Call-graph-based Optimizations in Scala

Semester Project

Romain Beguet

Mentored by Nicolas Stucki Supervised by Dmitry Petrashko Directed by Martin Odersky

École polytechnique fédérale de Lausanne, Switzerland
{first.last}@epfl.ch

1. Introduction

Scala [3] provides various high-level features that programmers can utilize in order to write readable, modular, and extensible code in a productive manner. Generics, virtual methods, higher-order functions and higher-kinded types [7] are some of these features. As one could expect, most of them come with a cost, generally in the form of an overhead in runtime performance. Take for example the following equivalent codes [10]:

Java

```
public double average(int[] data) {
  int sum = 0;
  for(int i = 0; i < data.length; i++) {
    sum += data[i];
  }
  return sum * 1.0d / data.length
}
```

Scala

```
def average(x: Array[Int]) = {
  x.reduce(_ + _) * 1.0 / x.size
}
```

The Scala version is more concise and descriptive because a lot of the implementation is abstracted out by the `reduce` method. However, it uses some costly mechanisms under-the-hood to achieve it. Observe the implementation of `reduce` and `foreach` below:

```
def reduce(op: Function2[Obj, Obj, Obj]): Obj = {
  var first = true
  var acc: Obj = null
  this.foreach{ e =>
    if (first) {
      acc = e
      first = false
    } else acc = op.apply(acc, e)
  }
  acc
}
def foreach(f: Function1[Obj, Obj]) {
  var i = 0
  val len = length
  while (i < len) {
    f.apply(this(i));
    i += 1
  }
}
```

- Boxing occurs when `this(i)` (an integer) has to be passed to the `apply` method of `Function1[Obj, Obj]`.
- Dynamic dispatch happens at several places: `this.foreach`, `op.apply`, `f.apply`, etc.

As a result, the performance of the Scala version is tremendously impacted: while the Java version runs in 20ms, the Scala version takes as long as 650ms, which is more than 30 times slower.

1.1 Existing Solutions

In order to address these overheads, several techniques have been developed. For example, specialization [4] is a feature offered by Scala compilers which aims at removing the important overhead introduced by the implicit boxing of primitive values in generic code by using heterogeneous compilation instead of the default homogeneous compilation (erasure) [12]. It operates more or less by duplicating the generic code marked specialized for every primitive type that Scala has [8], which is 10 including the reference type (`Unit`, `Boolean`, `Byte`, `Char`, `Short`, `Int`, `Long`, `Float`, `Double` and `AnyRef`). However, there are well-known pitfalls of using this method:

Code size explosion. For a generic code parameterized by n type variables, it needs to generate 10^n variants of it. Not only does it affect the final size of the executable, it also introduces additional drawbacks at runtime: it can pollute the CPU cache, and even prevent the JVM from performing more costly optimizations due to spending more time on optimizing every variant.

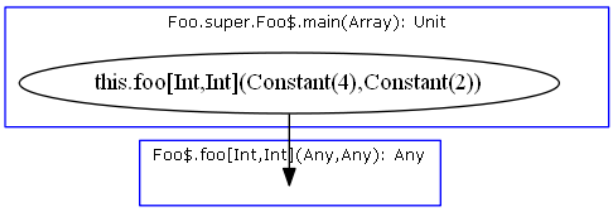
Erasure incompatibility. Specialized code must only be called from a non-generic callsite or another specialized generic callsite to be taken advantage of. Indeed, if a type that is erased to a non-primitive type is used to instantiate some specialized code, it will fallback to the erased version of the code, because the lack of knowledge about the true type argument prevents it from choosing the right variant. Thus, the optimization is completely canceled. This is very problematic because it means that it becomes totally ineffective when used with existing code-bases or libraries that do not use specialization.

Miniboxing [13] also tries to reduce the boxing overhead. It is a middle ground between the heterogeneous and homogeneous approaches, encoding several primitive types into a larger one and thus reducing the duplication factor, without paying the price of boxing. In most benchmarks, miniboxing matches the performance of specialization, while generating significantly less low-level code (3^n instead of 10^n). However, it still suffers from the incompatibility with erasure-based compilation.

