An Evaluation of Call Graph Construction Algorithms for Scala

Karim Ali[1], Marianna Rapoport[1], Ondřej Lhoták[1], Julian Dolby[2], and
Frank Tip[1]

[1] University of Waterloo
{ karim,mrapoport,olhotak,ftip }@uwaterloo.ca
[2] IBM T.J. Watson Research Center
dolby@us.ibm.com

Abstract. As Scala gains popularity, there is growing interest in programming tools for it. Such tools often require call graphs. However, call graph construction algorithms in the literature do not handle Scala features, such as traits and abstract type members. Applying existing call graph construction algorithms to the JVM bytecodes generated by the Scala compiler produces very imprecise results due to type information being lost during compilation. We adapt existing call graph construction algorithms, Name-Based Resolution (RA) and Rapid Type Analysis (RTA), for Scala, and present a formalization based on Featherweight Scala. We evaluate our algorithms on a collection of Scala programs. Our results show that careful handling of complex Scala constructs greatly helps precision and that our most precise analysis generates call graphs with 1.1-3.7 times fewer nodes and 1.5-18.7 times fewer edges than a bytecode-based RTA analysis.

1 Introduction

As Scala [19] gains popularity, the need grows for program analysis tools for it that automate tasks such as refactoring, bug-finding, verification, model checking, 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 (see, e.g., [10,20]); algorithms vary primarily in how they handle indirect function calls. Several of Scala's more intricate features such as traits, abstract type members, and closures affect method call behavior; however, to our knowledge, no call graph construction algorithms for Scala have yet been proposed or evaluated.

One could construct call graphs of Scala programs by compiling them to JVM bytecode, and then using existing bytecode-based program analysis frameworks such as WALA [14] or SOOT [27] on 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, which challenges sound static analysis.

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) [25], Class Hierarchy Analysis (CHA) [8], and Rapid Type Analysis (RTA) [5]. 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 algorithms. We formally define our most precise algorithm for FS_{alg}, the "Featherweight Scala" subset of Scala that was previously defined by Cremet et al. [7]. We prove correctness of our algorithm by demonstrating that for each execution of a method call in the operational semantics, a corresponding edge exists in the constructed call graph[3].

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 the 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 found that call graphs constructed from the JVM bytecodes using the RTA algorithm are significantly less precise than those constructed using our source-based algorithms, of significant type information that is lost due to the transformations and optimizations performed by the Scala compiler.

In summary, this paper makes the following contributions:

1. We present variations on the RA [25] and RTA [5] 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.

In addition, we formalize our most precise algorithm and prove its correctness (in the supplemental materials provided with this submission).

The remainder of this paper 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

---

[3] We only present an informal definition of the algorithms in this paper. The full formalization and proofs can be found in the supplemental materials that are available with this submission.

associated with constructing call graphs of Scala programs. Section 4 presents our algorithms. Section 5 presents the implementation in the context of the Scala compiler. An evaluation of our algorithms is presented in Section 6. Lastly, Section 7 concludes and briefly discusses directions for future work.

2   Background

Algorithms for call graph construction [10] have been studied extensively in the context of object-oriented programming languages such as Java [9,16], C++ [5] and Self [1], of functional programming languages such as Scheme [23] and ML [11], and of scripting languages such as JavaScript [24]. Roughly speaking, most call graph construction algorithms can be classified as being either *type-based* or *flow-based* [6, 12, 13, 16, 17]. 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 data flow.

We focus on type-based algorithms, so we will 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 [26], 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., [25]) 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:

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

$$\frac{c \mapsto M}{M \in R} \; \text{RA}_{\text{REACHABLE}} \qquad \frac{\begin{array}{c} M \in R \end{array}}{c \mapsto M'} \; \text{RA}_{\text{CALL}}$$

Intuitively, rule RA$_{\text{MAIN}}$ reads "the main method is reachable" by including it in the set R of reachable methods. Rule RA$_{\text{CALL}}$ states that "if a method is reachable, and a call site $c : e.m(\ldots)$ occurs in its body, then every method with name m is reachable from c." Finally, rule RA$_{\text{REACHABLE}}$ simply states that any method M that is 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 [8] 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 [26], 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 run-time type C. Using these definitions, CHA is defined as follows:

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

$$\frac{c \mapsto M}{M \in R} \; \text{CHA}_{\text{REACHABLE}} \qquad \frac{}{c \mapsto M'} \; \text{CHA}_{\text{CALL}}$$

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 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 [4, 5] observed 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 run time. 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 run time. The constraint formulation of RTA is as follows:

$$\frac{}{main \in R} \; \text{RTA}_{\text{MAIN}}$$

$$\frac{\text{"new } C() \text{" occurs in } M}{C \in \hat{\Sigma}} \; \text{RTA}_{\text{NEW}} \qquad \begin{array}{c} \text{call } e.m(\ldots) \text{ occurs in method } M \\ C \in SubTypes(StaticType(e)) \\ StaticLookup(C, m) = M' \\ M \in R \\ C \in \hat{\Sigma} \end{array}$$

$$\frac{c \mapsto M}{M \in R} \; \text{RTA}_{\text{REACHABLE}} \qquad \frac{}{c \mapsto M'} \; \text{RTA}_{\text{CALL}}$$

Rules RTA_{MAIN} and $\text{RTA}_{\text{REACHABLE}}$ are again the same as before. Intuitively, RTA_{CALL} refines the corresponding constraint of CHA 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 [21] recently presented an extension of RTA for the C# programming 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.

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.

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

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

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 trait may extend ("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, a trait C is declared that 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.

Before we can study the challenges associated with constructing call graphs, 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 [18, Section 5.1.2]. The *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 operand[4]. Scala defines the set of *members* of a class in terms of its linearization. Ignoring a number of complicating factors detailed in Sections 5.1.3 and 5.1.4 of the Scala specification [18], the members of a class C include all members m declared in classes in $\mathcal{L}(C)$, except for those that are overridden in classes that precede C in the linearization order. Given this notion of class membership, the resolution of method calls is straightforward: a call $x.m(\cdots)$ where x has type C at run time dispatches to the unique member named m in C.

For the example program of Figure 1, the linearization order of the type new A with B on line 15 is: X, B, A (here, we use X to denote the anonymous class

---

[4] The presence of an allocation expression such as new C with D is equivalent to a declaration of a new empty class with parents C with D.

that is implicitly declared by the allocation expression new A with B). Following the definitions above, the set of 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 result in printing "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[5], such that the resulting combination is syntactically correct[6]. Then, for each of these combinations, one could compute the members contained in the resulting type, and approximate the behavior of method calls by determining the method that would be selected in each case. Applying this approach to the example program of Figure 1 means assuming 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 case, the call is assumed to be invoked on an object of type C with B (or B with C), and would dispatch to C.foo. Hence, this conservative approach would create a call graph in which both A.foo and C.foo are reachable from the call on line 8.

