27,869	9,833	1,244	16,792
PHANTM	**Mono**	17,500	606	16,894	–
	Poly	18,611	554	98	17,959
SCALAP	**Mono**	4,198	335	3,863	–
	Poly	3,218	207	12	2,999
SCALARIFORM	**Mono**	12,001	767	11,234	–
	Poly	10,727	2,098	859	7,770
SCALAXB	**Mono**	22,170	9,320	12,850	–
	Poly	24,809	11,891	86	12,832
SCALISP	**Mono**	1,163	126	1,037	–
	Poly	1,106	99	165	842
SEE	**Mono**	5,327	97	5,230	–
	Poly	4,126	114	124	3,888
SQUERYL	**Mono**	6,453	1,709	4,744	–
	Poly	6,369	1,912	174	4,283
TICTACTOE	**Mono**	330	1	329	–
	Poly	204	0	0	204

Given this classification, the results can be summarized as follows.

—Table IV compares how calls are resolved by the RA and the TCA^{names} analyses. From the results shown in the table, it is clear that the TCA^{names} algorithm can be considerably more precise. For example, for the KIAMA program, we observe that of the 40,974 call sites classified as monomorphic by RA, 13,460 are determined to be unreachable by the more precise TCA^{names} analysis. Moreover, of the 27,689 polymorphic call sites that RA finds in KIAMA, 9,833 are found to be unreachable by TCA^{names} and 1,244 are classified as monomorphic by TCA^{names}, thus reducing the number of polymorphic call sites to 16,792. From these results, it is clear that taking trait compositions into consideration results in significantly more precise call graphs than what can be achieved using RA's naive name-based resolution.
—Table V compares how calls are resolved by the TCA^{names} and the TCA^{bounds} analyses. These results show that the use of the static types of receivers makes a significant difference in precision. For example, for FACTORIE, the TCA^{names} algorithm constructs

Table V. Number of Monomorphic and Polymorphic Reachable Call Sites in the Summarized Version of Call Graphs Computed using TCA^{names} and How Many of Them Became Unreachable, Monomorphic, or Polymorphic in TCA^{bounds}

		TCA^{names}	TCA^{bounds}		
			Unreachable	Mono	Poly
ARGOT	**Mono**	779	34	745	–
	Poly	756	39	682	35
CASBAH	**Mono**	1,753	157	1,596	–
	Poly	1,901	291	1,592	18
ENSIME	**Mono**	10,592	36	10,556	–
	Poly	8,012	58	7,495	459
FACTORIE	**Mono**	31,362	1,321	30,041	–
	Poly	52,365	3,498	41,941	6,926
FIMPP	**Mono**	4,013	0	4,013	–
	Poly	1,618	0	1,467	151
KIAMA	**Mono**	28,758	787	27,971	–
	Poly	16,792	762	14,970	1,060
PHANTM	**Mono**	16,992	89	16,903	–
	Poly	17,959	77	16,153	1,729
SCALAP	**Mono**	3,875	425	3,450	–
	Poly	2,999	327	2,471	201
SCALARIFORM	**Mono**	12,093	1,641	10,452	–
	Poly	7,770	1,592	5,985	193
SCALAXB	**Mono**	12,936	2,903	10,033	–
	Poly	12,832	5,273	7,014	545
SCALISP	**Mono**	1,202	42	1,160	–
	Poly	842	30	750	62
SEE	**Mono**	5,354	34	5,320	–
	Poly	3,888	56	2,933	899
SQUERYL	**Mono**	4,922	2,505	2,417	–
	Poly	4,283	2,763	1,443	77
TICTACTOE	**Mono**	329	0	329	–
	Poly	204	0	187	17

a call graph with 31,362 monomorphic call sites and 52,365 polymorphic call sites. The TCA^{bounds} algorithm classifies 1,321 of the 31,362 monomorphic call sites as unreachable, and of the 52,365 call sites reported as polymorphic by TCA^{names}, it finds that 3,498 are unreachable, 41,941 are monomorphic, and that only 6,926 polymorphic call sites remain. Clearly, using the static type of the receiver can make a significant precision difference in practice.

