Call Graphs for Languages with Parametric Polymorphism

Dmitry Petrashko
EPFL
dmitry.petrashko@gmail.com

Vlad Ureche
EPFL
vlad.ureche@gmail.com

Ondřej Lhoták
University of Waterloo
olhotak@uwaterloo.ca

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.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords keyword1, keyword2

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 analyzes 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 expert annotations provided 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 $\mathsf{TCA}^{\textit{expand-this}}$ analysis of (Ali et al. 2014). 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

```
1  implicit def Iterable[T](implicit ord: Ordering[T
     ]): Ordering[Iterable[T]] =
2   new Ordering[Iterable[T]] {
3     def compare(x: Iterable[T], y: Iterable[T]):
        Int = {
4       val xe = x.iterator
5       val ye = y.iterator
6
7       while (xe.hasNext && ye.hasNext) {
8         val res = ord.compare(xe.next(), ye.next())
9         if (res != 0) return res
10      }
11
12      Boolean.compare(xe.hasNext, ye.hasNext)
13    }
14  }
```

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 infeasible to inline.

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

$$\mathrm{TCA}^{\textit{expand-this}}_{\mathrm{MAIN}}\ \frac{}{\textit{main} \in R} \qquad \mathrm{TCA}^{\textit{expand-this}}_{\mathrm{NEW}}\ \frac{\begin{array}{c}\text{“new } C()\text{” occurs in } M\\ M \in R\end{array}}{C \in \hat{\Sigma}}$$

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

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

Figure 1. Inference rules of TCA$^{\text{expand-this}}$ from (Ali et al. 2014)

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.

```
15 def lexicographicSort[T](a: Iterable[T]*)(
       implicit o: Ordering[T]) = a.sorted
16
17 lexicographicSort("world", "Hello")
```

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

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. The program defines a generic method `lexicographicSort` that creates a sorted list of values of type `Iterable[T]` by calling the `sorted` method of `SeqLike`. 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 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 `char`.

```
18 def lexicographicSort[T](a: Seq[Iterable[T]])(
       implicit o: Ordering[T]) = a.sorted
19
20 lexicographicSort[Char](
21   Predef.wrapRefArray[WrappedString](
22     new Array(
23       Predef.wrapString("world"),
24       Predef.wrapString("Hello")
25     )
26   )
27 )(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 `static` 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$^{\textit{expand-this}}$ algorithm of (Ali et al. 2014). To enable compar-

$$\textsc{TCA}^{\textit{types}}_{\textsc{main}}\ \frac{}{(\mathit{main},\emptyset)\in R} \qquad\qquad \textsc{TCA}^{\textit{types}}_{\textsc{new}}\ \frac{\begin{array}{c}\text{“new } C()\text{” occurs in } M\\ (M,\ldots)\in R\end{array}}{C\in\hat{\Sigma}}$$

$$\textsc{TCA}^{\textit{types}}_{\textsc{call}}\ \frac{\begin{array}{c}\text{call } e.m\ [\sigma'] \ (\ldots) \text{ occurs in method } M\\ C\in \mathit{SubTypes}(\ \mathit{StaticType}(\mathrm{e})\sigma\ )\\ \text{method } M' \text{ has name } m\\ \text{method } M' \text{ is a member of type } C\\ (M,\sigma)\ \in R \qquad C\in\hat{\Sigma}\end{array}}{(M',\sigma'\sigma|_{\mathrm{dom}(\sigma')})\ \in R}$$

$$\textsc{TCA}^{\textit{types}}_{\textsc{abstract-call}}\ \frac{\begin{array}{c}\text{call } e.m\ [\sigma'] \ (\ldots) \text{ occurs in method } M\\ \mathit{StaticType}(\mathrm{e})\sigma\ \text{ is an abstract type } T\\ C\in \mathit{SubTypes}(\mathit{expand}(\mathrm{T}))\\ \text{method } M' \text{ has name } m\\ \text{method } M' \text{ is a member of type } C\\ (M,\sigma)\ \in R \qquad C\in\hat{\Sigma}\end{array}}{(M,\sigma'\sigma|_{\mathrm{dom}(\sigma')})\ \in R}$$

$$\textsc{TCA}^{\textit{types}}_{\textsc{this-call}}\ \frac{\begin{array}{c}\text{call } D.\mathit{this}.m\ [\sigma'] \ (\ldots) \text{ occurs in method } M\\ D \text{ is the declaring trait of } M\\ C\in \mathit{SubTypes}(D)\\ \text{method } M' \text{ has name } m\\ \text{method } M' \text{ is a member of type } C\\ \text{method } M \text{ is a member of type } C\\ (M,\sigma)\ \in R \qquad C\in\hat{\Sigma}\end{array}}{(M',\sigma'\sigma|_{\mathrm{dom}(\sigma')})\ \in R}$$

$$\textsc{TCA}^{\textit{types}}_{\textsc{local-call}}\ \frac{\begin{array}{c}\text{call } M'\ [\sigma'] \ (\ldots) \text{ occurs in method } M\\ M' \text{ is a local method inside some method } M''\\ (M,\sigma)\ \in R\end{array}}{(M',\sigma'\sigma|_{\mathrm{dom}(\sigma')})\ \in R}$$

Figure 2. Propagation of type arguments

ison 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 $\hat{\Sigma}$ keeps track of the types of objects that may be allocated in these reachable methods. The rule $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{main}}$ initializes R with the main entry point. The rule $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{new}}$ finds object instantiations in reachable methods and adds the types to $\hat{\Sigma}$. The rule $\mathsf{TCA}^{\textit{types}}_{\mathsf{call}}$ resolves a call site $e.m(\ldots)$ using the static type of the receiver e to determine all possible target methods M'. The rule $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{abstract\text{-}call}}$ 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 $\mathsf{TCA}^{\textit{expand-this}}$ algorithm uses the function *expand*() to determine the possible concrete types with which T could be instantiated. The *expand*() 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). The rule $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{this\text{-}call}}$ is a variation of $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{call}}$ 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 $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{local\text{-}call}}$ 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), 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 $\mathsf{TCA}^{\textit{types}}$: 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 $\mathsf{TCA}^{\textit{types}}_{\mathsf{main}}$ pairs the main method with the empty substitution $\emptyset$, since the entry point of the program has no type parameters.

The rule $\mathsf{TCA}^{\textit{types}}_{\mathsf{new}}$ iterates over all reachable method-substitution pairs, ignores the substitution, and adds the types instantiated in each reachable method to $\hat{\Sigma}$, as in the original algorithm.

In the rule $\mathsf{TCA}^{\textit{types}}_{\mathsf{call}}$, for each reachable pair (M,σ), where M is a method and σ is a substitution, σ 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 site, we define the substitution σ' that replaces each type parameter of M' with the argument that is passed for it. In the conclusion of the $\mathsf{TCA}^{\textit{types}}_{\mathsf{call}}$ rule, the caller's context substitution σ is composed with the call site substitution σ'. As a result, if σ' uses one of the type parameters of the caller, it will be replaced using σ with the concrete type that it is instantiated with 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 $\mathrm{dom}(\sigma')$. We use the notation $\sigma'\sigma|_{\mathrm{dom}(\sigma')}$ to denote this restriction.

We apply the analogous modifications to the rules $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{this\text{-}call}}$ and $\mathsf{TCA}^{\textit{expand-this}}_{\mathsf{abstract\text{-}call}}$ to obtain the new rules $\mathsf{TCA}^{\textit{types}}_{\mathsf{this\text{-}call}}$ and $\mathsf{TCA}^{\textit{types}}_{\mathsf{abstract\text{-}call}}$.

Because the set of possible types is unbounded, the set of reachable methods paired with type substitutions could grow without bound. This is demonstrated by the following example:

```
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 $[\mathtt{A} \mapsto \mathtt{Int}]$, calls `foo` in context $[\mathtt{A} \mapsto \mathtt{(Int, Int)}]$, which calls `foo` in context $[\mathtt{A} \mapsto \mathtt{((Int, Int), (Int, Int))}]$, 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 $(M, [\mathtt{N}_i \mapsto \mathtt{T}_i])$, we loosen the precision of the last created context for the same method $(M, [\mathtt{N}_i \mapsto \mathtt{T}'_i])$ 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: $(M, [\mathtt{N}_i \mapsto \mathit{lub}(\mathtt{T}_i, \mathtt{T}'_i)])$. 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. This can be achieved by transforming the code before doing the analysis, 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. We illustrate the transformation with the following example program, in which method `bar` is nested in method `foo`, which is nested in class `C`:

```
class C[T] {
 def foo[U](t: T, u: U) = {
  def bar[V](t: T, u: U, v: V) = {...}

  bar[Double](t, u, 1.0)
 }
}
(new C[Int]).foo[String](5, "")
```

The above program would be transformed as follows:

```
class C[T] {
 def foo[T2, U](t: T2, u: U) = {
  def bar[T3, U2, V](t: T3, u: U2, v: V) = {...}

  bar[T2,U,Double](t, u, 1.0)
 }
}
(new C[Int]).foo[Int,String](5, "")
```

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 `Int` in the type `C[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:

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

Buffer123.length()
```

The program gets transformed to:

```
abstract class Buffer {
  type U
  type T <: Seq[U]
  def elements[U2, T2<: Seq[U2]]: T2
  def length[U2, T2<: Seq[U2]] = elements[U2, T2
    ].length
}
class Buffer123 {
  type U = Int
  type T = List[Int]
  def elements[U2 = Int, T2 = List[U2]]: T2 =
    List(1,2,3)
}

