Abstract

The performance of contemporary object oriented languages depends on optimizations such as devirtualization, inlining, and specialization, and these in turn depend on precise call graph analysis. Existing call graph analyses do not take advantage of the information provided by the rich type systems of contemporary languages, in particular generic type arguments. Many existing approaches analyze Java bytecode, in which generic types have been erased. This paper shows that this discarded information is actually very useful as the context in a context-sensitive analysis, where it significantly improves precision and keeps the running time small. Specifically, we propose and evaluate call graph construction algorithms in which the contexts of a method are (i) the type arguments passed to its type parameters, and (ii) the static types of the arguments passed to its term parameters. The use of static types from the caller as context is effective because it allows more precise dispatch of call sites inside the callee.

Our evaluation indicates that the average number of contexts required per method is small. We implement the analysis in the Dotty compiler for Scala, and evaluate it on programs that use the type-parametric Scala collections library and on the Dotty compiler itself. The context-sensitive analysis runs 1.4x faster than a context-insensitive one and discovers 20% more monomorphic call sites at the same time. When applied to method specialization, the imprecision in a context-insensitive call graph would require the average method to be cloned 22 times, whereas the context-sensitive call graph indicates a much more practical 1.00 to 1.50 clones per method.

We applied the proposed analysis to automatically specialize generic methods. The resulting automatic transformation achieves the same performance as state-of-the-art techniques requiring manual annotations, while reducing the size of the generated bytecode by up to 5×.

Categories and Subject Descriptors F.3.2 [Semantics of Programming Languages]: Program analysis; D.3.4 [Programming Languages]: Processors—Compilers, Optimization; D.3.3 [Programming Languages]: Language Constructs and Features—Classes and objects, Polymorphism

Keywords call graphs, parametric polymorphism, static analysis, specialization

1. Introduction

Modern programming languages support modularity and scalability using abstraction facilities such as generic methods, interfaces and abstract type members. Unfortunately, these abstractions incur important performance costs. To achieve good performance, language implementations depend on compiler optimizations to eliminate abstractions. When the code to be optimized spans multiple methods, compilers first devirtualize, inline, or specialize the methods before other optimizations can be applied. These initial transformations require interprocedural information. A call site can be devirtualized if it is monomorphic: it is known to dispatch to only one specific method at run time. A method can be inlined into its caller after the call site has been devirtualized. A method can be specialized if the compiler has information about the values or types with which it will be called. In this paper, we propose and evaluate a call graph analysis that is especially effective for devirtualization, for specialization, and for both of these transformations applied together.
Analysis of call targets has long benefited from static types. Class hierarchy analysis (Dean et al. 1995) relies entirely on the static types of receivers to determine call targets. In propagation-based points-to analysis for Java (which is used in precise call graph construction algorithms), it has long been recognized that filtering points-to sets using static type information is critical for precision and efficiency (Lhoták and Hendren 2003).

Existing approaches to call graph construction do not take full advantage of the information provided by the type systems of modern programming languages. Most recent work in the context of object oriented languages targets Java bytecode. When Java programs are compiled to bytecode, generic type parameters and arguments are erased, so they are not available to bytecode-based analyses. In this paper, however, we show that this discarded type information is actually very useful: it enables us to construct more precise call graphs efficiently to enable devirtualization, and it provides the information necessary for specialization.

An interprocedural analysis is context-sensitive if it analyzes each method multiple times in different contexts. Ideally, the static contexts are selected so that invocations of the method with dissimilar run-time behaviours are abstracted by different analysis contexts, enabling the analysis to focus on each behaviour precisely. In the specific case of a call graph analysis, it is possible that a call site dispatches to multiple target methods overall, but is monomorphic in each specific analysis context. Unfortunately, in many analyses, the number of contexts often grows very large. As a result, the analysis becomes expensive and its output large, which makes client analyses expensive as well.

Our novel insight is that static type arguments, which have been erased in most previous work, are actually very effective contexts for call graph construction. Often, the static type of the receiver at a call site is a type parameter of the method in which the call site appears, or of the enclosing class of that method. Analyzing the enclosing method separately for each argument type provides static type information that is often precise enough to resolve the call to a single target method (i.e., monomorphically). Moreover, the number of contexts in which the average method needs to be analyzed remains small. At a given call site (in a given context), only one static type is passed as the argument for each type parameter, so the number of contexts grows only when a type parameter is really used with different type arguments in multiple places in the program.

Call graphs contain the information needed for devirtualization, but building them with static types as context also provides the information needed for specialization. One common specialization criterion is to create distinct implementations of polymorphic methods, and of methods in generic classes, for each type argument with which the method or containing class is instantiated. The context-sensitive call graph provides exactly the set of type arguments with which each parameter may be instantiated, and this is the set of specialized methods that need to be generated.

The context-sensitive call information is well suited to devirtualization after specialization has been applied. In particular, the context-sensitive call graph may say that a call site is monomorphic, but only in some specific context. Since the analysis contexts correspond directly to the specialized method implementations, this is exactly the information that is needed to know that a call site in a specific specialized implementation can be devirtualized.

We intend our analyses to be included in production compilers, rather than being limited to research prototypes. This is feasible thanks to the efficiency and relative simplicity of the proposed analyses. In our experiments, the context-sensitive analysis runs faster than a baseline context-insensitive analysis thanks to its higher precision.

The correctness of the approach does not depend on a closed-world assumption about the analysis. If the program is later extended and new type arguments become possible, the generated code falls back to the original, generic (unspecialized) version of the method. Similarly, devirtualized or inlined monomorphic call sites can contain fall-back calls to the original methods in case the call site is invoked with an unexpected type at runtime.

Our use of static types as contexts is distinct from the dynamic type tags used as contexts in the “type-sensitive” analysis of (Smaragdakis et al. 2011, 2014). That analysis traces the flow of objects (abstracted by their dynamic type tags) from allocation sites along dataflow paths through the program all the way to each call site, and then analyses the target of the call site in a separate context for each possible dynamic type of the receiver (and optionally of the other arguments (Agesen 1995)). In contrast, the context that we propose is formed from the static types of the receiver and arguments that are available locally at the call site. Unlike dynamic type tags, the static type does not need to be propagated from the allocation site to the call site. Moreover, a given call site may be reached by objects of many different runtime types, which gives rise to many contexts for the target method in the “type-sensitive” analysis. In contrast, only a single static type argument is passed for each type parameter, so the number of contexts in our proposed analysis remains small.

This paper makes the following contributions:
— The paper proposes two extensions to call graph construction algorithms for Scala. In the first extension, we define the contexts in which a method is analyzed using the actual (but static) type arguments that are substituted for the generic type parameters of the method. In the second extension, we further refine the contexts by replacing the declared types of the method’s term parameters with more precise subtypes, taken from the static types of actual arguments. Different combinations of choices of possible subtypes define distinct contexts. In the case of type class instances passed
using Scala's implicit mechanism, our analysis can often specialize the parameter type to a singleton type that represents one specific instance of the type class.

— The paper presents experimental results showing that (i) the proposed context-sensitive analyses are around 1.4x faster than a context-insensitive analysis on substantial programs, (ii) the context-sensitive analyses discover significantly more monomorphic call sites, and (iii) the precision due to context-sensitivity reduces the number of times that the average method would have to be specialized from 22 to a much more reasonable 1.00 to 1.50 times.