—Table VI compares how calls are resolved by the TCA^{bounds} and the TCA^{expand} analyses. From these results, it is clear that on several smaller programs that do not make significant use of abstract types (ARGOT, CASBAH, SCALARIFORM, SCALAXB, SCALISP, SQUERYL, TICTACTOE), the two algorithms compute exactly the same result. However, on larger programs that feature complex use of abstract type members, TCA^{expand}'s more precise analysis of how such types are instantiated makes a difference. For example, on FACTORIE, TCA^{bounds} computes a call graph with 71,892 monomorphic call sites and 6,926 polymorphic call sites. Of these 71,892 monomorphic call sites, the more precise TCA^{expand} algorithm finds 414 to be unreachable. Furthermore, of the 6,926 found to be polymorphic by TCA^{bounds}, 197 are classified as

Table VI. Number of Monomorphic and Polymorphic Reachable Call Sites in the Summarized Version of Call Graphs Computed using TCA^{bounds} and How Many of Them Became Unreachable, Monomorphic, or Polymorphic in TCA^{expand}

		TCA^{bounds}	TCA^{expand}		
			Unreachable	Mono	Poly
ARGOT	**Mono**	1,427	0	1,427	–
	Poly	35	0	0	35
CASBAH	**Mono**	3,188	0	3,188	–
	Poly	18	0	0	18
ENSIME	**Mono**	18,051	4	18,047	–
	Poly	459	0	0	459
FACTORIE	**Mono**	71,982	414	71,568	–
	Poly	6,926	197	101	6,628
FIMPP	**Mono**	5,480	0	5,480	–
	Poly	151	0	0	151
KIAMA	**Mono**	42,941	118	42,823	–
	Poly	1,060	3	13	1,044
PHANTM	**Mono**	33,056	328	32,728	–
	Poly	1,729	0	0	1,729
SCALAP	**Mono**	5,921	31	5,890	–
	Poly	201	0	0	201
SCALARIFORM	**Mono**	16,437	0	16,437	–
	Poly	193	0	1	192
SCALAXB	**Mono**	17,047	0	17,047	–
	Poly	545	0	0	545
SCALISP	**Mono**	1,910	0	1,910	–
	Poly	62	0	0	62
SEE	**Mono**	8,253	189	8,064	–
	Poly	899	89	3	807
SQUERYL	**Mono**	3,860	0	3,860	–
	Poly	77	0	0	77
TICTACTOE	**Mono**	516	0	516	–
	Poly	17	0	0	17

unreachable by TCA^{expand}, and 101 are classified as monomorphic. In summary, the more precise treatment of abstract type members by TCA^{expand} has limited impact on precision, on larger programs that feature more complex usage of abstract type members.

—Table VII compares how calls are resolved by the TCA^{expand} and the TCA$^{expand\text{-}this}$ analyses. From these results, it is clear that the special handling of calls on this makes no difference on small programs that do not have complex inheritance hierarchies. However, it is interesting to see that on our two largest subject programs, there is a noticable effect. For example, the TCA^{expand} algorithm found 42,836 monomorphic call sites on KIAMA of which 1,709 were classified as unreachable by the more precise TCA$^{expand\text{-}this}$ algorith. Furthermore, of the 1,044 call sites found to be polymorphic by TCA^{expand}, 47 were classified as unreachable by TCA$^{expand\text{-}this}$, whereas an additional 51 were classified as polymorphic. From these results, we conclude that although the impact of the special handling of this is limited, it can have a noticable effect as programs become larger and more complex.

Table VII. Number of Monomorphic and Polymorphic Reachable Call Sites in the Summarized Version of Call Graphs Computed using TCA^{expand} and How Many of Them Became Unreachable, Monomorphic, or Polymorphic in TCA$^{expand\text{-}this}$