Buffer123.length[Buffer123.U, Buffer123.T]()
```

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 σ 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 σ of a method M that contains a call site $e.m[\sigma'](\,)$. The only type variables used in the argument substitution σ' are the direct type parameters of M. The context substitution σ provides fully instantiated types for all of these type parameters. Therefore, when σ' and σ 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, *StaticType*$(e)\sigma$, is never abstract after the caller-context substitution σ has been applied to it. The rule $\mathsf{TCA}^{\mathit{types}}_{\mathsf{abstract\text{-}call}}$ is thus never needed

$$\text{TCA}^{\textit{types-terms}}_{\text{MAIN}}\ \frac{}{(\textit{main}, \emptyset, \colorbox{lightgray}{\textit{Array[String]}}) \in R}$$

$$\text{TCA}^{\textit{types-terms}}_{\text{NEW}}\ \frac{\begin{array}{c}\text{“new } C()\text{” occurs in } M\\ (M, \ldots, \colorbox{lightgray}{$\ldots$}) \in R\end{array}}{C \in \hat{\Sigma}}$$

$$\text{TCA}^{\textit{types-terms}}_{\text{CALL}}\ \frac{\begin{array}{c}\text{call } e.m[\sigma'](\colorbox{lightgray}{$\overline{\textit{args}}$}) \text{ occurs in method } M\\ C \in \textit{SubTypes}(\textit{StaticType}(\colorbox{lightgray}{π}, e)\sigma)\\ \text{method } M' \text{ has name } m\\ \text{method } M' \text{ is a member of type } C\\ (M, \sigma, \colorbox{lightgray}{π}) \in R \qquad C \in \hat{\Sigma}\\ \colorbox{lightgray}{$\pi' = (e :: \overline{\textit{args}}).\text{map}(\textit{arg} \Rightarrow \textit{StaticType}(\pi, \textit{arg})\sigma)$}\end{array}}{(M', \sigma'\sigma|_{\text{dom}(\sigma')}, \colorbox{lightgray}{π'}) \in R}$$

$$\text{TCA}^{\textit{types-terms}}_{\text{LOCAL-CALL}}\ \frac{\begin{array}{c}\text{call } M'[\sigma'](\colorbox{lightgray}{$\overline{\textit{args}}$}) \text{ occurs in method } M\\ M' \text{ is a local method inside some method } M''\\ (M, \sigma, \colorbox{lightgray}{π}) \in R\\ \colorbox{lightgray}{$\pi' = \overline{\textit{args}}.\text{map}(\textit{arg} \Rightarrow \textit{StaticType}(\pi, \textit{arg})\sigma)$}\end{array}}{(M', \sigma'\sigma|_{\text{dom}(\sigma')}, \colorbox{lightgray}{π'}) \in R}$$

$$\text{TCA}^{\textit{types-terms}}_{\text{THIS-CALL}}\ \frac{\begin{array}{c}\text{call } D.\textit{this}.m[\sigma'](\colorbox{lightgray}{$\overline{\textit{args}}$}) \text{ occurs in method } M\\ D \text{ is the declaring trait of } M\\ C \in \textit{SubTypes}(D)\\ \text{method } M' \text{ has name } m\\ \text{method } M' \text{ is a member of type } C\\ \text{method } M \text{ is a member of type } C\\ (M, \sigma, \colorbox{lightgray}{π}) \in R \qquad C \in \hat{\Sigma}\\ \colorbox{lightgray}{$\pi' = (D.\textit{this} :: \overline{\textit{args}}).\text{map}(\textit{arg} \Rightarrow \textit{StaticType}(\pi, \textit{arg})\sigma)$}\end{array}}{(M', \sigma'\sigma|_{\text{dom}(\sigma')}, \colorbox{lightgray}{π'}) \in R}$$

Figure 3. Propagation of term argument types

and can be removed from the algorithm, together with the rules for computing the *expand*() sets for abstract types.

We have described the handling of outer parameters here as a program transformation for clarity of presentation. For performance reasons, our implementation does not explicitly transform the code as described in this section. Instead, the analysis processes the original code directly, taking into consideration the modifications that would be made by the transformation as it reads the code. In particular, in the implementation, the set $\hat{\Sigma}$ contains pairs (σ, C), where σ is the type substitution that was used in the context where C was instantiated.

4.3 TCA$^{\textit{types-terms}}$: 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:

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

internalHashCode[Int](42, "null")
```

The receivers `el` and `nullRep` of the calls to `hashCode` are both parameters of `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 `el` is the type parameter `T`, which the context substitution instantiates to `Int`, so we know that the target of `el.hashCode` is the implementation of `hashCode` in `Int`. However, in the call `nullRep.hashCode`, we need to assume that the runtime type of the receiver `nullRep` could be any subtype of `Object`. To further improve precision, the analysis can be extended further to propagate the type of the argument from the call site of `internalHashCode`, which is `String`, into the context in which `internalHashCode` is analyzed. As a result, the analysis could then determine that the call `nullRep.hashCode` calls only the `String` implementation of `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 σ as before, and, in addition, a list π of more precise types for the term parameters of M (including the implict `this` receiver parameter).