— The paper evaluates the application of the proposed analyses to specialization of generic type arguments. Code specialized automatically using the analysis results achieves the same runtime performance as code specialized according to annotations provided by experts manually. Moreover, the automatic specialization generates substantially less bytecode than specialization guided by manual annotations.

The rest of the paper is organized as follows. In Section 2, we present an example program that motivates the need for specialization and therefore for precise call graphs. In Section 3, we provide a background discussion of the current state-of-the-art call graph construction algorithm for Scala, the TCA expand-this\(^1\) analysis of (Ali et al. 2014, 2015). We define our context-sensitive analyses in Section 4. Section 5 presents and discusses our experimental results. We discuss related work in Section 6, and conclude in Section 7.

2. Motivation

```scala
implicit def Iterable[T](implicit ord: Ordering[T]): Ordering[Iterable[T]] =
  new Ordering[Iterable[T]] {
    def compare(x: Iterable[T], y: Iterable[T]): Int = {
      val xe = x.iterator
      val ye = y.iterator
      while (xe.hasNext && ye.hasNext) {
        val res = ord.compare(xe.next(), ye.next())
        if (res != 0) return res
      }
      Boolean.compare(xe.hasNext, ye.hasNext)
    }
  }
```

Listing 1. Running example from scala.math.Ordering.

We motivate the need for a more precise call graph abstraction using the example method in Listing 1. This method is taken from the scala.math.Ordering class in the Scala standard library. Given any ordering ord for the type T, the method implicitly generates a lexicographic ordering for the type Iterable[T]. Since the compare method on Line 3 is called many times at run time, in loops, it is beneficial to specialize and inline the call sites within it as much as possible, especially those within the while loop on Line 7. In particular, a high-performance code generator should specialize the compare method for each value ord for which it is generated.

A context-insensitive call graph will contain a path to the compare method on Line 3 from the Arrays.sort method in the Java standard library. Therefore, for every type T that is ever sorted anywhere in the whole program, a sound analysis should find that an object of every such type could reach the parameters x and y of compare. In particular, in a large program, this is likely to include most of the possible subtypes of Iterable. In the Scala standard library, the trait Iterable has 214 concrete subtypes.

As a result, the calls to x.iterator and y.iterator on lines 4 and 5 will be highly polymorphic and not inlineable.

As a consequence, the sets of possible types of xe and ye will be highly imprecise. There are 44 concrete subtypes of Iterator in the Scala standard library.

Therefore, the calls to xe.hasNext and ye.hasNext on Line 7 will also be highly polymorphic and infeasible to inline, as well as the calls to xe.next() and ye.next() on Line 8. The bodies of these four methods are usually small, and are called for every element of the iterables, so they need to be inlined to achieve good performance.

Finally, the call to ord.compare on Line 8 is statically considered to be dispatched to every implementation of Ordering[T] that reaches the ord parameter. Therefore, this call is also highly polymorphic in a context-insensitive call graph.

Let us consider how the static polymorphism could be reduced using context sensitivity (or, equivalently, specialization). We will illustrate this with the example client program in Listing 2.

```scala
def lexicographicSort[T](a: Iterable[T]*)(implicit o: Ordering[T]) = a.sorted

lexicographicSort("world", "Hello")
```

Listing 2. Example program that uses the compare method from Listing 1.

The snippet defines a generic method lexicographicSort that creates a sorted list of values of type Iterable[T] by calling the sorted method of SeqLike. The * after the Iterable[T] parameter type indicates that the method takes a variable number of parameters, each of type Iterable[T]. The LexicographicSort method is called with two strings on Line 17.

Type inference and implicit resolution in the early stages of the Scala compiler desugar the program as shown in Listing 3.

One of the most serious impediments to good performance of the compare method is the need to box and unbox values of primitive Java types such as char. The bytecode

\(^1\) TCA stands for Trait Composition Analysis.
version of the Iterator.next method has a return type of Object. It is incompatible with primitive types, so each char that it returns must be boxed in a Character. Inside the compare method of Ordering.Char, the Character must again be unboxed into a primitive Char.

```scala
def lexicographicSort[T](a: Seq[Iterable[T]]):
  implicit o: Ordering[T] = a.sorted
  lexicographicSort[Char](
    Predef.wrapRefArray[WrappedString](
      new Array(
        Predef.wrapString("world"),
        Predef.wrapString("Hello")
      )
    )
  )(Ordering.Char)
```

Listing 3. Desugared version of example program from Listing 2.

Our first proposed improvement to the call graph is to analyze the entire outer Iterable method from Listing 1 separately in the context of each possible type argument with which the type parameter T is instantiated. In this example, T is specialized to Char. As a result, the type of xe and ye becomes Iterator[Char], and the calls to xe.next() and ye.next() in Line 8 can be redirected to versions of the methods that return a primitive Char without boxing. Similarly, the type of ord becomes Ordering[Char], so the call of ord.compare can be redirected to a version with primitive Char parameters that do not need to be unboxed.

Thus, all of the boxing and unboxing can be removed from the while loop.

Our second proposed improvement is to analyze methods separately in the contexts of more precise types of their parameters available at the call site. In our running example, we can determine that when T is Char, the compare method is only called with a small number of concrete types of Iterables. In particular, we can analyze it specifically in the context in which both of its parameters are of the type WrappedString that is returned by Predef.wrapString. The calls to x.iterator and y.iterator in Lines 4 and 5 become monomorphic, which enables the analysis to give a precise type to xe and ye. As a result, the calls to hasNext and next() become monomorphic as well. We can now rewrite the known monomorphic calls to target specific statically known versions of their target methods, which makes it easy for the Java JIT compiler to inline and aggressively optimize them. The resulting optimized code is a simple loop over the arrays underlying the implementations of the strings being compared, much like the loop that one would write in C to compare two strings.

3. Background

The existing state-of-the-art in call graph construction for Scala is the TCA expand-this algorithm of (Ali et al. 2014, 2015). To enable comparison of our results with previous work, we formulate our improvements as extensions of this existing framework. In this section, we present this baseline framework.

The main inference rules of the formulation are shown in Figure 1. The algorithm iterates the rules until a fixed
point is reached, using worklists to keep track of new facts and to determine which rules need to be reevaluated. The set \( R \) keeps track of the methods reachable from the entry points through the call graph constructed so far. The set \( \Sigma \) keeps track of the types of objects that may be allocated in these reachable methods. The rule \( TCA_{\text{expand-ths}} \) initializes \( R \) with the main entry point. The rule \( TCA_{\text{expand-ths}} \) finds object instantiations in reachable methods and adds the types to \( \Sigma \).

The rule \( TCA_{\text{expand-ths}} \) resolves a call site \( e.m[\sigma]() \) using the static type of the receiver \( e \) to determine all possible target methods \( M' \). The rule \( TCA_{\text{abstract-call}} \) handles the specific case of a call site at which the static type \( T \) of the receiver \( e \) is an abstract type. In this case, the \( TCA_{\text{expand-ths}} \) algorithm uses the function \( \text{expand}(T) \) to determine the possible concrete types with which \( T \) could be instantiated. The \( \text{expand}(T) \) function is computed by additional inference rules that find all of the concrete types with which the abstract type \( T \) could ever be instantiated. We do not show those rules here; for details, refer to (Ali et al. 2014, 2015). The rule \( TCA_{\text{expand-ths}} \) is a variation of \( TCA_{\text{call}} \) that is more precise in the specific case when the receiver of the call is the this pointer in the caller (i.e. the receiver of the callee is the same object as the receiver of the caller). In this case, the rule adds precision using the additional precondition that the caller \( M \) must also be a member of some type \( C \) that the callee \( M' \) is a member of. The rule \( TCA_{\text{local-call}} \) handles calls to local functions that are nested inside some other function rather than being members of a class. This rule was not given explicitly by (Ali et al. 2014, 2015), but we have added it here for completeness. Calls to such functions do not have a receiver, and they are not dispatched dynamically; the method specified at the call site is the exact method that is executed.