		TCA^{expand}	TCA$^{expand\text{-}this}$		
			Unreachable	Mono	Poly
ARGOT	**Mono**	1,427	0	1,427	–
	Poly	35	0	1	34
CASBAH	**Mono**	3,188	2	3,186	–
	Poly	18	0	0	18
ENSIME	**Mono**	18,047	0	18,047	–
	Poly	459	0	1	458
FACTORIE	**Mono**	71,669	162	71,507	–
	Poly	6,628	4	43	6,581
FIMPP	**Mono**	5,480	0	5,480	–
	Poly	151	0	0	151
KIAMA	**Mono**	42,836	1,709	41,127	–
	Poly	1,044	47	51	946
PHANTM	**Mono**	32,728	0	32,728	–
	Poly	1,729	0	136	1,593
SCALAP	**Mono**	5,890	0	5,890	–
	Poly	201	0	0	201
SCALARIFORM	**Mono**	16,438	0	16,438	–
	Poly	192	0	0	192
SCALAXB	**Mono**	17,047	0	17,047	–
	Poly	545	0	0	545
SCALISP	**Mono**	1,910	0	1,910	–
	Poly	62	0	0	62
SEE	**Mono**	8,067	0	8,067	–
	Poly	807	0	0	807
SQUERYL	**Mono**	3,860	1	3,859	–
	Poly	77	0	0	77
TICTACTOE	**Mono**	516	0	516	–
	Poly	17	0	0	17

In summary, we conclude that for finding monomorphic call sites, the use of trait compositions in TCA^{names} is vastly superior to the naive name-based resolution in RA. Similarly, the use of bounds to approximate the behavior of calls on abstract receivers in TCA^{bounds} is a significant improvement over the naive way in which TCA^{names} resolves such calls. The impact of the more precise modeling of abstract receiver types in TCA^{expand} and of calls on this in TCA$^{expand\text{-}this}$ has more limited impact. However, it is encouraging to see that the impact of these optimizations appears to become more pronounced as applications become larger and more complex.

7.3. Discussion

The subject programs that we used for the evaluation of our work cover a range of different programming styles. This includes the use of traditional object-oriented constructs as well as some programs that rely heavily on closures as is promoted by the functional programming style that Scala encourages. There are some additional Scala features that are not exercised by our subject programs such as actors and concurrency, but these features do not pose soundness problems for our type-based algorithms since

our algorithms are flow-insensitive and do not track the interleaving of threads. That said, given that programs that use actors tend to make heavy use of closures, it would be advisable to use one of our more precise algorithms (at least TCA^{bounds}) in such cases in order to avoid a detrimental loss of precision while resolving calls to apply methods discussed previously.

The algorithms presented in this article are all type-based. The design and evaluation of flow-based and context-sensitive algorithms is a topic for future work. In the context of Java, it has been shown that it is crucial for the precision of flow-based algorithms to also enforce statically declared types [Lhoták and Hendren 2003]. A flow-based algorithm for Scala could do this by building on the techniques that we have proposed here. It will be particularly interesting to see whether such algorithms provide greater benefits for Scala than they do for Java, given the pervasive use of functional programming idioms in Scala.

8. CONCLUSIONS

We presented a family of low-cost algorithms for constructing call graphs of Scala programs in the spirit of Name-Based Resolution (RA) [Srivastava 1992], Class Hierarchy Analysis (CHA) [Dean et al. 1995], and Rapid Type Analysis (RTA) [Bacon and Sweeney 1996]. Our algorithms consider how traits are combined in a Scala program to improve precision and handle the full Scala language, including features such as abstract type members, closures, and path-dependent types. Furthermore, we propose a mechanism for resolving calls on the this reference more precisely, by considering overriding definitions of the method containing the call site.

We implemented the algorithms in the context of the Scala compiler and compared their precision and cost on a collection of Scala programs. We found that TCA^{names} is significantly more precise than RA, indicating that maintaining a set of instantiated trait combinations greatly improves precision. Furthermore, TCA^{bounds} is significantly more precise than TCA^{names}, indicating that subtyping-based call resolution is superior to name-based call resolution. The improvements of TCA^{expand} over TCA^{bounds} occur on a few larger subjects that make nontrivial use of abstract type members and type parameters. Similarly, TCA$^{expand\text{-}this}$ only did significantly better than TCA^{expand} on programs that make nontrivial use of subtyping and method overriding.

Prior to our work, if one needed a call graph for a Scala program, the only available method was to analyze the JVM bytecodes produced by the Scala compiler. Since significant type information is lost during the compilation process, RTA call graphs constructed from the JVM bytecodes can be expected to be much less precise than the call graphs constructed using our new algorithms, as confirmed by our experimental results.

While our research has focused on Scala, several aspects of the work are broadly applicable to other statically typed object-oriented languages. In particular, the special handling of calls on this can be integrated with existing algorithms such as CHA and RTA for languages such as Java, C#, and C++.