The inference rules for the extended algorithm are shown in Figure 3. Changes compared to Figure 2 are shaded. The *StaticType* function is extended to take a list π of more precise parameter types. If e is a parameter of M, then *StaticType*(π, e) returns the more precise type of e given by π; otherwise it just returns the same static type of e as in the previous analyses. We also extend *StaticType* to map over a sequence of terms and return a sequence of their types. The last premise of the TCA$^{\textit{types}}_{\text{call}}$ rule uses *StaticType* to get the precise types of the arguments passed at the call site. The substitution σ is applied to these types. These precise types π' are then included in the context that is added to R in the conclusion of the rule.

5. Evaluation

We implemented the TCA$^{\textit{expand-this}}$ analysis of Ali et al. (2014) and our two extensions TCA$^{\textit{types}}$ and TCA$^{\textit{types-terms}}$ on top of the Dotty compiler[1], 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 1246 Scala programs. To the best of our knowledge, our

[1] https://github.com/lampepfl/dotty

Program	Algorithm	# Instantiated classes	# Classes with reachable method	# Reachable methods	# Reachable contexts	# Maximum contexts per method	# Discovered specializations	Code growth factor	% monomorphic call sites	% bimorphic call sites	% megamorphic call sites	Running time, seconds
List creation	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	149	64	207	207	1	3469	16.75	80.2	7.0	12.8	0.76
	$\mathsf{TCA}^{\mathit{types}}$	117	33	90	90	1	90	1.00	93.0	4.7	2.3	1.30
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	117	31	83	101	2	83	1.00	95.4	2.3	2.3	1.32
List & Vector creation	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	152	79	268	268	1	6358	24.73	73.4	4.1	22.4	1.89
	$\mathsf{TCA}^{\mathit{types}}$	130	36	95	114	2	114	1.20	86.0	2.1	11.9	1.58
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	130	34	90	138	4	112	1.24	88.1	4.5	7.5	1.41
List create and sort	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	157	65	209	209	1	3919	18.75	77.6	6.4	16.0	0.77
	$\mathsf{TCA}^{\mathit{types}}$	126	34	92	92	1	92	1.00	87.2	9.6	3.2	1.54
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	126	34	89	147	2	89	1.00	89.4	8.5	2.1	1.58
List & Vector create and sort	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	170	83	357	357	1	7725	21.64	72.4	2.4	25.2	2.30
	$\mathsf{TCA}^{\mathit{types}}$	142	39	115	140	2	140	1.21	86.2	3.9	9.8	1.64
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	142	37	109	147	5	131	1.20	89.2	2.6	8.2	1.47
List create, sort and print	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	171	68	212	212	1	4146	19.56	78.6	4.1	17.4	1.29
	$\mathsf{TCA}^{\mathit{types}}$	131	37	95	95	1	95	1.00	87.8	9.2	3.1	5.43
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	131	35	92	206	6	92	1.00	89.8	8.2	2.0	3.25
lexicographicSort	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	182	88	293	293	1	5529	18.87	72.7	2.8	24.5	1.50
	$\mathsf{TCA}^{\mathit{types}}$	134	41	102	104	2	104	1.01	86.6	7.7	5.6	5.91
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	134	41	98	231	3	102	1.04	89.1	6.5	4.4	4.08
Page rank	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	229	92	341	341	1	12490	36.63	59.4	8.5	32.1	10.28
	$\mathsf{TCA}^{\mathit{types}}$	145	50	127	173	3	173	1.36	77.4	7.6	15.1	11.22
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	145	45	118	293	5	165	1.40	85.9	9.9	4.3	6.24
Round robin	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	189	76	252	252	1	6272	24.89	72.6	8.1	19.3	9.69
	$\mathsf{TCA}^{\mathit{types}}$	147	46	130	174	1	174	1.34	87.9	8.1	4.0	8.79
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	147	44	123	310	3	165	1.34	87.9	8.9	3.2	3.91
Dotty type-checker	$\mathsf{TCA}^{\mathit{expand\text{-}this}}$	1028	822	10694	10694	1	45278	4.23	55.6	1.8	42.6	893.52
	$\mathsf{TCA}^{\mathit{types}}$	832	695	9347	14011	4	14011	1.50	82.3	0.6	17.1	1071.71
	$\mathsf{TCA}^{\mathit{types\text{-}terms}}$	832	629	8992	37992	43	13122	1.46	90.7	2.6	6.7	637.10

Table 1. Results of the $\mathsf{TCA}^{\mathit{expand\text{-}this}}$, $\mathsf{TCA}^{\mathit{types}}$, and $\mathsf{TCA}^{\mathit{types\text{-}terms}}$ 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, $\mathsf{TCA}^{\mathit{types\text{-}terms}}$. 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.

analyses soundly handle the entire Scala language dialect supported by Dotty, including Dotty-specific extensions to Scala such as trait parameters[2] and repeated by name parameters[3].