4. Algorithms

4.1 \( TCA_{\text{types}} \): Propagation of Type Arguments

We now introduce the first extension to the TCA algorithm. The main idea is to construct a context-sensitive call graph in which each context for a given method is a substitution of concrete types for the type parameters of that method. Specifically, the elements of the set \( R \), which were the reachable methods in TCA, now become pairs of a reachable method and a type substitution. The inference rules for the extended algorithm are shown in Figure 2. Changes from the original algorithm are shaded.

The rule \( TCA_{\text{types}} \) pairs the main method with the empty substitution \( \emptyset \), since the entry point of the program has no type parameters.

The rule \( TCA_{\text{types}} \) iterates over all reachable method-substitution pairs, ignores the substitution, and adds the types instantiated in each reachable method to \( \Sigma \), as in the original algorithm.

In the rule \( TCA_{\text{types}} \), for each reachable pair \( (M, \sigma) \), where \( M \) is a method and \( \sigma \) is a substitution, \( \sigma \) is applied to the static type of the receiver \( e \). We use the postfix notation StaticType\( (e) \sigma \) to denote substitution application. From the actual type arguments passed to the callee \( M' \) at the call

---

Figure 2. Propagation of type arguments
site, we define the substitution \( \sigma' \) that replaces each type parameter of \( M' \) with the argument that is passed for it. In the conclusion of the \( \text{TCA}^{\text{type}} \) rule, the caller’s context substitution \( \sigma \) is composed with the call site substitution \( \sigma' \). As a result, if \( \sigma' \) uses one of the type parameters of the caller, it will be replaced using \( \sigma \) with the concrete type that is instantiated within in the specific caller context. We use the notation \( \sigma' \sigma \) to denote substitution composition. We restrict the resulting composed substitution to only the type parameters of \( M' \), formally \( \text{dom}(\sigma') \). We use the notation \( \sigma' |_{\text{dom}(\sigma')} \) to denote this restriction.

We apply similar modifications to the rules \( \text{TCA}^{\text{expansion\-this\-call}} \) and \( \text{TCA}^{\text{abstract\-call}} \) to obtain the new rules \( \text{TCA}^{\text{types}} \) and \( \text{TCA}^{\text{abstract\-call}} \).

Because the set of possible types is unbounded, the set of reachable methods paired with type substitutions could grow without bound. In particular, this happens in the case of polymorphic recursion in the following example:

```scala
def foo[A](a: List[A], d: Int): List[_] = 
  if (d == 0) a
  else foo(a.zip(a), d - 1)
```

The method foo in context \([A \mapsto \text{Int}] \), calls foo in context \([A \mapsto ([\text{Int}, \text{Int}]) \), which later calls foo in context \([A \mapsto ([\text{Int}, \text{Int}], [\text{Int}, \text{Int}])]) \), and so on. To ensure the termination of call graph construction, we define a limit for the number of contexts under which each method is considered. If the limit is exceeded, then instead of creating a new context \((M, [N_i \mapsto T_i)])\), we loosen the precision of the last created context for the same method \((M, [N_i \mapsto T'_i])\) by replacing each type in it with the least upper bound of the type in the old context and the type in the new context: \((M, [N_i \mapsto \text{lub}(T_i, T'_i)])\). The loosened context conservatively overapproximates the types in both the old, last created context for the method and the new context that we intended to create.

We did not encounter any cases of such unbounded growth in any of the benchmark programs that we evaluated.

### 4.2 Propagation of Outer Type Parameters

In the previous section, the context of each method substituted concrete types only for the direct type parameters of that method. For even greater precision, we can extend the context with the type parameters of the classes and methods that the method is nested within. Specifically, in our implementation, each element of \( \Sigma \) is not just an instantiated type \( C \), but a pair \( (\sigma, C) \), where \( \sigma \) is a substitution that assigns a concrete type to every type parameter that is in scope at the program location where \( C \) is instantiated.

An equivalent method to achieve the same precision is to split the analysis into two phases. The first phase transforms the code using a transformation similar to lambda lifting (Johnsson 1985), but applied to type parameters. Specifically, whenever a class or method has some type parameter \( T \) that can be implicitly used in methods nested within it, we add \( T \) as an explicit type parameter to each of those nested methods, and pass it explicitly at every call site. The second phase is then to perform the simple analysis described in the previous section. For performance reasons, our implementation uses the first approach of associating a substitution with each instantiated type. For clarity of presentation, our description in this paper follows the second approach, which decouples the issue of instantiating parameters of enclosing classes and methods from the analysis itself.

We illustrate the transformation with the following example program, in which method bar is nested in method foo, which is nested in class C:

```scala
class C[T] {
  def foo[U](t: T, u: U) = {
    if (d == 0) a
    else foo(a.zip(a), d - 1)
  }
}
```

The above program would be transformed as follows:

```scala
class C[T] {
  def foo[T2, U](t: T2, u: U) = {
    if (d == 0) a
    else foo(a.zip(a), d - 1)
  }
}
```

The type parameter \( T \) of class \( C \) has been explicitly added to the methods foo and bar nested within it as \( T2 \) and \( T3 \). The type parameter \( U \) of method foo has been explicitly added to the method bar that is nested within it as \( U2 \).

Type parameters need to be passed explicitly when an outer method calls an inner one. When a given type parameter comes from a method in the original program, it is available at the call site as an explicit parameter of the caller method in the transformed program: for example, in the call of bar from foo, type parameters \( T2 \) and \( U \) of foo are passed as arguments for the parameters \( T3 \) and \( U2 \) of bar. When a given type parameter comes from a class in the original program, it is also available at the call site as an argument in the type of the receiver: for example, in the call to foo, the type argument \( \text{Int} \) in the type \( C[\text{Int}] \) of the receiver determines the type argument to be passed for the parameter \( T2 \) of foo.

Note that the erasure of both the original and the transformed program is the same, so the runtime behavior is left unchanged.

In addition to type parameters, we also transform abstract type members of each class in the same way, turning them into explicit type parameters of all methods nested inside the class. Consider the following program:
The program gets transformed to:

```java
abstract class Buffer {
  type U
  type T <: Seq[U]
  def elements: T
  def length = elements.length
}
class Buffer123 extends Buffer {
  type U = Int
  type T = List[Int]
  def elements = List(1, 2, 3)
}
Buffer123.length()
```

A consequence of this transformation is that the body of each method refers only to type parameters defined on the method itself, and does not refer to any type parameters or type members of outer enclosing classes or methods. As a result, on the transformed program, the substitution context defined in the previous section now provides arguments for all the type parameters of each method, including those that came indirectly from outer classes and methods in the original program.

It is now easy to prove inductively that the range of every substitution \( \sigma \) that ever appears in a pair in \( R \) consists only of fully instantiated types (which do not contain any type parameters). Suppose that this is true of the substitution context \( \sigma \) of a method \( M \) that contains a call site \( \pi, e.M[\sigma'](\cdot) \). The only type variables used in the argument substitution \( \sigma' \) are the direct type parameters of \( M \). The context substitution \( \sigma \) provides fully instantiated types for all of these type parameters. Therefore, when \( \sigma' \) and \( \sigma \) are composed, the range of the composed substitution contains only fully instantiated types. It is this composed substitution with fully instantiated types that becomes the new context for the target method called by the call site.