The conservative approach discussed above is likely to be imprecise and inefficient in cases where a program contains many traits that can be composed with each other. An alternative 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 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 is only one outgoing edge for the call on line 8.

3.2 Abstract Type Members

Scala supports a flexible mechanism for declaring *abstract type members* in traits and classes. A *type declaration* [18, Section 4.3] defines a name for an abstract

---

[5] 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.

[6] 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 [18, Section 5.1.4].

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

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

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* [18, Section 4.3] that explicitly binds the abstract type to some specified concrete type.

Figure 2 shows an example 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. Traits Y and Z each declare an abstract type member B and a field o, which is assigned a new A object in Y. Note that Y defines its B to be the same as Y.A. Observe that, in type Z, the abstract member type B 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, as shown on line 36.

The example program creates an object by composing Y with Z on line 40, and calls bar on it. Following Scala's semantics for method calls, the call to bar on this line will dispatch to Z.bar. Then, the call o.foo on line 36 is executed. To understand how this call is resolved, we must understand how abstract type members are bound to concrete types as a result of trait composition. In this case, the composition of Y with Z means that the types Y.B and Z.B are unified. Since Y.B was 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 type of the receiver expression is abstract? A conservative way to approximate the behavior of such calls would rely 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

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

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

assume that the call could dispatch to definitions of $f(\cdots)$ in any subtype of B. This approach, which is implemented in our TCA^{bounds} algorithm, would identify both X.A.foo and Y.A.foo in our example as possible targets of the call on line 36.

However, the above approach is likely to be imprecise if certain subtypes of the bound are not instantiated. A more precise approach (which is implemented in our TCA^{expand} algorithm) would consider 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 that we discussed above. In the case of the example in Figure 2, we can observe that the program only creates an object of type Y with Z, and that 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.

In addition to abstract type members, Scala also supports parameterized types [18, Section 3.2.4]. These features are very similar, and it is therefore unsurprising that the same analysis issues as the ones discussed above arise for parameterized types. In fact, similar issues arise in programming languages with generics, such as Java and C# [21].

3.3 Closures

Scala allows functions to be bound to variables and passed as arguments to other functions. Figure 3 shows an example of this feature, commonly known as "closures" because they close over their surrounding environment. On line 49, the program creates a function and assigns it to a variable foo1. The type of this function is declared as () => A, indicating that it takes no parameters and returns an object of type A. Likewise, on line 50, another function is bound to variable foo2, which takes no arguments and returns an object of type B.

Next, on line 51, method bar1 is called with foo1 as an argument. Method bar1 (line 43) takes a closure with type () => A as an argument and binds it to its parameter y. It then calls the function bound to y on line 43. Similarly, on line 52 method bar2 is called with foo2 as an argument and binds it to a parameter z, which is of type () => B. Method bar2 calls the function bound to z on line 44. From the simple data flow 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.

8

```scala
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
65        scala.runtime.AbstractFunction0[A] {
66          def apply(): A = { new A() }
67        };
68        new $anonfun()
69      };
70      val foo2: () => B = {
71      class $anonfun extends
72        scala.runtime.
73              AbstractFunction0[B] {
74          def apply(): B = {
75            new B()
76          }
77        };
78        new $anonfun()
79      };
80      this.bar1(foo1)
81      this.bar2(foo2)
82    }
83  }
```

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

Implications for call graph construction. In principle, one could consider the declared types of function-valued expressions and the types of the 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 a closure that can be bound to a variable of this type is created on line 51. Therefore, a call graph edge needs to be constructed from the call site y() to the closure on line 51. Following 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 resolve calls on closures. Rather than performing the analysis at the source level, we apply it to a "desugared" version of the code. In this code, closures have been transformed by the Scala compiler into anonymous classes that extend the appropriate scala.runtime.AbstractFunction$N$. For each anonymous class, the Scala compiler generates an apply() method that contains the closure's original code. Figure 4 shows a desugared version of the example program of Figure 3. After this transformation, our analysis can treat closures just like 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 an example 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 also declares a main method that creates objects of type B and C, and calls bar on each of them (see lines 100–101). Executing the call to bar on line 100 dispatches to A.bar(). Executing the call this.foo() in that method will then dispatch to B.foo().

```scala
84  object This {                          94    class C extends A {
85    trait A {                            95      def foo = println("C.foo")
86      def foo                            96      override def bar = println("C.bar")
87      // can only call B.foo              97    }
88      def bar = this.foo                 98
89    }                                    99    def main(args: Array[String]) = {
90                                         100      (new B).bar
91    class B extends A {                  101      (new C).bar
92      def foo = println("B.foo")         102    }
93    }                                    103  }
```

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

Finally, executing the call to bar on line 101 dispatches to C.bar, so the program will print "B.foo" then "C.bar".

Now consider how a call graph construction algorithm would approximate the behavior of the call this.foo() at line 88. Since the type of the receiver expression is A, CHA would conclude that either B.foo or C.foo might be invoked from this call site, because B and C are both subtypes of A.

However, in this case, one can observe that the type of this cannot be C 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 that will be presented in Section 4, exploits such knowledge. Care must be taken in the presence of super-calls, as we will discuss in Section 5.

3.5 Bytecode-based Analysis

The above 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 5. 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 have experimentally determined that such an approach is not a viable option 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. This causes the computed call graphs to become imprecise. Second, the Scala compiler generates code containing hard-to analyze reflection for certain Scala idioms. We briefly discuss each of these issues.