1.2 Whole Program optimizations

The techniques presented above both try to solve a problem that is inherent to the fact that the compilation is done in an *open world*. That is, our code could be used as a library for another project, which means it must keep its interface intact and only optimizations local to procedures can be applied. Dotty-linker [1] [9] is a project

Figure 1. Call graph with type propagation



which is designed to be used as the final compilation step towards releasing an executable program. Therefore, it has way more freedom in terms of transformations that it can apply to the code: it can perform any transformation that keep the code functionally equivalent. It also means that it has knowledge of the whole program. In particular, it knows the entry points of the program, thus enabling call-graph-based whole program optimizations [2], which are the focus of this project.

2. Call-graph-based Optimizations

In this section, I will present some optimizations that can be implemented in dotty-linker using the call graph construction algorithm.

2.1 Dead code elimination

Dead code elimination is an optimization phase which aims at removing all the code that is unreachable in our final binaries. Typically, the Scala standard library is very well furnished and it is very rare that a single program uses every aspect of it. As a result, the final standalone binaries of that program will contain a lot of dead code, which could have been eliminated using dead code elimination. In dotty-linker, this phase is pretty straightforward to implement once we have access to the call graph. We basically write a `MiniPhaseTransform` which, upon transforming a method m, first checks that it is reachable using the call graph ($m \in reachableMethods$) and eliminates it if not. Eliminating for now means replacing its body with `throw new DeadCodeEliminated`. Note that for obvious reasons, this optimization is totally incompatible with reflection.

2.2 Auto specialization for types

Auto specialization [10] is an optimization phase which tries to solve the problem stated in the introduction, i.e. remove the overhead introduced by the boxing of primitive values when entering generic code. Remember that specialization (or miniboxing) both generate every possible duplicate of the subject code regardless of which variants are actually used in the program. Consider the following program:

```
1 object Foo {
2   def foo[@specialized A, @specialized B](x: A, y: B):
      A = x
3
4   def main(args: Array[String]): Unit = {
5     foo[Int, Int](4, 2)
6   }
7 }
```

Method `foo` will have 100 variants generated for it, even though the only call to this method in the whole program is `foo[Int, Int]`. Generating the call graph for this program using type arguments propagation [11] yields the call graph from figure 1. Therefore, we can simply generate the specialization of `foo` for `[Int, Int]` in our example.

More generally, assume that reachable methods are stored in the call graph alongside the context they were called with. A call context maps the type parameters $(T_1, ..., T_n)$ of the method called to the type arguments $(X_1, ..., X_n)$ it was called with. Then we can simply generate specializations of a method for each unique instantiation of this method appearing in the call graph. Formally, we can specialize a method m with type parameters $T_1, ..., T_n$ for every type argument tuple $(X_1, ..., X_n)$ in the set:

$$\{(X_1, ..., X_n) : \exists\, sub \mid (m, sub) \in reachableMethods \\ \wedge\, sub = \{T_i \to X_i\}_{1 \le i \le n}\}$$

Therefore, our example could be specialized to:

```
1 object Foo {
2   def foo[A, B](x: A, y: B): A = x // could be dead
      code eliminated
3
4   def foo_Int_Int(x: Int, y: Int): Int = x
5
6   def main(args: Array[String]): Unit = {
7     foo_Int_Int(4, 2)
8   }
9 }
```

Finally, note that *Auto* does not only mean that the needed specializations are automatically chosen, it also means that the programmer does not have to annotate manually what to specialize. Although it may be useful to offer some way to control how auto specialization should work for some particular applications, most of the time letting the compiler specialize everything should yield good results. In fact, this is precisely what a C++ compiler does when compiling code which instantiates templates. For Scala programs, it was shown [11] that the average method needed to be specialized 1.5 times in the worst case, and often much less.