Therefore, the static type of the receiver of a call, \( \text{StaticType}(e)\sigma \), is never abstract after the caller-context substitution \( \sigma \) has been applied to it. The rule \( \text{TCA}_{\text{abstract-call}} \) is thus never needed and can be removed from the algorithm, together with the rules for computing the \( \text{expand}() \) sets for abstract types.

### 4.3 \( \text{TCA}_{\text{types-terms}} \): Propagation of Term Argument Types

It is very common for the receiver at a call site to be one of the (term) parameters of the method containing the call site. The implicit receiver parameter this is the most common such receiver, but other parameters are also common. As an example, consider the following code:

```java
def internalHashCode[T](el: T, nullRep: Object) =
  if (el != null)
    el.hashCode
  else
    nullRep.hashCode
internalHashCode[Int](42, "null")
```

The receivers \( e1 \) and \( nullRep \) of the calls to \( \text{hashCode} \) are both parameters of \( \text{internalHashCode} \). When the type of the receiver is itself a type variable of the caller, the propagation of type arguments that we have described above helps to resolve the call precisely. In the example, the type of \( e1 \) is the type parameter \( T \), which the context substitution instantiates to \( \text{Int} \), so we know that the target of \( e1.\text{hashCode} \) is the implementation of \( \text{hashCode} \) in \( \text{Int} \). However, in the call \( \text{nullRep.\text{hashCode}} \), we need to assume that the runtime type of the receiver \( \text{nullRep} \) could be any subtype of \( \text{Object} \). To further improve precision, the analysis can be extended further to propagate the type of the argument from the call site of \( \text{internalHashCode} \), which is \( \text{String} \), into the context in which \( \text{internalHashCode} \) is analyzed. As a result, the analysis could then determine that the call \( \text{nullRep.\text{hashCode}} \) calls only the \( \text{String} \) implementation of \( \text{hashCode} \).

To implement this precision improvement in our call graph construction algorithm, we further extend the method contexts contained in the set \( R \). Each element of \( R \) becomes a triple that contains a reachable method \( M \) and a type parameter substitution \( \sigma \) as before, and, in addition, a list \( \pi \) of more precise types for the term parameters of \( M \) (including the implicit this receiver parameter).

The inference rules for the extended algorithm are shown in Figure 3. Changes compared to Figure 2 are shaded. The \( \text{StaticType} \) function is extended to take a list \( \pi \) of more precise parameter types. If \( e \) is a parameter of \( M \), then \( \text{StaticType}(\pi, e) \) returns the more precise type of \( e \) given by \( \pi \); otherwise it just returns the same static type of \( e \) as in the previous analyses. We also extend \( \text{StaticType} \) to map over a sequence of terms and return a sequence of their types. The last premise of the \( \text{TCA}_{\text{types-calls}} \) rule uses \( \text{StaticType} \) to get the precise types of the arguments passed at the call site. The substitution \( \sigma \) is applied to these types. These precise types
We implemented the TCA\textsubscript{types-terms} analysis of Ali et al. (2014, 2015) and our two extensions TCA\textsubscript{types} and TCA\textsubscript{types-terms} on top of the Dotty compiler\footnote{https://github.com/lampepl/dotty}, a new compiler for the future evolution of the Scala language. Although Dotty is not yet finished, it is not a research prototype: it is intended to eventually replace the current nsc to become the standard production-quality compiler for Scala. We tested our implementation on the full test suite of Dotty, which includes 1403 Scala programs. To the best of our knowledge, our analyses are then included in the context that is added to $R$ in the conclusion of the rule.

**5. Evaluation**

We implemented the TCA\textsubscript{expand-this} analysis of Ali et al. (2014, 2015) and our two extensions TCA\textsubscript{types} and TCA\textsubscript{types-terms} on top of the Dotty compiler\footnote{http://docs.scala-lang.org/sips/pending/trait-parameters.html}, a new compiler for the future evolution of the Scala language. Although Dotty is not yet finished, it is not a research prototype: it is intended to eventually replace the current nsc to become the standard production-quality compiler for Scala. We tested our implementation on the full test suite of Dotty, which includes 1403 Scala programs. To the best of our knowledge, our analyses soundly handle the entire Scala language dialect supported by Dotty, including Dotty-specific extensions to Scala such as trait parameters\footnote{http://docs.scala-lang.org/sips/pending/repeated-byname.html} and repeated by name parameters\footnote{http://docs.scala-lang.org/sips/pending/repeated-byname.html}.

The analysis runs after the type checker stage of Dotty. At this stage, all expressions have their original, unerased and unsimplified Scala types. This means that our implementation correctly handles types that may contain generic types and path dependent types (Odersky 2014, §3.5). When the analysis requires subtyping checks, we use the implementation of subtype testing included in the Dotty compiler.

In this section, we first evaluate the TCA\textsubscript{types-terms} analysis implemented in Dotty, and then show how it can be used for program performance.

**5.1 Analysis Evaluation**

We evaluated our implementation on the nine Scala programs listed in Table 1. The first six programs were selected to exercise the Scala collections library, which is implemented in a very generic style with multiple layers of abstraction. The collections library is also highly polymorphic: for example, it contains 214 named subclasses of \texttt{Iterable}. The next two benchmarks are moderately-sized applications implemented in idiomatic Scala. The largest benchmark is the parser and type checker of the Dotty compiler itself. The Dotty compiler is still under development, and only recently became able to bootstrap itself. More development of the Dotty compiler is necessary before it can compile more mainstream Scala applications.