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 were defined on lines 49 and 50 in Figure 3 have been translated into classes Closures$$anonfun$1 (lines 126–136 in Figure 6) and Closures$$anonfun$2 (lines 138–148). These classes extend scala.runtime.AbstractFunction0<T>, which is

```
104  public  final  class  Closures$  {
105    public  void  main(java.lang.String []);
106        0:  new  Closures$$anonfun$1
107        ...
108        8:  new  Closures$$anonfun$2
109        ...
110       18:  invokevirtual  Closures$.bar1( scala .Function0) :  void
111        ...
112       23:  invokevirtual  Closures$.bar2( scala .Function0) :  void
113       26:  return
114    public  void  bar1( scala .Function0);
115        0:  aload_1
116        1:  invokeinterface  scala .Function0.apply() :  java .lang .Object
117        6:  pop
118        7:  return
119    public  void  bar2( scala .Function0);
120        0:  aload_1
121        1:  invokeinterface  scala .Function0.apply() :  java .lang .Object
122        6:  pop
123        7:  return
124  }
125
126  public  final  class  Closures$$anonfun$1 extends scala .runtime .AbstractFunction0 {
127    public  final  Closures$A apply ();
128        0:  new  Closures$A
129        3:  dup
130        4:  invokespecial  Closures$A()
131        7:  areturn
132    public  final  java .lang .Object apply ();
133        0:  aload_0
134        1:  invokevirtual  Closures$$anonfun$1.apply() :  Closures$A
135        4:  areturn
136  }
137
138  public  final  class  Closures$$anonfun$2 extends scala .runtime .AbstractFunction0 {
139    public  final  Closures$B apply ();
140        0:  new  Closures$B
141        3:  dup
142        4:  invokespecial  Closures$B()
143        7:  areturn
144    public  final  java .lang .Object apply ();
145        0:  aload_0
146        1:  invokevirtual  Closures$$anonfun$2.apply() :  Closures$B
147        4:  areturn
148  }
```

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

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 a situation where:

– 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

11

– there are two calls to scala.Function0.apply on lines 116 and 121

Given this situation, the RTA algorithm will create edges to Closures\$\$anonfun\$1.apply and Closures\$\$anonfun\$2.apply from each of the calls on lines 116 and 121. In other words, a bytecode-based RTA analysis creates 4 call graph edges for the closure-related calls, whereas the analysis we discussed in Section 3.3 only created 2 edges. In Section 6, we show that this scenario is very common in practice, and that bytecode-based call graphs tend to become extremely imprecise as a result.

Reflection in Generated Code. We detected several cases where the Scala compiler generates code that contains reflection, where methods are invoked using calls to java.lang.reflect.Method.invoke(). 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.

Figure 7 shows a small example for which the Scala compiler generates code containing reflection. This example was taken from one of the programs that we studied as part of our evaluation (see Section 6). The particular idiom that causes the Scala compiler to generate reflective code appears to have to do with the presence of mutable fields in anonymous classes. We are not involved in the development of the Scala compiler, and we do not have a good understanding why reflective code is being generated. However, situations such as the one shown in Figure 7 do occur in practice, as we shall see in Section 6.

4   Algorithms

We present a family of call graph construction algorithms using generic inference rules, in the same style that we used in Section 2. The algorithms presented

```
149  trait ClassHandler
150
151  object LuceneIndex {
152    def buildStaticIndex (): Int = {
153      val handler = new ClassHandler {
154        var classCount = 0
155        var methodCount = 0
156      }
157      handler.classCount + handler.methodCount
158    }
159  }
```

Fig. 7. A Scala program for which the compiler generates code containing reflective method calls (taken from the *ensime* program, see Section 6).

here are: 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 $\mathrm{TCA}^{expand\text{-}this}$, which is more precise for call sites where the receiver is this.

We use Figure 8 to illustrate differences between the algorithms. When executed, the call site on line 170 calls method B.foo; our different algorithms resolve this call site to various subsets of the foo methods in classes A, B, C, and D.

```
160  class  A { def foo = "A.foo" }
161  class  B extends A { override def foo = "B.foo" }
162  class  C { def foo = "C.foo" }
163  class  D { def foo = "D.foo" }
164  class   CallSiteClass [T <: A](val  receiver :  T) {
165    def  callsite  = {
166      /∗ resolves  to:
167        ∗   TCA^expand: { B.foo },  TCA^bounds: { B.foo, A.foo }
168        ∗   TCA^names: { B.foo, A.foo, C.foo } ,   RA: { B.foo, A.foo, C.foo, D.foo }
169        ∗/
170        receiver .foo
171    }
172  }
173  def main(args: Array[String ]):  Unit = {
174    new A
175    val  receiver  = new B
176    new C
177    val  callSiteClass  = new CallSiteClass[B]( receiver );
178     callSiteClass . callsite
179  }
```

Fig. 8. A Scala program illustrating the varying precision of the analyses.

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 8, 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. We add rule $\mathrm{RTA}_{\mathrm{NEW}}$ from the RTA analysis, which computes a set $\hat{\Sigma}$ of types instantiated in reachable methods. The CALL rule[7] is then adapted as follows

---

[7] When we present an inference rule in this section, we use shading to highlight which parts of the rule are modified relative to similar preceding rules.

to make use of $\hat{\Sigma}$:

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

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[8].

For the program of Figure 8, 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 $C \in SubTypes(StaticType(e))$ 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[9].

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 can 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 name-based algorithms RA and TCA^{names}, though it would obviously 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

---

[8]   Calls on super require special handling, as will be discussed in Section 5.

[9]   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.

collect the set $\hat{\Sigma}$ of trait combinations occurring at reachable allocation sites. The resulting set is used in the following call resolution rule:

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

The added check $C \in SubTypes(StaticType(e))$ 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.

However, 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 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 we can approximate an abstract type T using its upper bound B:

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

For the program of Figure 8, 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 actual concrete types with which each abstract type is instantiated. This approach is also used by Sallenave and Ducourneau for .NET programs [21]. To this end, we introduce a mapping $expand()$, which

maps each abstract type[10] T to those concrete types with which it has been instantiated:

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

$$\frac{\begin{array}{c} C \in \hat{\Sigma} \\ \text{``trait } A \ \{ \dots \}\text{'' is a member of } C \\ D \text{ is a supertype of } C \\ \text{``type } A\text{'' is a member of } D \end{array}}{C.A \in expand(D.A)} \ \text{TCA}_{\text{EXPAND-TRAIT}}^{expand} \qquad \frac{\begin{array}{c} R \in expand(S) \\ S \in expand(T) \end{array}}{R \in expand(T)} \ \text{TCA}_{\text{EXPAND-TRANS}}^{expand}$$

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

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

Rule $\text{TCA}_{\text{EXPAND-TYPE}}^{expand}$ handles situations such as the one where a type assignment 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 $\text{TCA}_{\text{EXPAND-TYPE}}^{expand}$. Rule $\text{TCA}_{\text{EXPAND-TRAIT}}^{expand}$ handles a similar case where an abstract type is instantiated by defining a member trait with the same name as the abstract type. Because the right-hand-side of a type assignment might also be an abstract type, it is necessary to compute the transitive closure of the $expand()$ mapping. This is accomplished by rule $\text{TCA}_{\text{EXPAND-TRANS}}^{expand}$.