ACKNOWLEDGMENTS

We are grateful to Max Schäfer and the reviewers of previous versions of this article for many invaluable comments and suggestions, and to Rob Schluntz for assistance with testing.

REFERENCES

Ole Agesen. 1994. Constraint-based type inference and parametric polymorphism. In *Proceedings of the 1st International Static Analysis Symposium (SAS)*. 78–100.

Karim Ali. 2014. *The Separate Compilation Assumption*. Ph.D. Dissertation. University of Waterloo, Canada.

Karim Ali and Ondřej Lhoták. 2012. Application-only call graph construction. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 688–712.

Karim Ali and Ondrej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 378–400.

Karim Ali, Marianna Rapoport, Ondrej Lhoták, Julian Dolby, and Frank Tip. 2014. Constructing call graphs of Scala programs. In *Proceedings of the 28th European Conference Object-Oriented Programming*. 54–79. DOI:http://dx.doi.org/10.1007/978-3-662-44202-9_3

David Francis Bacon. 1997. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. Ph.D. Dissertation. University of California, Berkeley.

David F. Bacon and Peter F. Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 324–341.

Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 243–262.

Vincent Cremet, François Garillot Sergueï Lenglet, and Martin Odersky. 2006. A core calculus for Scala type checking. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science (MFCS)*. 1–23.

Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 77–101.

Greg DeFouw, David Grove, and Craig Chambers. 1998. Fast interprocedural class analysis. In *Proceedings of the ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL)*. 222–236.

Gilles Dubochet and Martin Odersky. 2009. Compiling structural types on the JVM: A comparison of reflective and generative techniques from Scala's perspective. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS'09)*. ACM, New York, NY, 34–41. DOI:http://dx.doi.org/10.1145/1565824.1565829

David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.* 23, 6 (2001), 685–746.

Nevin Heintze. 1994. Set-based analysis of ML programs. In *Proceedings of the ACM Conference on LISP and Functional Programming*. 306–317.

Nevin Heintze and Olivier Tardieu. 2001. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 254–263.

Fritz Henglein. 1992. Dynamic typing. In *Proceedings of the 4th European Symposium on Programming (ESOP)*. 233–253.

IBM. 2013. T.J. Watson Libraries for analysis WALA. http://wala.sourceforge.net/. (Last accessed April 2013).

Etienne Kneuss, Philippe Suter, and Viktor Kuncak. 2010. Phantm: PHP analyzer for type mismatch. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT FSE)*. 373–374.

Ondřej Lhoták and Laurie J. Hendren. 2003. Scaling Java points-to analysis using SPARK. In *Proceedings of the International Conference on Compiler Construction (CC)*. 153–169.

Ondřej Lhoták and Laurie J. Hendren. 2006. Context-sensitive points-to analysis: Is it worth it?. In *Proceedings of the International Conference on Compiler Construction (CC)*. 47–64.

Martin Odersky. 2011. *The Scala Language Specification Version 2.9*. Technical Report. EPFL.

Martin Odersky, Lex Spoon, and Bill Venners. 2012. *Programming in Scala* (2nd ed.). Artima Press.

B. G. Ryder. 1979. Constructing the call graph of a program. *IEEE Trans. Softw. Eng.* 5, 3 (1979), 216–226. DOI:http://dx.doi.org/10.1109/TSE.1979.234183

Olivier Sallenave and Roland Ducourneau. 2012. Lightweight generics in embedded systems through static analysis. In *Proceedings of the SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. 11–20.

Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da capo con scala: Design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. 657–676.

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph.D. Dissertation. CMU.

Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. 435–458.

A. Srivastava. 1992. Unreachable procedures in object oriented programming. *ACM Lett. Program. Lang. Syst.* 1, 4 (December 1992), 355–364.

Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. Practical virtual method call resolution for Java. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. 264–280.

Frank Tip and Jens Palsberg. 2000. Scalable propagation-based call graph construction algorithms. In *Proceedings of the ACM International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*. 281–293.

Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. 2002. Practical extraction techniques for Java. *ACM Trans. Program. Lang. Syst.* 24, 6 (2002), 625–666.

Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible?. In *Proceedings of the International Conference on Compiler Construction (CC)*. 18–34. http://portal.acm.org/citation.cfm?id=647476.727758.

Received March 2015; revised July 2015; accepted September 2015