To construct each call graph, we provided all of the dependencies written in Scala as source code to the analysis. All Scala programs also implicitly depend on the Java Standard Library, which is in the form of Java bytecode that our implementation does not analyze. We made conservative assumptions about the effects of the Java library, and used the Separate Compilation Assumption (Ali and Lhoták 2012; Ali and Lhoták 2013) to construct a sound partial call graph for the parts of the program that were written in Scala and therefore available for analysis. The only methods of the
<table>
<thead>
<tr>
<th>Program</th>
<th>Algorithm</th>
<th># Instantiated classes</th>
<th># Classes with at least one reachable method</th>
<th># Reachable methods</th>
<th># Reachable contexts</th>
<th># Maximum contexts per method</th>
<th># Discovered specializations</th>
<th>Code growth factor</th>
<th>% monomorphic call sites</th>
<th>% bimorphic call sites</th>
<th>% megamorphic call sites</th>
<th>Running time, seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>List creation</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>149</td>
<td>64</td>
<td>207</td>
<td>207</td>
<td>1</td>
<td>3469</td>
<td>16.75</td>
<td>80.2</td>
<td>7.0</td>
<td>12.8</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>117</td>
<td>33</td>
<td>90</td>
<td>90</td>
<td>1</td>
<td>90</td>
<td>1.00</td>
<td>93.0</td>
<td>4.7</td>
<td>2.3</td>
<td>1.30</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>117</td>
<td>31</td>
<td>83</td>
<td>83</td>
<td>1</td>
<td>83</td>
<td>1.00</td>
<td>95.4</td>
<td>2.3</td>
<td>2.3</td>
<td>1.32</td>
</tr>
<tr>
<td>List &amp; Vector creation</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>152</td>
<td>79</td>
<td>268</td>
<td>268</td>
<td>1</td>
<td>6358</td>
<td>24.73</td>
<td>73.4</td>
<td>4.1</td>
<td>22.4</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>130</td>
<td>36</td>
<td>95</td>
<td>114</td>
<td>2</td>
<td>114</td>
<td>1.20</td>
<td>86.0</td>
<td>2.1</td>
<td>11.9</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>130</td>
<td>34</td>
<td>90</td>
<td>138</td>
<td>4</td>
<td>112</td>
<td>1.24</td>
<td>88.1</td>
<td>4.5</td>
<td>7.5</td>
<td>1.41</td>
</tr>
<tr>
<td>List create and sort</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>157</td>
<td>65</td>
<td>209</td>
<td>209</td>
<td>1</td>
<td>3919</td>
<td>18.75</td>
<td>77.6</td>
<td>6.4</td>
<td>16.0</td>
<td>0.77</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>126</td>
<td>34</td>
<td>92</td>
<td>92</td>
<td>1</td>
<td>92</td>
<td>1.00</td>
<td>87.2</td>
<td>9.6</td>
<td>3.2</td>
<td>1.54</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>126</td>
<td>34</td>
<td>89</td>
<td>147</td>
<td>2</td>
<td>89</td>
<td>1.00</td>
<td>89.4</td>
<td>8.5</td>
<td>2.1</td>
<td>1.58</td>
</tr>
<tr>
<td>List &amp; Vector create and sort</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>170</td>
<td>83</td>
<td>357</td>
<td>357</td>
<td>1</td>
<td>7725</td>
<td>21.64</td>
<td>72.4</td>
<td>2.4</td>
<td>25.2</td>
<td>2.30</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>142</td>
<td>39</td>
<td>115</td>
<td>140</td>
<td>2</td>
<td>140</td>
<td>1.21</td>
<td>86.2</td>
<td>3.9</td>
<td>9.8</td>
<td>1.64</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>142</td>
<td>37</td>
<td>109</td>
<td>147</td>
<td>5</td>
<td>131</td>
<td>1.20</td>
<td>89.2</td>
<td>2.6</td>
<td>8.2</td>
<td>1.47</td>
</tr>
<tr>
<td>List create, sort and print</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>171</td>
<td>68</td>
<td>212</td>
<td>212</td>
<td>1</td>
<td>4146</td>
<td>19.56</td>
<td>78.4</td>
<td>4.1</td>
<td>17.4</td>
<td>1.29</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>131</td>
<td>37</td>
<td>95</td>
<td>95</td>
<td>1</td>
<td>95</td>
<td>1.00</td>
<td>87.8</td>
<td>9.2</td>
<td>3.1</td>
<td>5.43</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>131</td>
<td>35</td>
<td>92</td>
<td>206</td>
<td>6</td>
<td>92</td>
<td>1.00</td>
<td>89.8</td>
<td>8.2</td>
<td>2.0</td>
<td>3.25</td>
</tr>
<tr>
<td>lexicographicSort</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>182</td>
<td>88</td>
<td>293</td>
<td>293</td>
<td>1</td>
<td>5529</td>
<td>18.87</td>
<td>72.7</td>
<td>2.8</td>
<td>24.5</td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>134</td>
<td>41</td>
<td>102</td>
<td>104</td>
<td>2</td>
<td>104</td>
<td>1.01</td>
<td>86.6</td>
<td>7.7</td>
<td>5.6</td>
<td>5.91</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>134</td>
<td>41</td>
<td>98</td>
<td>231</td>
<td>3</td>
<td>102</td>
<td>1.04</td>
<td>89.1</td>
<td>6.5</td>
<td>4.4</td>
<td>4.08</td>
</tr>
<tr>
<td>Page rank</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>229</td>
<td>92</td>
<td>341</td>
<td>341</td>
<td>1</td>
<td>12490</td>
<td>36.63</td>
<td>59.4</td>
<td>8.5</td>
<td>32.1</td>
<td>10.28</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>145</td>
<td>50</td>
<td>127</td>
<td>173</td>
<td>3</td>
<td>173</td>
<td>1.36</td>
<td>77.4</td>
<td>7.6</td>
<td>15.1</td>
<td>11.22</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>145</td>
<td>45</td>
<td>118</td>
<td>293</td>
<td>5</td>
<td>165</td>
<td>1.40</td>
<td>85.9</td>
<td>9.9</td>
<td>4.3</td>
<td>6.24</td>
</tr>
<tr>
<td>Round robin</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>189</td>
<td>76</td>
<td>252</td>
<td>252</td>
<td>1</td>
<td>6272</td>
<td>24.89</td>
<td>72.6</td>
<td>8.1</td>
<td>19.3</td>
<td>9.69</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>147</td>
<td>46</td>
<td>130</td>
<td>174</td>
<td>1</td>
<td>174</td>
<td>1.34</td>
<td>87.9</td>
<td>8.1</td>
<td>4.0</td>
<td>8.79</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>147</td>
<td>44</td>
<td>123</td>
<td>310</td>
<td>3</td>
<td>165</td>
<td>1.34</td>
<td>87.9</td>
<td>8.9</td>
<td>3.2</td>
<td>3.91</td>
</tr>
<tr>
<td>Dotty typechecker</td>
<td>TCA&lt;sup&gt;expand-this&lt;/sup&gt;</td>
<td>1028</td>
<td>822</td>
<td>10694</td>
<td>10694</td>
<td>1</td>
<td>45278</td>
<td>4.23</td>
<td>55.6</td>
<td>1.8</td>
<td>42.6</td>
<td>893.52</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types&lt;/sup&gt;</td>
<td>832</td>
<td>695</td>
<td>9347</td>
<td>14011</td>
<td>4</td>
<td>14011</td>
<td>1.50</td>
<td>82.3</td>
<td>0.6</td>
<td>17.1</td>
<td>1071.71</td>
</tr>
<tr>
<td></td>
<td>TCA&lt;sup&gt;types-terms&lt;/sup&gt;</td>
<td>832</td>
<td>629</td>
<td>8992</td>
<td>37992</td>
<td>43</td>
<td>13122</td>
<td>1.46</td>
<td>90.7</td>
<td>2.6</td>
<td>6.7</td>
<td>637.10</td>
</tr>
</tbody>
</table>

Table 1. Results of the TCA<sup>expand-this</sup>, TCA<sup>types</sup>, and TCA<sup>types-terms</sup> analyses on the benchmark programs. The first two columns specify the benchmark program and the analysis algorithm. The next three columns show the number of classes found to be instantiated, including their superclasses, classes that have at least one reachable method, and methods reachable by the analysis. The following two columns show the total number of reachable method contexts and the maximum number of such contexts per method. If every reachable method were specialized for all of the type arguments that the analysis determines may flow to its type parameters, the next two columns show the total number of such specialized methods that would be created and the factor by which this number is greater than the number of reachable methods in the original program. The next three columns show the percentage of call sites found to be monomorphic, bimorphic, and megamorphic by each analysis. For consistency, to enable comparisons between the three analyses, we take as the universe of all call sites only those in methods found to be reachable by the most precise analysis, TCA<sup>types-terms</sup>. Otherwise, the results would be confounded by the fact that each analysis discovers a different set of reachable methods and therefore a different set of reachable call sites. The final column gives the running time of the analysis.
Java standard library called by any of our benchmark programs and their Scala dependencies are the methods of the java.lang.Object and java.lang.Comparable classes.