2.3 Auto specialization for terms

Another interesting aspect of auto specialization is that in addition to specializing generic methods for different type arguments, it can also be used to specialize methods for different term arguments: in the actual implementation of the call graph construction algorithm, type parameters and term parameters are treated the same way. Therefore, if we are certain that some argument `x` passed to a method has precisely the type `T` (not just a subtype of it), we can create another variant of this method which handles specially this case. In this specialized variant, any virtual call to `x` can be inlined because we know exactly which implementation would be called at runtime. Consider the following code:

```
1 object Foo {
2   class A {
3     def hi = println("hi from A")
4   }
5   class B extends A {
6     override def hi = println("hi from B")
7   }
8   def bar(x: A) = x.hi
9   def foo(x: A) = bar(x)
10  def main(args: Array[String]): Unit = {
11    foo(new B)
12  }
13 }
```

Here, both methods `foo` and `bar` could benefit from term specialization:

- The call to `foo` in method `main` is an **initiator**. For now, the call graph construction algorithm can only initiate a precise tracing of the term argument if the argument passed is of the form `new T[...](...)` or if it is a literal value. However a

stronger analysis could also consider the call `foo(x)` in the following code as an initiator:

```
def main(args: Array[String]): Unit = {
  val x = new T[...](...)
  ...
  foo(x)
}
```

- The call to `bar` in method `foo` is a **propagator**. If a precise tracing of some term argument has been initiated, it will be propagated (the same way type arguments are propagated as explained in subsection 3.2) as long as the concerned term is directly used. Likewise, a stronger analysis could also consider the call `bar(y)` in the following code as a propagator:

```
def foo(x: A) = {
  val y = x
  ...
  bar(y)
}
```

In the end, the code produced by term specialization could look like:

```
object Foo {
  class A {
    def hi = println("hi from A")
  }
  class B extends A {
    override def hi = hi_B(this)
    static def hi_B(x: B) = println("hi from B")
  }
  def bar(x: A) = x.hi
  def foo(x: A) = bar(x)

  def bar_B(x: B) = B.hi_B(X)
  def foo_B(x: B) = bar_B(x)

  def main(args: Array[String]): Unit = {
    foo_B(new B)
  }
}
```

Here, static methods by definition do not require runtime dispatch. At best, they could even be inlined.

3. Call graph construction

After having shown concrete use cases of a call graph algorithm in a Scala compiler, I will now briefly introduce how the call graph is built in dotty-linker, as it is needed to understand my contributions which I present later on. Let's take an example of program for which we want to build the call graph:

```
class A {
  def foo: A = new B
}
class B extends A {
  override def foo: A = this
}
object Foo {
  def main(args: Array[String]): Unit = {
    var x: A = new A
    var i = 0
    while (i < 2) {
      x = x.foo
      i += 1
    }
  }
}
```

3.1 Collecting Summaries

The first step towards building the call graph is to generate a simpler representation of every method in the code, which is done in dotty-linker by the `CollectSummaries` phase: without diving too much into details, to each method is associated a `MethodSummary` which contains different kind of information about it:

```
case class MethodSummary(methodDef: Symbol,
   thisAccessed: Boolean,
   methodsCalled: Map[Type, List[CallInfo]],
   accessedModules: List[Symbol],
   argumentReturned: Byte,
   argumentStoredToHeap: List[Boolean]
)
```

In particular, `methodsCalled` is collection of `CallInfo`s representing every call that occurs inside the method's body. A `CallInfo` stores data about the call that is relevant to the call graph construction, including the method called, as well as the type and term arguments. In our example, the (simplified) summaries will be:

- `A.foo`: `[new B]`
- `B.foo`: `[]`
- `main`: `[new A, x.foo]`

Algorithm 1 Call graph construction algorithm

edges ← {}
reachableMethods ← {}
reachableTypes ← {}
for all e ∈ entryPoints **do**
  reachableMethods ← reachableMethods ∪ e
end for
while new reachableMethods **or** new reachableTypes **do**
  for all caller ∈ reachableMethods **do**
    for all call c ∈ $\text{summary}_{\text{caller}}$ **do**
      case c **is** "new T"
        reachableTypes ← reachableTypes ∪ T
        edges ← edges ∪ caller → T.<*init*>
        reachableMethods ← reachableMethods ∪ T.<*init*>
      case c **is** "$rec.callee$"
        for all T ∈ reachableTypes, T <:< $type$(rec) **do**
          if ∃ method $o \in T : sig(\text{o}) = sig(\text{callee})$ **then**
            edges ← edges ∪ caller → callee
            reachableMethods ← reachableMethods ∪ callee
          end if
        end for
    end for
  end for
end while