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 $\mathsf{TCA}^{\mathit{types\text{-}terms}}$ analysis implemented in Dotty and then show how it can be used for program performance.

[2] http://docs.scala-lang.org/sips/pending/trait-parameters.html

[3] http://docs.scala-lang.org/sips/pending/repeated-byname.html

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 megamorphic: for example, it contains 214 named subclasses of `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 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 M 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 $\mathsf{TCA}^{\textit{expand-this}}$, call graphs constructed by $\mathsf{TCA}^{\textit{types}}$ have 22 % fewer reachable classes and 56% fewer reachable methods on average. The most significant cause of the precision improvement was that $\mathsf{TCA}^{\textit{types}}$ precisely resolved calls on generic super classes where $\mathsf{TCA}^{\textit{expand-this}}$ was imprecise. For example, a call on a `Seq[T]` could dispatch to both `List[Int]` and `Vector[Double]` according to $\mathsf{TCA}^{\textit{expand-this}}$, but $\mathsf{TCA}^{\textit{types}}$ would analyze the call separately within the context of the two different type arguments.

On the Dotty typechecker, the $\mathsf{TCA}^{\textit{types}}$ call graph has 15 % fewer reachable methods than the $\mathsf{TCA}^{\textit{expand-this}}$ 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, $\mathsf{TCA}^{\textit{types-terms}}$ further reduces the number of reachable methods by 5% compared to $\mathsf{TCA}^{\textit{types}}$.

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

On the Dotty type checker, $\mathsf{TCA}^{\textit{types-terms}}$ reduces the number of megamorphic call sites by 60 % compared to $\mathsf{TCA}^{\textit{types}}$. 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 $\mathsf{TCA}^{\textit{types}}$ than in $\mathsf{TCA}^{\textit{expand-this}}$. $\mathsf{TCA}^{\textit{types-terms}}$ does generate more reachable contexts than $\mathsf{TCA}^{\textit{types}}$, but generally still fewer than $\mathsf{TCA}^{\textit{expand-this}}$.

The Dotty typechecker is a special case in this regard. It has substantial amount 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 $\mathsf{TCA}^{\textit{types-terms}}$ as there are reachable methods in $\mathsf{TCA}^{\textit{expand-this}}$.

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 `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 $\mathsf{TCA}^{\textit{expand-this}}$ 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, $\mathsf{TCA}^{\textit{types}}$ and $\mathsf{TCA}^{\textit{types-terms}}$, 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 $\mathsf{TCA}^{\textit{types}}$ substantially increases the percentage of call sites that are statically monomorphic compared to $\mathsf{TCA}^{\textit{expand-this}}$, by around 10 percentage points on small programs and by around 20 percentage points on large programs. $\mathsf{TCA}^{\textit{types-terms}}$ 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 $\mathsf{TCA}^{\textit{expand-this}}$. This is indeed the case on some of the small programs that exercise the library: $\mathsf{TCA}^{\textit{types}}$ takes up to 4x as long as $\mathsf{TCA}^{\textit{expand-this}}$. This is due to more complex rules that require more work performed to process each call site. However, on the three larger programs, $\mathsf{TCA}^{\textit{types}}$ takes on average only 20% more time than $\mathsf{TCA}^{\textit{expand-this}}$, and $\mathsf{TCA}^{\textit{types-terms}}$ is actually always *faster* than $\mathsf{TCA}^{\textit{expand-this}}$. This is explained by the more precise (and therefore smaller) sets R and $\hat{\Sigma}$ computed by the context-sensitive algorithms. A major source of the speedup of $\mathsf{TCA}^{\textit{types-terms}}$ over $\mathsf{TCA}^{\textit{types}}$ 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 direct results of the $\mathsf{TCA}^{\textit{types-terms}}$ 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 in two general ways. A heterogeneous translation generates separate copies of the generic code for every set of type arguments that its type parameters can be instantiated with (Kennedy and Syme 2001; Leroy 1992; Morrison et al. 1991). A homogeneous translation generates a single copy in which each type parameter is erased to a top type such as `Object` that can accommodate values of any type (Bracha et al. 1998). The homogeneous approach has poor performance. When values of primitive types flow into and out of generic code, they must be boxed into freshly-allocated objects and unboxed back to primitive types. On the other hand, the heterogeneous translation depends on knowing the set of possible type arguments. Furthermore, the number of combinations of type arguments with which a class or method can be instantiated grows exponentially with the number of type parameters, and can quickly become impractical. For these reasons, the homogeneous translation is used by Java, and it is the default in Scala, despite its negative effect on performance.