We ran all of our experiments on a machine with a quad core 2.8 GHz Intel i7-4980HQ CPU (running in 64-bit mode) and capped available memory for experiments to 768 MB of RAM.

5.1.1 Research Questions

Our evaluation aims to answer the following Research Questions:

**RQ1.** How do the three analysis algorithms compare in terms of the precision of the call graphs that they generate?

**RQ2.** Type and term argument propagation increase the size of the set $R$ by tracking methods multiple times with different type and term arguments. How severe is the increase?

**RQ3.** How usable are the call graphs generated by the three analysis algorithms for the purposes of specialization and inlining?

**RQ4.** How many call sites can the algorithms prove to be monomorphic?

**RQ5.** How does tracking of type and term arguments affect the running time of the analysis?

5.1.2 Results

**RQ1.** Relative to $\text{TCA}^{\text{expand-this}}$, call graphs constructed by $\text{TCA}^{\text{types}}$ have 22% fewer reachable classes and 56% fewer reachable methods on average. The most significant cause of the precision improvement was that $\text{TCA}^{\text{types}}$ precisely resolved calls on generic super classes where $\text{TCA}^{\text{expand-this}}$ was imprecise. For example, a call on a $\text{Seq}[\text{T}]$ could dispatch to both $\text{List}[\text{Int}]$ and $\text{Vector}[\text{Double}]$ according to $\text{TCA}^{\text{expand-this}}$, but $\text{TCA}^{\text{types}}$ would analyze the call separately within the context of the two different type arguments.

On the Dotty typechecker, the $\text{TCA}^{\text{types}}$ call graph has 15% fewer reachable methods than the $\text{TCA}^{\text{expand-this}}$ call graph. The improvement is smaller because Dotty makes little use of the generic collections in the standard library. For example, Dotty uses its own custom tuned implementations of sets. Of 629 classes with reachable methods, only 40 are from the standard library.

On average over all of the benchmark programs, the analysis $\text{TCA}^{\text{types-terms}}$ further reduces the number of reachable methods by 5% compared to $\text{TCA}^{\text{types}}$.

The number of megamorphic call sites is, on average, 70% lower with $\text{TCA}^{\text{types}}$ than with $\text{TCA}^{\text{expand-this}}$. $\text{TCA}^{\text{types-terms}}$ further reduces the number of megamorphic call sites to 32% fewer than $\text{TCA}^{\text{types}}$.

On the Dotty type checker, $\text{TCA}^{\text{types-terms}}$ reduces the number of megamorphic call sites by 60% compared to $\text{TCA}^{\text{types}}$. The main source of this improvement is apply methods, which implement closures.

**RQ2.** We might expect that the number of reachable contexts would grow as the amount of context sensitivity is increased. In fact, due to the substantial improvement in precision and the decrease in the number of reachable methods, the average number of reachable contexts is 53% smaller in $\text{TCA}^{\text{types}}$ than in $\text{TCA}^{\text{expand-this}}$. $\text{TCA}^{\text{types-terms}}$ does generate more reachable contexts than $\text{TCA}^{\text{types}}$, but generally still fewer than $\text{TCA}^{\text{expand-this}}$.

The Dotty typechecker is a special case in this regard. It has a substantial number of closures that are passed as arguments, with multiple different closures being passed to the same method. Tracking all of these closures requires 4x as many reachable method contexts in $\text{TCA}^{\text{types-terms}}$ as there are reachable methods in $\text{TCA}^{\text{expand-this}}$.

As we mentioned in Section 4.1, it is theoretically possible for the number of contexts to grow without bound, and we must stop generating new contexts after a fixed limit has been exceeded in order for the analysis to terminate. We did not observe unbounded growth in any of the benchmark programs. To determine how to select the limit, we counted the maximum number of contexts for any given reachable method for each benchmark. The maximum number of contexts was 6 or less for all of the benchmarks, except for the special case of the Dotty typechecker. It contains a function $\text{track(String)(Closure)}$ that is used to track how many times a particular computation is performed. This function is called with 43 different closures, and term argument type propagation tracks all of them as separate contexts. Aside from this function, only 5 other functions in the Dotty typechecker are analyzed with more than 10 contexts.

**RQ3.** The call graphs generated by the three algorithms provide information about the concrete type arguments with which each type parameter in the program can be instantiated. Our intended application is to specialize each generic method for each of the type arguments that it may be called with. Methods that have been specialized in this way can be easily inlined as an additional step, either in a static optimizer or in a JIT compiler.

The type argument information provided by the context-insensitive $\text{TCA}^{\text{expand-this}}$ analysis is too imprecise to be practical for this application. It indicates that each method should be specialized 22 times on average.

Both of the context-sensitive analyses, $\text{TCA}^{\text{types}}$ and $\text{TCA}^{\text{types-terms}}$, provide much more usable information for specialization. They indicate that on average methods need to be specialized 1.50 times.

**RQ4.** Our intended applications of call graphs, specialization and inlining, apply to call sites that have only a single possible target method (are monomorphic). The precision of many other analyses such as points-to analysis and escape analysis benefits significantly from precisely knowing the targets of virtual calls. We therefore measure the ability of different algorithms to resolve each call site to a unique target method.
Adding type propagation in TCA\textsuperscript{types} substantially increases the percentage of call sites that are statically monomorphic compared to TCA\textsuperscript{expand-this}, by around 10 percentage points on small programs and by around 20 percentage points on large programs. TCA\textsuperscript{types-terms} further increases monomorphic call sites by up to 8 percentage points on the large programs.

RQ5. We might expect that the more precise context-sensitive analyses require more time than TCA\textsuperscript{expand-this}. This is indeed the case on some of the small programs that exercise the library: TCA\textsuperscript{types} takes up to 4x as long as TCA\textsuperscript{expand-this}. This is due to more complex rules that require more work to process each call site. However, on the three larger programs, TCA\textsuperscript{types} takes on average only 20% more time than TCA\textsuperscript{expand-this}, and TCA\textsuperscript{types-terms} is actually always faster than TCA\textsuperscript{expand-this}. This is explained by the more precise (and therefore smaller) sets $\hat{R}$ and $\hat{X}$ computed by the context-sensitive algorithms. A major source of the speedup of TCA\textsuperscript{types-terms} over TCA\textsuperscript{types} is that the implementation of substituting a type for a type parameter that occurs inside a complicated type is slow. In many cases, term argument type propagation can copy the entire (already substituted type) faster than it would take to replace the type parameters within it.

5.2 Application to Specialization

The evaluation so far has focused on the output of the TCA\textsuperscript{types-terms} analysis. In this section, we show how the analysis improves the effectiveness of specialization.

Generic classes and methods can be compiled to low-level code using two approaches. A heterogeneous approach duplicates the generic code and adapts it for every set of type arguments (Kennedy and Syme 2001; Morrison et al. 1991). This produces many low-level versions of a generic class or method, each adapted to efficiently handle a single type of data. A homogeneous approach generates a single copy with the type parameters erased to their upper bound, commonly Object, that can accommodate values of any type (Bracha et al. 1998).

Similar approaches have also been developed for functional languages with polymorphic types. Intentional type analysis (Harper and Morrisett 1995) introduces user-facing syntax that is similar to runtime reflection that can be used to inspect types and generate specialized classes at run time. For functional programs requiring boxing, (Henglein and Jørgensen 1994) introduces a rewriting algorithm that places the boxing and unboxing operations to minimize the number of coercions executed according to a formal optimality criterion.