It is possible to have cycles among type assignments. In Scala, cyclic references between abstract 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]], A[A[A[B]]], ...[11]. Our solution to

---

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

[11] This issue also occurs in Java and C#, and was previously noted by Sallenave and Ducourneau [21]. However, Sallenave and Ducourneau do not provide a solution for this case and only issue a warning if recursion is detected [21, Algorithm 8].

handling this case is to limit the size of $expand(T)$ for any abstract type T to 1000 types. If this threshold is exceeded, we fall back on using the upper bound of T to resolve calls. However, this bound was never exceeded in our experimental evaluation.

In the example of Figure 8, the type parameter T of the generic class CallSiteClass is instantiated only with type B (on line 177). Therefore, $expand(T)$ evaluates to B, so TCA^{expand} resolves the call site at line 170 only to B.foo.

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

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[12].

For example, at the call this.foo() on line 88 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 92) and C.foo (line 95). However, any object that has method C.foo as a member will also have C.bar as a member, which overrides method A.bar that contains the call site. Therefore, the call site at line 88 can never resolve to method C.foo.

This pattern is handled precisely by the following rule:

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

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 $\text{TCA}^{expand\text{-}this}_{\text{THIS-CALL}}$, we add a precondition to rule $\text{TCA}^{expand\text{-}this}_{\text{CALL}}$ so that it does not apply when $\text{TCA}^{expand\text{-}this}_{\text{THIS-CALL}}$ does, i.e., when the receiver is the special variable this of the declaring trait D of the caller method M.

---

[12] Care must be taken in the presence of super-calls, as will be discussed in Section 5.1.

4.5 Correctness

The supplemental material associated with this submission provides a formalization of the inference rules for $\text{TCA}^{expand\text{-}this}$ based on the FS_{alg} ("Featherweight Scala") representation of Cremet et al. [7]. We have proven the $\text{TCA}^{expand\text{-}this}$ analysis correct with respect to the operational semantics of FS_{alg} by demonstrating that:

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 $\text{TCA}^{expand\text{-}this}$.
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 $\text{TCA}^{expand\text{-}this}$ applied to P derives $c \mapsto M$.

5 Implementation

We implemented RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, and $\text{TCA}^{expand\text{-}this}$ 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, but we assume that all code to be analyzed is available and we ignore reflection and dynamic code generation. We also used the implementation of RTA in the WALA framework to construct call graphs from JVM bytecode.

The analysis runs after the `uncurry` phase, which is the 12th of 30 compiler phases. 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. Operations such as subtype testing are implemented using methods provided by the Scala compiler. Two issues, however, require special handling in the implementation: super calls and partial programs.

5.1 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 generally 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 [18, Section 6.5]. Our analysis resolves such calls using a similar procedure as for ordinary dynamically dispatched calls. For each possible run-time type of the receiver, the specified procedure is followed to find the actual call target.

The $\text{TCA}^{expand\text{-}this}$ 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 TCA^{expand} analysis instead of $\text{TCA}^{expand\text{-}this}$.

5.2 Incomplete programs

Our analyses are defined for complete programs, but a practical implementation must deal with situations where only parts of the program are available for analysis. Our implementation analyzes Scala source files presented to the compiler, but not any referenced classes provided only as bytecode, including the Scala and Java standard libraries. The analysis soundly analyzes all call sites occurring in the provided Scala source files using a Scala analogue of the Separate Compilation Assumption [2, 3], which asserts that unanalyzed "library" classes do not directly reference analyzed "application" classes. 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 these call sites, but does not attempt to resolve them soundly.

However, these call sites, as well as call sites in unanalyzed code, may call methods in the analyzed code via call-backs. For soundness, the analysis must treat such target methods as reachable. The implementation achieves this by considering every method reachable if it occurs in an instantiated type and if it overrides some method declared in the 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. However, determining the method overriding relationship is more difficult than in Java. Consider two methods declared in two independent traits. The methods do not override each other unless the two traits are composed in the instantiation of some object. Therefore, the analysis implementation updates the method overriding relation as new trait compositions are discovered.

6 Evaluation

We evaluated our implementation on a number of publicly available Scala programs covering a range of different application areas and programming styles.[13]

---

[13] The benchmark source code is available from `http://github.com/bmc/argot`, `http://github.com/aemoncannon/ensime`, `http://github.com/KarolS/fimpp`,

Table 1 shows, for each program, the number of lines of Scala source code (excluding any 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. 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 Integrated Development Environments, such as live error-checking, package/type browsing, and basic refactorings. FIMPP is an interpreter for an imperative, dynamically-typed, interpreted language that supports integer arithmetic, console output, dynamically growing arrays, and subroutines. KIAMA is a Scala 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 [15]. 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. Both KIAMA and SCALAXB are part of the DaCapo Scala Benchmarking project [22]. We did not use the other DaCapo Scala benchmarks because their full source is not compatible with the latest version of Scala.

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 16 GB of RAM.

6.1 Research Questions

Our evaluation aims to answer the following Research Questions:

RQ1. How precise are call graphs constructed for the JVM bytecode produced by the Scala compiler compared to analyzing Scala source code?

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?

6.2 Results

Table 2 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

---

Table 1. 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	1,591	1	143
ENSIME	7,832	223	172	36	1,016	4,878	532	12,939	22	2,293
FIMPP	1,089	42	53	5	685	2,060	549	4,677	4	902
KIAMA	17,914	801	664	162	5,324	19,172	3,963	45,472	368	9,237
PHANTM	9,319	317	358	13	1,492	7,208	561	21,277	14	3,278
SCALAXB	10,290	324	259	222	2,978	10,503	2,204	33,536	32	5,279
SCALISP	795	20	14	0	125	428	115	1,382	13	167
SEE	4,311	130	151	17	419	2,280	262	6,367	6	954
SQUERYL	7,432	255	55	110	1,036	3,793	826	9,436	111	1,698
TICTACTOE	247	2	7	0	32	112	24	377	0	29

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 4.4x 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[14]. This further reaffirms that a bytecode-based approach to call graph construction is highly problematic.

RQ2. To answer the second research question, we compare TCA^{names} and TCA^{bounds} and find that name-based analysis incurs a very significant precision

---

[14] The summarized call graph computed by RTA^{wala} shown in Table 2 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 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.

Table 2. Number of nodes and edges in the summarized version of call graphs computed using the RA, TCA^{names}, TCA^{bounds}, TCA^{expand}, $\mathrm{TCA}^{expand\text{-}this}$, and RTA^{wala}.