Specialization is a technique that enables a heterogeneous translation of only selected classes or methods for only a selected subset of type arguments (Goetz 2014; Dragos and Odersky 2009; Dragos 2010). The Scala compiler allows the programmer to annotate a type parameter of a class or method as `@specialized`. In this case, 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 to 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. However, the increase in code size quickly becomes impractical. Even a map data structure, which has two type parameters, requires 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 implementation of specialization that reduces the number of variants from 10^n to 3^n, where n is the number of type parameters. The miniboxing implementation also provides warnings ("performance advisories") to notify the programmer about the code locations where boxing and unboxing operations are inserted and to suggest annotations that would eliminate them (Ureche et al. 2015). However, the fundamental problems remain.

The problems with the current state of the art in specialization for Scala can be summarized as follows. Because of the excessive code growth, it is infeasible to apply specialization to all generic type parameters. Therefore, the programmer must manually direct the compiler by annotating the parameters to be specialized. Tuning those annotations requires deep knowledge of the entire code base, including dependent libraries. Manual annotations are error prone, and missing an annotation can seriously harm performance. Furthermore, since different applications use a library in different ways, no specific set of annotations of a library is ideal for all applications that use it. For these reasons, programmers often err on the side of adding too many annotations, which causes large increases in code size.

To alleviate these problems, the $\texttt{TCA}^{types}$ analysis can infer 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 $\texttt{TCA}^{types}$ 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 $\texttt{TCA}^{types}$ analysis.

Specialization guided by the $\texttt{TCA}^{types}$ 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 correctly, though it will understandably not enjoy the same performance improvement as the analyzed code.

To test the effectiveness of our analyses applied to specialization, we have reproduced 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. This is the same setup as was 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 $\texttt{TCA}^{types}$ analysis. 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 scalameter benchmarking framework (Prokopec) version 0.7 as a harness: for each benchmark, scalameter 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 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 without any specialization. The "Specialization - Naive" results simulate a fully heterogeneous translation by annotating every type parameter with `@specialize`, and using the implementation of specialization transformation in the standard Scala compiler to generate specialized versions of the methods. The "Miniboxing" results do the same using the miniboxing plugin. Finally, the "Specialization -

	`ArrayBuffer.append`		`ArrayBuffer.reverse`		`ArrayBuffer.contains`	
	Monomorphic	Megamorphic	Monomorphic	Megamorphic	Monomorphic	Megamorphic
Erasure	36.0	34.6	12.1	12.3	2576.1	5844.5
Miniboxing	18.0	18.1	1.8	1.8	429.8	432.4
Specialization - Naive	12.6	12.6	1.8	1.8	409.5	412.3
Specialization - Call Graph	12.6	12.4	1.8	1.8	412.5	412.2
	`List creation`		`List.hashCode`		`List.contains`	
	Monomorphic	Megamorphic	Monomorphic	Megamorphic	Monomorphic	Megamorphic
Erasure	18.4	18.4	20.7	21.0	2716.7	2867.7
Miniboxing	14.7	14.9	19.4	19.7	2189.3	2140.1
Specialization - Naive	14.7	14.6	19.2	19.1	2181.1	2188.3
Specialization - Call Graph	14.6	14.6	19.2	19.7	2192.5	2214.2

Table 2. Benchmark running time, for 3 million elements. The time is reported in milliseconds. Lower is better.

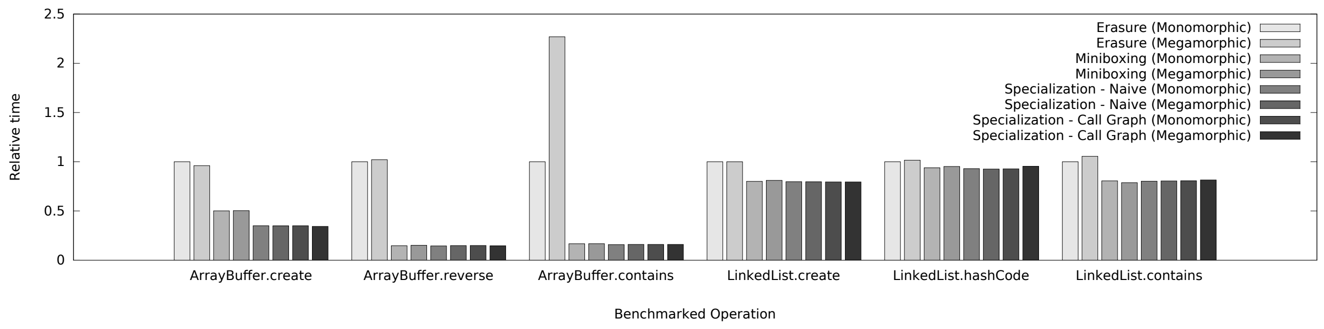

Figure 4. Graphical representation of the data in Table 2, in milliseconds. Lower is better.