Both approaches have benefits and drawbacks: Although the homogeneous approach minimizes the amount of generated low-level code, it has poor performance: Each time a value of primitive type flows into and out of generic code, it must be boxed into a freshly-allocated object and respectively unboxed back to its primitive type (Leroy 1992). The heterogeneous approach avoids boxing and unboxing, but it requires knowing the set of possible type arguments. Furthermore, the number of combinations of type arguments used to instantiate a generic class or method grows exponentially with the number of type parameters, making the heterogeneous approach impractical. Both Java and Scala use the homogeneous translation by default, despite its negative effect on performance.

Specialization is a technique that allows compiling selected classes and methods using the heterogeneous approach (Dragos and Odersky 2009; Dragos 2010; Goetz 2014), while leaving the rest of the generic code to use the default homogeneous translation. In Scala, specialization allows the programmer to annotate the type parameter of a class or method as @specialized. Based on this annotation, the compiler generates 10 versions of the code, one for the universal Object type and one for each of the 9 primitive Scala types. When the class or method has $n$ type parameters annotated as @specialized, the compiler generates $10^n$ versions of the code. The compiler also allows a more fine-grained annotation to specialize a type parameter only for a specified subset of the primitive types. For example, the annotation @specialized(Int) would cause two versions of the code to be generated, one for primitive integers and the other for the universal Object type (in which all other primitive types can be encoded using boxing). To make use of these newly created code variants, the compiler rewrites each generic class instantiation and each generic method call to refer to the appropriate specialized version indicated by the type arguments.

Specialization produces significant speedups, sometimes in excess of 10x, because boxing and unboxing operations often end up in hot loops (Dragos and Odersky 2009; Dragos 2010). However, the increase in code size quickly becomes impractical. For example, specializing a map data structure, which has two type parameters, generates 100 variants, which makes distribution infeasible. A function type with two arguments and one return value requires three type parameters, and therefore an unreasonable 1000 variants.

Miniboxing (Ureche et al. 2013) is an alternative heterogeneous approach that encodes multiple primitive values in a single (larger) slot, thus reducing the number of variants from $10^n$ to $3^n$. Using miniboxing, the map data structure, Map[Key, Value], requires only 9 variants, while the two-argument function, Function2[T1, T2, R] requires 27 variants. As we will see later, the TCA\textsuperscript{types} analysis can further reduce the number of variants generated by miniboxing.

The fundamental problem remains: both specialization and miniboxing trigger excessive bytecode growth, making them infeasible to use as the default compilation scheme for generics. To avoid the excessive bytecode growth, programmers must carefully choose which type parameters are to be specialized. Furthermore, they must decide the exact prim-
### Table 2. Benchmark running time, for 3 million elements. The time is reported in milliseconds. Lower is better.

<table>
<thead>
<tr>
<th></th>
<th>ArrayBuffer.append</th>
<th>ArrayBuffer.reverse</th>
<th>ArrayBuffer.contains</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>speedup</td>
<td>time</td>
</tr>
<tr>
<td>Erasure</td>
<td>37.3 ± 0.1</td>
<td>1x</td>
<td>12.5 ± 0.1</td>
</tr>
<tr>
<td>Specialization - Naive</td>
<td>13.0 ± 0.1</td>
<td>2.9x</td>
<td>1.7 ± 0.1</td>
</tr>
<tr>
<td>Specialization - Call Graph</td>
<td>13.0 ± 0.1</td>
<td>2.9x</td>
<td>1.7 ± 0.1</td>
</tr>
<tr>
<td>Miniboxing - Naive</td>
<td>19.9 ± 0.1</td>
<td>1.9x</td>
<td>1.7 ± 0.1</td>
</tr>
<tr>
<td>Miniboxing - Call Graph</td>
<td>19.9 ± 0.1</td>
<td>1.9x</td>
<td>1.7 ± 0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LinkedList creation</th>
<th>LinkedList.hashCode</th>
<th>LinkedList.contains</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>speedup</td>
<td>time</td>
</tr>
<tr>
<td>Erasure</td>
<td>171.2 ± 40.3</td>
<td>1x</td>
<td>17.0 ± 0.1</td>
</tr>
<tr>
<td>Specialization - Naive</td>
<td>34.1 ± 0.7</td>
<td>5.0x</td>
<td>16.9 ± 0.1</td>
</tr>
<tr>
<td>Specialization - Call Graph</td>
<td>33.7 ± 0.8</td>
<td>5.1x</td>
<td>16.9 ± 0.1</td>
</tr>
<tr>
<td>Miniboxing - Naive</td>
<td>31.0 ± 0.6</td>
<td>5.5x</td>
<td>16.2 ± 0.1</td>
</tr>
<tr>
<td>Miniboxing - Call Graph</td>
<td>31.1 ± 0.6</td>
<td>5.5x</td>
<td>16.2 ± 0.1</td>
</tr>
</tbody>
</table>

**Figure 4.** Graphical representation of the data in Table 2, in milliseconds. Lower is better.

In this context, we introduce the TCA_{types} analysis to determine the specific contexts that we have introduced in the TCA_{types} analysis.

When the TCA_{types} analysis is employed, the specialization annotations generated contain the exact primitives used in the code and nothing more, reducing the bytecode generated as much as possible while avoiding the boxing operations completely. In the case of miniboxing, the TCA_{types} analysis can indicate if any of the miniboxing encodings is redundant, again saving the creation of redundant heterogeneous variants.

Specialization guided by the TCA_{types} analysis results is fully correct in an open-world context. The specialization transformation does not depend on any soundness assumptions about the specialization annotations, which are normally provided by the programmer. If a type parameter is instantiated by a type argument that was not included in the annotation, the generated code falls back to the default universal object-based implementation and its associated boxing and unboxing. Therefore, unanalyzed code that passes type arguments that the analysis is not aware of will still work.

The TCA_{types} analysis solves this problem by inferring the specialization annotations automatically. In particular, the necessary information is, for each generic class or method, the set of type argument instantiations of its type parameters. This set is exactly the set of contexts explored by the TCA_{types} analysis. Note that the information is not generally obtainable from just a (context-insensitive) call graph. The automatic inference of the specialization annotations depends on the specific contexts that we have introduced in the TCA_{types} analysis.
correctly, though it will understandably not enjoy the same performance improvement as the analyzed code.

To test the effectiveness of our analyses, we have applied them to specialization and miniboxing, reproducing the performance experiments from the miniboxing paper (Ureche et al. 2013). The benchmarks are adapted from two collection classes in the Scala standard library, ArrayBuffer and (linked) List, and selected to cover code patterns commonly used throughout the collection library. They cover a wide range of scenarios: both contiguous and sparse memory storage, custom equality checks, hash code computations, and tight loops that can be further optimized by the JIT compiler (e.g., ArrayBuffer.reverse). Each benchmark method is exercised by a driver program that executes it on collections of 3 million integers. The setup is similar to the one used in the miniboxing paper.

To evaluate the automated inference of specialization annotations, we used the following experimental setup. We first compiled the benchmark programs with the dotty compiler and the TCA\textsuperscript{types-terms} analysis. In general, the TCA\textsuperscript{types-terms} analysis could be more precise, but on these benchmark programs, both analyses produce the same results. The type contexts found by the analysis were translated into specialization annotations inserted into the code. The annotated code was then compiled with the standard Scala compiler and evaluated for performance. We used the standard Scala compiler for this last step for consistency with the experiments in the miniboxing paper, and because the porting of the specialization transformations from the standard Scala compiler to dotty is still in progress. Once the specialization feature is completely ported to dotty, the overall process can be implemented in a single compilation pass that performs the analysis and applies the specializations.