		RA	TCA^{names}	TCA^{bounds}	TCA^{expand}	$\mathrm{TCA}^{expand\text{-}this}$	RTA^{wala}
ARGOT	nodes	265	184	161	161	161	236
	edges	3,516	1,538	442	442	440	648
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
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	34,062	32,494	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
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

loss: The call graphs generated by TCA^{names} have, on average, 10.9x as many call edges as those generated by TCA^{bounds}. Investigating further, we found that, on average, 66% of the spurious call edges computed by the name-based analysis were to `apply` methods, which implement closures.

RQ3. To answer the third research question, we compare the TCA^{bounds} and TCA^{expand} analyses. On the smaller benchmark programs that make little use of abstract types, the two produce identical results. Since KIAMA, PHANTM, and SEE contain some call sites on receivers with abstract types, TCA^{expand} com-

Table 3. The 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	3.7	3	3	3.3	3.5	11.5	25
ENSIME	31.7	26	25.3	27.4	27.3	488.6	58.2
FIMPP	5.1	5	7.2	7.5	7.5	13.7	34.6
KIAMA	293.4	85.3	122.5	125.6	111.4	67.5	102.5
PHANTM	53.9	43.9	56.1	54.7	54	26.2	67.6
SCALAXB	110.8	17.2	11.4	11.9	12.2	20.4	81
SCALISP	2.9	2.8	2.8	3.1	3.2	12.2	25.2
SEE	6.9	6.3	8	8.4	8.4	13.1	38.2
SQUERYL	20.7	11.6	5.8	6.3	6.2	19.6	50.5
TICTACTOE	1.5	1.5	1.7	1.8	1.8	10.4	15.9

putes more precise call graphs for them. For SCALAXB, 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. In summary, tracking concrete types bound to abstract types improves call graph precision, but only on larger benchmarks that make significant use of abstract types.

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 found unreachable by TCA$^{expand\text{-}this}$.

RQ5. The running times of the analyses are presented in Table 3. For comparison, the last column of the table also shows the time required to compile each benchmark using the unmodified Scala compiler. Although our implementation has not been heavily tuned for performance, the analysis times are reasonable compared to `scalac` compilation times. 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 the 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).

7   Conclusions

We presented a family of low-cost algorithms for constructing call graphs of Scala programs, in the spirit of Name-Based Resolution (RA) [25], Class Hierarchy Analysis (CHA) [8] and Rapid Type Analysis (RTA) [5]. 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 proposed 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, $\text{TCA}^{expand-this}$ only did significantly better than TCA^{expand} on programs that make nontrivial use of subtyping and method overriding. Finally, RTA call graphs constructed from the JVM bytecodes are less precise because significant type information is lost during compilation.

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++.

References

1. AGESEN, O. Constraint-based Type Inference and Parametric Polymorphism. In *SAS* (1994), pp. 78–100.
2. ALI, K., AND LHOTÁK, O. Application-only Call Graph Construction. In *ECOOP* (2012), pp. 688–712.
3. ALI, K., AND LHOTÁK, O. Averroes: Whole-program analysis without the whole program. In *ECOOP* (2013), pp. 378–400.
4. BACON, D. F. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, 1997.
5. BACON, D. F., AND SWEENEY, P. F. Fast static analysis of C++ virtual function calls. In *OOPSLA* (1996), pp. 324–341.
6. BRAVENBOER, M., AND SMARAGDAKIS, Y. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA* (2009), pp. 243–262.

7. CREMET, V., GARILLOT, F., LENGLET, S., AND ODERSKY, M. A core calculus for Scala type checking. In *MFCS* (2006), pp. 1–23.

8. DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP* (1995), pp. 77–101.

9. DEFOUW, G., GROVE, D., AND CHAMBERS, C. Fast Interprocedural Class Analysis. In *POPL* (1998), pp. 222–236.

10. GROVE, D., AND CHAMBERS, C. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*, 6 (2001), 685–746.

11. HEINTZE, N. Set-Based Analysis of ML Programs. In *LISP and Functional Programming* (1994), pp. 306–317.

12. HEINTZE, N., AND TARDIEU, O. Ultra-fast Aliasing Analysis using CLA: A Million Lines of C Code in a Second. In *PLDI* (2001), pp. 254–263.

13. HENGLEIN, F. Dynamic Typing. In *ESOP* (1992), pp. 233–253.

14. IBM. T.J. Watson Libraries for Analysis WALA. `http://wala.sourceforge.net/`, April 2013.

15. KNEUSS, E., SUTER, P., AND KUNCAK, V. Phantm: PHP analyzer for type mismatch. In *SIGSOFT FSE* (2010), pp. 373–374.

16. LHOTÁK, O., AND HENDREN, L. J. Scaling Java Points-to Analysis Using SPARK. In *CC* (2003), pp. 153–169.

17. LHOTÁK, O., AND HENDREN, L. J. Context-Sensitive Points-to Analysis: Is It Worth It? In *CC* (2006), pp. 47–64.

18. ODERSKY, M. The Scala Language Specification version 2.9. Tech. rep., EPFL, May 2011. DRAFT.

19. ODERSKY, M., SPOON, L., AND VENNERS, B. *Programming in Scala*, 2nd ed. Artima Press, 2012.

20. RYDER, B. Constructing the call graph of a program. *IEEE Transactions on Software Engineering 5*, 3 (1979), 216–226.

21. SALLENAVE, O., AND DUCOURNEAU, R. Lightweight generics in embedded systems through static analysis. In *LCTES* (2012), pp. 11–20.

22. SEWE, A., MEZINI, M., SARIMBEKOV, A., AND BINDER, W. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. In *OOPSLA* (2011), pp. 657–676.

23. SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991.

24. SRIDHARAN, M., DOLBY, J., CHANDRA, S., SCHÄFER, M., AND TIP, F. Correlation Tracking for Points-To Analysis of JavaScript. In *ECOOP* (2012), pp. 435–458.

25. SRIVASTAVA, A. Unreachable procedures in object oriented programming. *ACM Letters on Programming Languages and Systems 1*, 4 (December 1992), 355–364.

26. TIP, F., AND PALSBERG, J. In *OOPSLA* (2000), pp. 281–293.

27. VALLÉE-RAI, R., GAGNON, E., HENDREN, L. J., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *CC* (2000), pp. 18–34.

In this supplemental appendix, we review FS_{alg} (Featherweight Scala), formalize the TCA$^{expand\text{-}this}$ analysis for FS_{alg}, and prove it sound.

A Extensions to Featherweight Scala

Cremet et al. [7] 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 felt 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 straight-forward manner, by allowing abstract type members to have upper bounds just like in full Scala[15]. 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 .