Call Graph" results evaluate a program automatically annotated for specialization using the TCA^{types} analysis, and specialized by the standard implementation in the Scala compiler.

The last three compilation strategies achieve the same improvement in performance over the baseline "Erasure" configuration.

However, 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 requires a prohibitive 11.8x increase in the size of the code compared to the standard homogeneous translation. Miniboxing reduces this overhead to a still substantial 4.3x. Automatic specialization using the TCA^{types} analysis achieves the same performance as these two techniques with a code size increase of only 2.3x.

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 every 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 does provide.

Transformation	Bytecode Size (Bytes)
Specialization - Naive	86146
Miniboxing	31372
Specialization - Call Graph	16458
Erasure	7291

Table 3. The bytecode size produced by specializing the ArrayBuffer and LinkedList classes with different approaches. Lower is better.

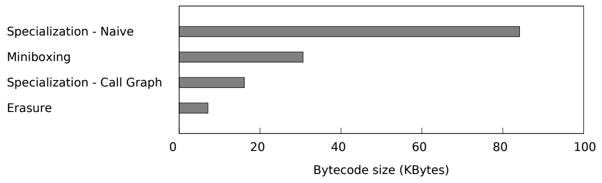

Figure 5. Graphical representation of the data in Table 3, showing the bytecode size in kilobytes. Lower is better.

6. Related Work

We survey two separate areas of related work. First, we discuss the main intended application of our analysis, specialization techniques that have been proposed for Scala and similar languages. Second, we discuss context sensitivity in call graph construction in general, in various programming languages, and compare our analysis to other related analyses.

6.1 Specialization Techniques

In the context of generating efficient Java bytecode from Scala programs, (Dragos 2010) observes that "compilation of polymorphic code through type erasure gives compact code but performance on primitive types is significantly hurt". Consider the following method `foo`:

```
def foo[A](a: A) = a

foo[Int](1)
```

This code is compiled as follows

```
def foo(a: Object) = a

foo(new Integer(1)).asInstanceOf[Integer].value
```

Dragos proposes a specialization technique for Scala that requires the programmer to mark methods to be specialized. The compiler generates specialized versions of each such method for each primitive type. If such a `@specialized` annotation were applied to the `foo` method in our example, the compiler would generate the following code:

```
def foo(a: Object) = a
def foo_i(a: Int)  = a // synthetic clone

foo_i(1)
```

The implementation conservatively generates clones for all 9 of the primitive types in Scala, as well as the reference type (erased to `Object`). For a method with n type parameters, 10^n clones are needed. This limits the use of specialization in Scala. For example, the standard library type `Function2` that represents a function with two parameters has three type parameters (one for the type of each parameter, and a third for the return type). Specializing `Function2` would require $10^3 = 1000$ clones, which is impractical.

Miniboxing (Ureche et al. 2013) is a technique that reduces the number of clones required from 10^n to 2^n. It encodes all primitive types into a single type, a 64-bit `long`, and uses a marker byte to indicate the original type. For each type parameter, only two clones are needed: one for primitive types (encoded as `long`), and one for reference types (encoded as `Object`). This approach makes it viable to mark as `@miniboxed` methods with up to 6 type parameters.

Wider use of miniboxing suggested that similar specialization techniques can harm performance if specialized code is called frequently from generic code and vice versa (Ureche et al. 2015). Consider the following example.

```
def foo[A](a: A) = a
def bar[@miniboxed A](a: A) = while(true) foo(a)
def bar1[A](a: A) = while(true) foo(a)
```

In order to call the generic method `foo`, the specialized method `bar` will need to box `a` in every iteration. In contrast, the value `a` in the generic method `bar1` will already be boxed before `bar1` is called, so it will not have to be boxed again in every iteration of the loop. The miniboxing implementation tries to help users to solve this problem by providing comprehensive warnings that suggest possible changes to the code (Ureche et al. 2015).

Similar techniques are available as part of the .Net runtime (Kennedy and Syme 2001) and are under development for Java as part of Project Valhalla (Goetz 2014).

6.2 Call Graph Construction and Context Sensitivity

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 Bodík 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 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 (Agesen 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). 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 TCA$^{\textit{expand-this}}$ algorithm of (Ali et al. 2014) 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 TCA$^{\textit{types}}$ is significantly more precise than TCA$^{\textit{expand-this}}$, indicating that tracking type parameters would allow to greatly improve the precision for common Scala code. Furthermore, we showed that TCA$^{\textit{types-terms}}$ is slightly more precise than TCA$^{\textit{types}}$, 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 $\mathsf{TCA}^{expand\text{-}this}$ algorithm are too imprecise to be usable for method specialization and inlining. The call graphs from both the TCA^{types} and $\mathsf{TCA}^{types\text{-}terms}$ 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 Java, C#, C, Haskell, Swift, and D.