We ran the benchmarks on a server machine with an 8-core Intel i7-4770 processor with the frequency fixed at 3GHz, running the Oracle Java distribution 1.7.0.79 on the Ubuntu 12.04.5 LTS operating system. We used the JMH benchmarking framework (Shipilev) as a harness, due to its close integration with the OpenJDK execution platform: for each benchmark, JMH started the Java Virtual Machine (JVM) with 3GB of memory, warmed up the benchmark code until it was compiled by the HotSpot Just-in-time (JIT) C2 compiler, and then took 20 measurements. To minimize the noise, the process was repeated 10 times for each benchmark. This ensured that the variability introduced by the JIT compiler, the garbage collector (GC) and other processes running on the server was reduced as much as possible.

The performance results are shown in Table 2 and Figure 4. The “Erasure” results are for an unannotated program compiled using a homogeneous translation. The “Specialization - Naive” results simulate a fully heterogeneous translation by annotating every type parameter with @specialize, and using the implementation of the specialization transformation in the standard Scala compiler to generate clones of the

methods. The “Specialization - Call Graph” results evaluate a program with annotations for specialization inferred by the TCA\textsuperscript{types-terms} analysis, and specialized by the standard implementation in the Scala compiler. The same types of naive vs call-graph-based annotations are shown for the “Miniboxing” transformation.

Although the last four compilation strategies achieve similar speedups over the baseline “Erasure” configuration, there is a stark difference in the size of the generated bytecode. The total bytecode size for the two data structures is shown in Table 3. Figure 5 shows the same data graphically. The fully heterogeneous translation (“Specialization - Naive”) requires a prohibitive 11.8x increase in the size of the code compared to the standard homogeneous translation. Miniboxing (“Miniboxing - Naive”) reduces this overhead to a still substantial 4.5x. Using the TCA\textsuperscript{types-terms} analysis to drive the two heterogeneous transformations produces the same performance while further reducing the bytecode size by 5.2x for specialization and 1.7x for miniboxing (the “Specialization - Call Graph” and “Miniboxing - Call Graph” entries, compared to their “Naive” counterparts).

In fact, the code size increase can easily be reduced even further by a tighter integration of the analysis and the specialization transformation. In the current implementation of specialization, if two or more type parameters are annotated, the compiler generates specialized versions of the code for the cross product of the possible argument types. For example, if the keys and values of a map can each be of type Int or Long, the compiler generates all four combinations. However, the analysis could have more precise information that indicates, for example, that only Map[Int, Int] and Map[Long, Long] are ever instantiated. Using this information, the specialization transformation would generate only two versions instead.
of four. However, the current annotation mechanism is not expressive enough to encode this precise information that the analysis provides.

6. Related Work

Context sensitivity has been studied extensively in call graphs for dynamically typed functional languages (Shivers 1988). However, because of Scala’s expressive static type system, call graph construction algorithms for statically-typed languages are more closely related. In object-oriented languages, call graph construction and points-to analysis are interdependent, because virtual calls are resolved using the runtime type of the receiver object pointed to by the call site.

For Java, the most thoroughly studied forms of context are call strings (Shivers 1988) and object sensitivity (Milanova et al. 2002, 2005). Analyses using these forms of context sensitivity have a high cost, and much work has been done to balance analysis cost against the precision of the analysis results (Sridharan and Bodik 2006; Xu and Rountev 2008; Xu et al. 2009; Yan et al. 2011; Bravenboer and Smaragdakis 2009; Smaragdakis et al. 2011; Kastrinis and Smaragdakis 2013; Smaragdakis et al. 2014). In Java, context sensitivity has been found to improve precision of pointer information. On call graph precision, its effect is more modest (Lhoták and Hendren 2006; Lhoták and Hendren 2008; Smaragdakis et al. 2011, 2014), unless very sophisticated context abstractions are used (Feng et al. 2015). In Scala, where use of generic type parameters and abstract type members is pervasive, our static-type-based context-sensitive analysis that can precisely model these features significantly improves call graph precision.

The technique of using type arguments as context is most closely related to the C# type analysis of (Sallenave and Ducournau 2012). Their analysis adds type arguments as context to types of instantiated objects (their analogue of the set $\hat{\Sigma}$). In contrast, our analysis adds context to reachable methods (the set $\mathcal{R}$). The goal of their analysis is to specialize the memory layout of objects, in contrast to our goal of specializing method implementations. As we discussed in Section 4.2, the transformation that propagates type parameters from outer classes and methods into inner methods already gives our analysis the precision that would be gained from adding context to instantiated object types.

The technique of using term argument types as context is most closely related to the Cartesian Product Algorithm (Agiesen 1995) and object sensitivity (Milanova et al. 2002, 2005). Both of these techniques analyze a method in contexts determined by the runtime types of their parameters (CPA) or of only their receiver (object sensitivity). The key difference compared to our technique is that these contexts are estimates of the dynamic type tags of the objects that may flow to the parameters, while our contexts are the statically declared types of the arguments at the call site of the method. This difference is important for scalability. In the existing approaches, the number of contexts grows with the number of types instantiated anywhere in the program that flow to the parameters (raised to the power of the number of parameters in the case of CPA). In our approach, the number of contexts of a method is bounded by the number of its call sites (although those call sites may themselves be replicated in different contexts of the caller).

As we indicated in Section 3, our analysis is defined as an extension of the context-insensitive Scala call graph construction analysis of (Ali et al. 2014, 2015). Our implementation analyzes only the Scala source code presented to the Dotty compiler, not any of the Java bytecode that forms the rest of the complete program. We use the Separate Compilation Assumption to construct a sound partial call graph for the part of the program that is available for analysis (Ali and Lhoták 2012; Ali and Lhoták 2013).

7. Conclusion

We presented several extensions to the $\text{TCA}_{\text{expand-this}}$ algorithm of (Ali et al. 2014, 2015) that both improve call graph precision and decrease analysis time for non-trivial Scala programs. Our algorithms consider type arguments and term argument types, and use them to select more precise targets for virtual dispatch.

We implemented the algorithms in the context of the Dotty compiler and compared their precision and running time on a collection of Scala programs. We have found that $\text{TCA}_{\text{types}}$ is significantly more precise than $\text{TCA}_{\text{expand-this}}$, indicating that tracking type parameters would allow to greatly improve the precision for common Scala code. Furthermore, we showed that $\text{TCA}_{\text{types-terms}}$ is slightly more precise than $\text{TCA}_{\text{types}}$, but is substantially faster, indicating that tracking the static types of the arguments at each call site is beneficial. In particular, the call graphs generated by the context-insensitive $\text{TCA}_{\text{expand-this}}$ algorithm are too imprecise to be usable for method specialization and inlining. The call graphs from both the $\text{TCA}_{\text{types}}$ and $\text{TCA}_{\text{types-terms}}$ algorithms are very precise for this client optimization: they would require specializing the average method only 1.5 times in the worst case, and often much less.

Our work suggests that expressive type systems can not only protect users from writing incorrect code, but could also be used to gather more knowledge about the program to enable more performance optimizations.

While our work was primarily focused on Scala, the ideas of our work are applicable to other statically typed languages with generic types. In particular, type and term propagation could be used to improve call graph construction algorithms for languages such as Java and C#.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada.


References


M. Odersky. The Scala language specification v 2.9, 2014.