Figure 9 shows the syntax of FS_{alg}. An FS_{alg} *program* is a term which is usually of the form

$$\texttt{val}\ \ z = \texttt{new}\ \ \left\{\varphi \,|\, \overline{M}\right\}; t$$

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. [7] is the addition of a *bounded type declaration* of the form $\texttt{type}_n A <: T$, which we introduced to allow calls on variables whose type is an abstract type member, as was illustrated in Figure 2.

Figure 10 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 [7].

Figure 11 presents a set of lookup judgments that define the set of members of a given object based on the type signature that it was instantiated with.

---

[15] Note that, in full Scala, abstract type members may also have lower bounds, which we do not support here because they are not used by TCA$^{expand\text{-}this}$.

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 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 that is used in Figure 11 deserves some additional discussion. In FS_{alg} [7], the $\uplus$ operator is defined as concatenation with rewriting of common members: $\overline{M} \uplus \overline{N} = \overline{M}|_{\mathrm{dom}(\overline{M})\backslash\mathrm{dom}(\overline{N})}, \overline{N}$, where $\mathrm{dom}(\overline{M})$ is the set of labels 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 contains the optional $= t$ or $= T$ or if it is a trait declaration, and *abstract* otherwise. In particular, the bounded type declaration is considered abstract. In contrast, the Scala specification [18, 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:
$$\overline{M} \uplus \overline{N} = \overline{M}|_{(\mathrm{cdom}(\overline{M})\backslash\mathrm{cdom}(\overline{N}))\cup(\mathrm{adom}(\overline{M})\backslash\mathrm{dom}(\overline{N}))}, \overline{N}|_{\mathrm{cdom}(\overline{N})\cup(\mathrm{adom}(\overline{N})\backslash\mathrm{cdom}(\overline{M}))},$$

x, y, z, φ	Variable
a	Value label
A	Type label
$P ::=$	Program
$\quad \{x \mid \overline{M}\, t\}$	
$M, N ::=$	Member decl
$\quad \mathtt{val}_n a\, :\, T\, (= t)^?$	Field decl
$\quad \mathtt{def}_n a(y\, :\, \overline{S})\, :\, T(= t)^?$	Method decl
$\quad \mathtt{type}_n A(= T)^?$	Type decl
$\quad \mathtt{type}_n A <: T$	Bounded type decl
$\quad \mathtt{trait}_n A\ \mathtt{extends}(\overline{T})\,\{\varphi \mid \overline{M}\}$	Class decl
$s, t, u ::=$	Term
$\quad x$	Variable
$\quad t.a$	Field selection
$\quad s.a(\overline{t})$	Method call
$\quad \mathtt{val}\, x = \mathtt{new}\ T;\ t$	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. 9. FeatherWeight Scala Syntax

$$\frac{\mathtt{val}_n\,a : T = t \in \Sigma(x)}{\Sigma\,;\,x.a \to \Sigma\,;\,t} \quad (\text{RED-VALUE})$$

$$\frac{\Sigma \vdash T \prec_x \overline{M}}{\Sigma\,;\,\mathtt{val}\,x = \mathtt{new}\,T;t \to \Sigma, x : \overline{M}\,;\,t} \quad (\text{RED-NEW})$$

$$\frac{\mathtt{def}_n\,a(\overline{z : S}) : T = t \in \Sigma(x)}{\Sigma\,;\,x.a(\overline{y}) \to \Sigma\,;\,[\overline{y}/\overline{z}]t} \quad (\text{RED-METHOD})$$

$$\frac{\Sigma\,;\,t \to \Sigma'\,;\,t'}{\Sigma\,;\,e[t] \to \Sigma'\,;\,e[t']} \quad (\text{RED-CONTEXT})$$

where

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

$\langle\rangle$

$e.a$

$e.a(t)$

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

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

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

$e.a$

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

Fig. 10. Reduction

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 12 and 13 define the type assignment relations that determine a static type T for each form of term t. 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 14 and 15 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 the expansion judgment is not defined for abstract type members of the form $\mathbf{type}_n A$. Therefore, in the original FS_{alg}, it was impossible to call a method on a receiver whose type was 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 $\mathbf{type}_n A <: T$ the members of their bound T. This makes a method call on a (bounded) abstract bound 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 $\mathbf{this}$, 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.

In Figure 16, we define a relation that extends the members of a type to also include members of all of its supertypes. This ancestor membership relation is needed for $\mathrm{TCA}^{expand\text{-}this}$, and is not present in the original FS_{alg} formalization.

$$\frac{\forall i, \; \Sigma \vdash T_i \prec_{\varphi} \overline{N_i}}{\Sigma \vdash \left(\overline{T}\right)\left\{\varphi \,|\, \overline{M}\right\} \prec_{\varphi} \left(\biguplus_i \overline{N_i}\right) \uplus \overline{M}} \quad \text{(LOOKUP-SIG)}$$

$$\frac{\mathbf{trait}_n A \,\mathbf{extends}\, \left(\overline{T}\right)\left\{\varphi \,|\, \overline{M}\right\} \in \Sigma(y)}{\Sigma \vdash \left(\overline{T}\right)\left\{\varphi \,|\, \overline{M}\right\} \prec_{\varphi} \overline{N}}{\Sigma \vdash y.A \prec_{\varphi} \overline{N}} \quad \text{(LOOKUP-CLASS)}$$

$$\frac{\mathbf{type}_n A = T \in \Sigma(y)}{\Sigma \vdash T \prec_{\varphi} \overline{M}}{\Sigma \vdash y.A \prec_{\varphi} \overline{M}} \quad \text{(LOOKUP-ALIAS)}$$

$$\frac{\mathbf{type}_n A <: T \in \Sigma(y)}{\Sigma \vdash T \prec_{\varphi} \overline{M}}{\Sigma \vdash y.A \prec_{\varphi} \overline{M}} \quad \text{(LOOKUP-BOUNDED-TYPE)}$$

Fig. 11. Lookup

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

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

Fig. 12. Path typing

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

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

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

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

Fig. 13. Type assignment

The $\text{TCA}^{expand\text{-}this}$ algorithm uses this relation to determine which abstract type members defined in supertypes are overridden by concrete members in instantiated subtypes.

Figure 17 defines the subtyping relation $<:$ between types. $\text{TCA}^{expand\text{-}this}$ 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 an alias expansion relation, shown in Figure 18, that expands type aliases of the form $\texttt{type}_n A = T$, so that subtypes of T can also be considered subtypes of A within the context of the object that defines the type alias $\texttt{type}_n A = T$. The subtyping relation also uses the member subtyping relation, shown in Figure 19, to enforce that a type is allowed to extend another type only when their respective members have compatible types.