References

O. Agesen. The Cartesian product algorithm. In *ECOOP '95, Object-Oriented Programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 2–51, 1995.

K. Ali and O. Lhoták. Application-only call graph construction. In J. Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 688–712. Springer, 2012. ISBN 978-3-642-31056-0.

K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In G. Castagna, editor, *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer, 2013. ISBN 978-3-642-39037-1. doi: 10.1007/978-3-642-39038-8. URL http://dx.doi.org/10.1007/978-3-642-39038-8.

K. Ali, M. Rapoport, O. Lhoták, J. Dolby, and F. Tip. Constructing call graphs of scala programs. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 54–79. Springer Berlin Heidelberg, 2014. ISBN 978-3-662-44201-2.

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 183–200, New York, NY, USA, 1998. ACM. ISBN 1-58113-005-8. doi: 10.1145/286936.286957. URL http://doi.acm.org/10.1145/286936.286957.

M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640108. URL http://doi.acm.org/10.1145/1640089.1640108.

J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95, Object-Oriented Programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, 1995.

I. Dragos. *Compiling Scala for Performance*. PhD thesis, IC, Lausanne, 2010.

I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 42–47. ACM, 2009.

Y. Feng, X. Wang, I. Dillig, and C. Lin. EXPLORER : query- and demand-driven exploration of interprocedural control flow properties. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 520–534. ACM, 2015. doi: 10.1145/2814270.2814284. URL http://doi.acm.org/10.1145/2814270.2814284.

B. Goetz. State of the Specialization, 2014. URL http://web.archive.org/web/20140718191952/http://cr.openjdk.java.net/~briangoetz/valhalla/specialization.html.

T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.

G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 423–434. ACM, 2013. ISBN 9781-450-3201-4-6. doi: 10.1145/2462156.2462191. URL http://dl.acm.org/citation.cfm?id=2491956.

A. Kennedy and D. Syme. Design and implementation of generics for the. net common language runtime. In *ACM SigPlan Notices*, volume 36, pages 1–12. ACM, 2001.

X. Leroy. Unboxed Objects and Polymorphic Typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 177–188, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: 10.1145/143165.143205. URL http://doi.acm.org/10.1145/143165.143205.

O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, Apr. 2003. Springer.

O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th International Conference*, volume 3923 of *LNCS*, pages 47–64, Vienna, Mar. 2006. Springer.

O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1391984.1391987.

A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–11. ACM Press, 2002. ISBN 1-58113-562-9. doi: http://doi.acm.org/10.1145/566172.566174.

A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1): 1–41, 2005. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1044834.1044835.

R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An Ad Hoc Approach to the Implementation of Polymorphism. *ACM Trans. Program. Lang. Syst.*, 13(3):342–371, July 1991. ISSN 0164-0925. doi: 10.1145/117009.117017. URL http://doi.acm.org/10.1145/117009.117017.

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

A. Prokopec. ScalaMeter. URL https://web.archive.org/web/20160129115447/https://scalameter.github.io/.

O. Sallenave and R. Ducournau. Lightweight generics in embedded systems through static analysis. In R. Wilhelm, H. Falk, and W. Yi, editors, *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2012, LCTES '12, Beijing, China - June 12 - 13, 2012*, pages 11–20. ACM, 2012. ISBN 9781-450-3121-2-7. doi: 10.1145/2248418.2248421. URL http://dl.acm.org/citation.cfm?id=2248418.

O. Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press, 1988. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54007.

Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 17–30. ACM, 2011. ISBN 9781-450-3049-0-0.

Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: context-sensitivity, across the board. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 50. ACM, 2014. ISBN 9781-450-3278-4-8. doi: 10.1145/2594291.2594320. URL http://dl.acm.org/citation.cfm?id=2594291.

M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 387–400, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1134027.

V. Ureche, C. Talau, and M. Odersky. Miniboxing: improving the speed to code size tradeoff in parametric polymorphism translations. In *ACM SIGPLAN Notices*, volume 48, pages 73–92. ACM, 2013.

V. Ureche, M. Stojanovic, R. Beguet, N. Stucki, and M. Odersky. Improving the interoperation between generics translations. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, pages 113–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3712-0. doi: 10.1145/2807426.2807436. URL http://doi.acm.org/10.1145/2807426.2807436.

G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 225–236, New York, NY, USA, 2008. ACM. ISBN 9781-605-5805-0-0. doi: http://doi.acm.org/10.1145/1390630.1390658.

G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*, pages 98–122. Springer, 2009. ISBN 978-3-642-03012-3.

D. Yan, G. H. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. In M. B. Dwyer and F. Tip, editors, *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 155–165. ACM, 2011. ISBN 9781-450-3056-2-4.