B Formalization of $\text{TCA}^{expand\text{-}this}$ for FS_{alg}
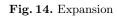
In this part of the appendix, we present a formalization of the $\text{TCA}^{expand\text{-}this}$ algorithm on FS_{alg}. The full $\text{TCA}^{expand\text{-}this}$ algorithm formalized for FS_{alg} is presented in Figure 20.

$$\dfrac{\begin{array}{c} \mathcal{S},\ \Gamma \vdash p.\textbf{type} \ni \texttt{trait}_n A\ \texttt{extends}(\overline{T})\ \{\varphi \mid \overline{M}\} \\ \{n\} \cup \mathcal{S},\ \Gamma \vdash (\overline{T})\ \{\varphi \mid \overline{M}\} \prec_\varphi \overline{N} \qquad n \notin \mathcal{S} \end{array}}{\mathcal{S},\ \Gamma \vdash p.A \prec_\varphi \overline{N}} \qquad (\prec\text{-CLASS})$$

$$\dfrac{\begin{array}{c} \mathcal{S},\ \Gamma \vdash p.\textbf{type} \ni \texttt{type}_n A = T \\ \{n\} \cup \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M} \qquad n \notin \mathcal{S} \end{array}}{\mathcal{S},\ \Gamma \vdash p.A \prec_\varphi \overline{M}} \qquad (\prec\text{-TYPE})$$

$$\dfrac{\begin{array}{c} \mathcal{S},\ \Gamma \vdash p.\textbf{type} \ni \texttt{type}_n A <: T \\ \{n\} \cup \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M} \qquad n \notin \mathcal{S} \end{array}}{\mathcal{S},\ \Gamma \vdash p.A \prec_\varphi \overline{M}} \qquad (\prec\text{-BOUNDED-TYPE})$$

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

Fig. 14. Expansion

$$\dfrac{\begin{array}{c} \mathcal{S},\ \Gamma \vdash p \simeq q \qquad \mathcal{S},\ \Gamma \vdash_{path} q : T \\ \psi(p) \cup \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M} \qquad \psi(p) \nsubseteq \mathcal{S} \end{array}}{\mathcal{S},\ \Gamma \vdash p.\textbf{type} \ni [p/\varphi]M_i} \qquad (\ni\text{-SINGLETON})$$

$$\dfrac{\begin{array}{c} T \text{ is not a singleton type} \\ \mathcal{S},\ \Gamma \vdash T \prec_\varphi \overline{M} \qquad \varphi \notin \mathrm{fn}(M_i) \end{array}}{\mathcal{S},\ \Gamma \vdash T \ni M_i} \qquad (\ni\text{-OTHER})$$

Fig. 15. Membership

$$\frac{\mathcal{S},\, \Gamma \vdash T \ni M}{\mathcal{S},\, \Gamma \vdash T \ni^{\cup} M} \qquad (\ni^{\cup}\text{-}\textsc{Base})$$

$$\frac{\mathcal{S},\, \Gamma \vdash p.\textbf{type} \ni \texttt{trait}_n A \, \texttt{extends}(\overline{T})\,\{\varphi \mid \overline{M}\} \qquad \mathcal{S},\, \Gamma \vdash T_i \ni^{\cup} N}{\mathcal{S},\, \Gamma \vdash p.A \ni^{\cup} N} \qquad (\ni^{\cup}\text{-}\textsc{class})$$

$$\frac{\mathcal{S},\, \Gamma \vdash p.\textbf{type} \ni \texttt{type}_n A = T' \qquad \mathcal{S},\, \Gamma \vdash T' \ni^{\cup} N}{\mathcal{S},\, \Gamma \vdash p.A \ni^{\cup} N} \qquad (\ni^{\cup}\text{-}\textsc{alias})$$

$$\frac{\mathcal{S},\, \Gamma \vdash p.\textbf{type} \ni \texttt{type}_n A <: T' \qquad \mathcal{S},\, \Gamma \vdash T' \ni^{\cup} N}{\mathcal{S},\, \Gamma \vdash p.A \ni^{\cup} N} \qquad (\ni^{\cup}\text{-}\textsc{bounded})$$

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

Fig. 16. Ancestor membership

The FS_{alg} type 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. In our formalization of $\text{TCA}^{expand\text{-}this}$, 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 $p.\textbf{type}$ 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 $\text{TCA}^{expand\text{-}this}$, 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 $\text{TCA}^{expand\text{-}this}$, 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 $\texttt{val } x = \texttt{new } T; u$. 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

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

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

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

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

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

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

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

Fig. 17. Algorithmic subtyping

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

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

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

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

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

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

Fig. 18. Type alias expansion

33

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

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

$$\frac{T <: T'}{\mathcal{S}, \Gamma \vdash \mathtt{type}_n A = T <: \mathtt{type}_n A <: T'} \quad (<: -\text{BOUNDED-TYPE})$$

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

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

Fig. 19. Member subtyping

are finally composed into the types of actual run-time 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 $\text{TCA}_{\text{EXPAND-TRAIT}}^{expand\text{-}this}$.

The informal rules $\text{TCA}_{\text{CALL}}^{expand\text{-}this}$ and $\text{TCA}_{\text{CALL-THIS}}^{expand\text{-}this}$ from the paper are merged into a single rule $\text{TCA}_{\text{CALL}}^{expand\text{-}this}$ 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 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 run-time 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 non-trivial 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 shaded precondition 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.

In rule $\text{TCA}_{\text{CALL-ABSTRACT}}^{expand\text{-}this}$, 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 ($p.\mathbf{type} \ni \mathtt{type}_n A$).[16] Here, the difficulty is that the subtyping rules in Scala

---

[16] In every member declaration such as $\mathtt{type}_n A$, the subscript n is a unique label that is used in FS_{alg} to give an identity to that particular declaration. The full syntax of FS_{alg} member declarations is presented in the Appendix.

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

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

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

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

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

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

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

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

Fig. 20. $\mathrm{TCA}^{expand\text{-}this}$ formalized for FS_{alg}.

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 $\mathrm{TCA}^{expand\text{-}this}_{\mathrm{EXPAND\text{-}TYPE}}$, $\mathrm{TCA}^{expand\text{-}this}_{\mathrm{EXPAND\text{-}TRAIT}}$, and $\mathrm{TCA}^{expand\text{-}this}_{\mathrm{EXPAND\text{-}TRANS}}$, which generate the set $expand(p.A)$ of all concrete types that are possibly assigned to the abstract type $p.A$. The key complication here is that the call site $s.a(\bar{t})$ 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 $\mathrm{TCA}^{expand\text{-}this}_{\mathrm{EXPAND\text{-}BASE}}$, 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 $\mathtt{type}_n A$, 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 $expand(\mathtt{type}_n A)$ 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 $\mathtt{type}_n A$, we define a new relation $T \ni^{\cup} \mathtt{type}_n A$ to hold whenever there exists a supertype S of T such that $S \ni \mathtt{type}_n A$.

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 $\mathtt{type}_n A$ implicitly, by mixing in the trait that defines $\mathtt{type}_n A$ with another trait that contains a member trait $\mathtt{trait}_{n'} A$ of the same name A. This case is handled by rule $\mathrm{TCA}^{expand\text{-}this}_{\mathrm{EXPAND\text{-}TRAIT}}$. 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 $\mathrm{TCA}^{expand\text{-}this}_{\mathrm{EXPAND\text{-}TRANS}}$.

C   Correctness Proof

The definition of FS_{alg} [7] 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 $\mathrm{TCA}^{expand\text{-}this}$ relies on the soundness of the FS_{alg} type system, we consider it outside the scope of this paper type to prove type soundness for FS_{alg}. Instead, we prove that $\mathrm{TCA}^{expand\text{-}this}$ 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.

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 $T <: S$. 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. Our definition of type soundness can assert that 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 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 U' and T' such that $p : U'$ and $x : T'$,*
   (e) *if $U' \ni \mathbf{type}_n A$, then $T' \ni^\cup \mathbf{type}_n A$ and $T' \ni \mathbf{type}_{n'} A = S'$, 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 notions of type soundness of FS_{alg} that our correctness proof of $\mathrm{TCA}^{expand\text{-}this}$ 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* $\mathbf{val}\,x = \mathbf{new}\,T$; t, *that the type T cannot be a singleton type of the form $p.\mathbf{type}$ 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 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 10. 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.*

Lemma 1. *Suppose that when $\mathrm{TCA}^{expand\text{-}this}$ 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(τ) $\subseteq R$,*
2. *allocs(τ) $\subseteq \hat{\Sigma}$,*
3. *$s : p.A$, where A is an abstract type member in p,*
4. *$x : T$, and*
5. *$p.\mathbf{type} \ni type_n A$,*

then 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 actually occurs within the execution trace τ:

1. 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 11, 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 $\mathbf{val}\,y = \mathbf{new}\,T'$; u must also occur in the trace τ. Therefore, by the assumptions of the lemma, $(y, T') \in \hat{\Sigma}$. By Definition 1, $T' \ni^\cup type_n A$, and $T' \ni type_{n'} A = S$ for some S such that $T <<: S$. Therefore, by analysis rule $\mathrm{EXPAND}_{\mathrm{BASE}}$, $S \in expand(type_n A)$. If S is a concrete type, then $T <<: S$

implies that $T <: S$ due to Definition 1, and we are done proving the lemma. If S is an abstract type $p'.A'$, then we can repeat the same argument as above 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 EXPAND$_{\text{TRANS}}$ can then be applied: since

1. $p'.A' \in expand(\texttt{type}_n A)$,
2. $p'.A' \ni \texttt{type}_{n'''} A'$, 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.

Lemma 2. *Suppose that when* 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 $calls(\tau') \subseteq R$ and $allocs(\tau') \subseteq \hat{\Sigma}$. There are three cases to consider depending on the reduction rule used in the transition $\Sigma', t' \to \Sigma'', t''$.

1. The reduction rule may be RED-NEW. In this case, t' is of the form $\texttt{val } x = \texttt{new } T; t'''$, and $allocs(\tau'') = allocs(\tau') \cup \{T\}$. We must therefore prove that $T \in \hat{\Sigma}$. The conclusion of analysis rule TCA$_{\text{NEW}}^{expand\text{-}this}$ is $(M \in R) \Rightarrow (\texttt{val } x = \texttt{new } T; u \in \Sigma)$, where M is the method containing the allocation site, that is, our term t'. M is in R because it is in $calls(\Sigma, t \to^* \Sigma', t')$. Therefore, by the conclusion of rule TCA$_{\text{NEW}}^{expand\text{-}this}$, 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 $calls(\tau'') = calls(\tau') \cup \{M'\}$, 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 $\bar{t}$ reduced to $\overline{y}$ in the trace τ'. The overall plan is to show that either rule $\text{TCA}_{\text{CALL}}^{expand\text{-}this}$ or $\text{TCA}_{\text{CALL-ABSTRACT}}^{expand\text{-}this}$ applies to assert that $c \mapsto M'$, and then rule $\text{TCA}_{\text{REACHABLE}}^{expand\text{-}this}$ concludes that $M' \in R$. We examine each of the preconditions of rules $\text{TCA}_{\text{CALL}}^{expand\text{-}this}$ and $\text{TCA}_{\text{CALL-ABSTRACT}}^{expand\text{-}this}$ 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 $\mathtt{val}\, x = \mathtt{new}\ T; t'''$, 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) $T \ni M'$, $M' \equiv \mathtt{def}_n a(\overline{y : S'}) : T'(= t')^?$

   This is given by Assumption 1.

(d) (for $\text{TCA}_{\text{CALL}}^{expand\text{-}this}$) $T <: S$, where $s : S$

   When the type S of s is a concrete type, this is given by Assumption 1.

(e) (for $\text{TCA}_{\text{CALL}}^{expand\text{-}this}$) $(S \ni M \wedge s \equiv \varphi) \Rightarrow (T \ni M)$

   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 $abstract \mapsto$) $T <: S$, $S \in expand(\mathtt{type}_n A)$, $p.\mathbf{type} \ni \mathtt{type}_n A$, $s : p.A$

   We need to ensure that these conditions are satisfied when the type $p.A$ of s is an abstract type. Lemma 1 applies to trace τ', and shows that the analysis infers a type $S \in expand(\mathtt{type}_n A)$, where $\mathtt{type}_n A$ is the declaration of A that is a member of $p.\mathbf{type}$, such that $T <: S$.

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

Theorem 1. *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 $\text{TCA}^{expand\text{-}this}$ 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 non-empty, let τ' be the sub-trace of τ consisting of all except the last reduction step. If the induction hypothesis holds for τ', then by Lemma 2, the induction hypothesis also holds for τ. Therefore, by induction, the theorem holds for every execution trace τ.

Corollary 1. *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 $\text{TCA}^{expand\text{-}this}$ 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 1) and following the same reasoning as in Case 2 (for rule RED-METHOD) of the proof of Lemma 2.