3.2 Building the Call Graph

The algorithm for building the call graph is based on a *worklist* data structure storing methods and types that are found reachable during the construction of the call graph. As such, the call graph is extended as long as the worklist has not reached a fixed point. In other words, the algorithm reiterates as long as the last iteration introduced new reachable methods or types. Algorithm 1 is a pseudo code giving a good intuition of the actual implementation. The reason why a worklist data structure is necessary is because it is possible to discover new potential calls from an already processed callsite later during the call graph construction. Indeed, a callsite can have several out edges: in the case of dynamic dispatch for example [6], where some method m of class C is called, we cannot know at compile time which implementation will be called at runtime if C has several subclasses overriding m. Therefore, we must add an edge from the callsite to each implementation of m from every subtype of C that are found reachable in the program. The

Figure 2. Call graph after one iteration

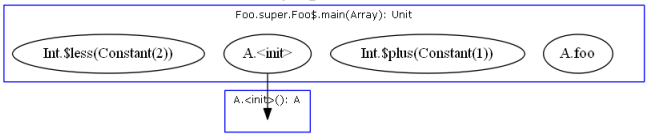
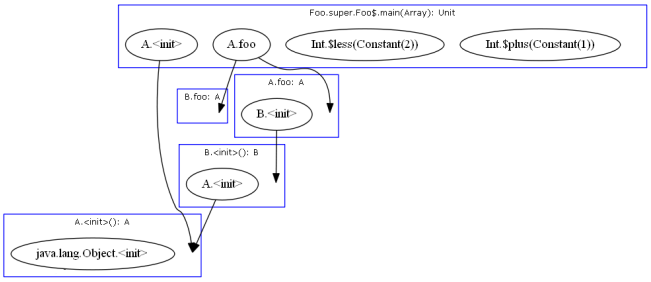


Figure 3. Call graph after two iterations



fact that this information will build up throughout the call graph construction is precisely why it may be necessary to come back to expand an already processed callsite, thus justifying the worklist-based algorithm. Let's take back our example to illustrate this. After running step by step one iteration of our algorithm, the call graph is as in figure 2. The attentive reader may notice that it is missing a call to `B.foo`. This happens because the type `B` is found reachable only after the call `x.foo` is processed. Fortunately, since the reachable types after that first iteration are `[A, B]`, the second iteration of the algorithm will yield the correct call graph which can be seen in figure 3. The algorithm is obviously way more complex in reality, as it has to deal with a lot of different constructs which may introduce new reachable methods or types: `super` calls, calls through *Java* code, calls to inner functions, etc.

Another aspect that we want to handle is the propagation of type arguments. Consider the following example containing generic code:

```
1  object Foo {
2    def foo[A]: Unit = {}
3    def bar[A, B]: Unit = {
4      Foo.foo[A]
5      Foo.foo[B]
6    }
7    def main(args: Array[String]): Unit = {
8      Foo.bar[Int, Double]
9    }
10 }
```

In such scenario, it can be very interesting to evaluate exactly what type arguments each generic method is called with and to output an edge for each different instantiation. Indeed, this information can be then utilized by later optimization phases to great help, particularly for *auto specialization* as explained previously. Since we know the entry points of the program, we can clearly perform this evaluation. The intuition is the following:

- When a generic method o with type parameters $O_1, ..., O_n$ is called with type arguments $X_1, ..., X_n$, we build a substitution map `sub` defined as $\{O_i \to X_i\}_{1 \leq i \leq n}$, which we store alongside o in our set of reachable methods. Additionally, we assume that entry points have an empty context.
- When we process a callsite: we have to put ourselves in its context: any time a generic code is instantiated with type arguments $X_1, ..., X_n$, we first apply to it the substitution map `ctx` of our current context. The result $Y_1, ..., Y_n$ of the application is simply defined as:

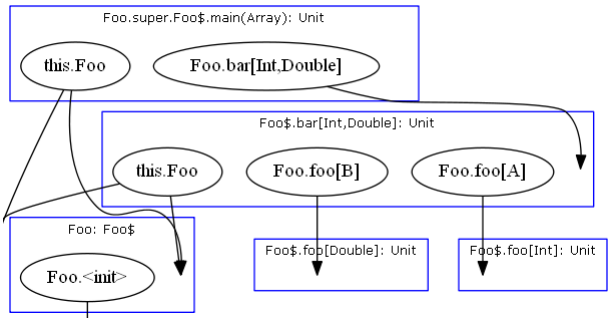
Algorithm 2 Call graph construction algorithm - updated

edges ← {}
reachableMethods ← {}
reachableTypes ← {}
for all e ∈ entryPoints **do**
  reachableMethods ← reachableMethods ∪ (e, {})
end for
while new reachableMethods **or** new reachableTypes **do**
  for all (caller, ctx) ∈ reachableMethods **do**
    for all call c ∈ $\text{summary}_{\text{caller}}$ **do**
      case c **is** "new T"
        reachableTypes ← reachableTypes ∪ T
        edges ← edges ∪ caller → T.<*init*>
        reachableMethods ← reachableMethods ∪ T.<*init*>
      case c **is** "$rec.callee[X_1, ..., X_n]$"

$$Y_i \leftarrow \begin{cases} S & \text{if } \exists S : (X_i \to S) \in \text{ctx} \\ X_i & \text{otherwise} \end{cases} \quad \forall 1 \leq i \leq n$$

        for all T ∈ reachableTypes, T <:< $type$(rec) **do**
          if ∃ method $o \in T : sig(\text{o}) = sig(\text{callee})$ **then**
            let $O_1, ..., O_n$ be the type parameters or o
            sub ← $\{O_i \to Y_i\}_{1 \leq i \leq n}$
            edges ← edges ∪ (caller, ctx) → (callee, sub)
            reachableMethods ← reachableMethods ∪ (callee, sub)
          end if
        end for
      end for
    end for
  end while

Figure 4. Call graph with type propagation



$$Y_i = \begin{cases} S & \text{if } \exists S : (X_i \to S) \in \text{ctx} \\ X_i & \text{otherwise} \end{cases} \quad \forall 1 \leq i \leq n$$

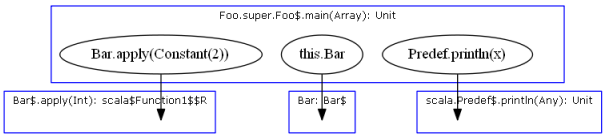
The updated algorithm is given in algorithm 2. For our example, it generates the call graph from figure 4.

In reality there are many other cases which have to be handled but I do not cover here: generic classes, inheriting generic base classes, inner functions/classes depending on outer type variables, etc. The complete set of rules to perform context-sensitive call graph construction and a detailed explanation of the algorithm can be found in paper [11].

4. Contributions

In this section, I will present my contributions to the project. I will only concentrate on the biggest contributions, omitting bug fixes, testing, etc. which, even though took a lot if not most of the time I spent on the project, are not really worth presenting in a report.

Figure 5. Call graph with no pattern matching support



4.1 Adding support for Pattern Matching

Consider the following code:

```
case class Bar(x: Int)
object Foo {
  def main(args: Array[String]): Unit = {
    Bar(2) match {
      case Bar(x) => println(x)
      case _ =>
    }
  }
}
```

Before adding support for pattern matching [5], the call graph looks like in figure 5. When compiling the program with dead code elimination activated, we get:

```
Exception in thread "main"
    dotty.runtime.DeadCodeEliminated
        at Bar$.unapply(Test.scala:112)
        at Foo$.main(Test.scala:116)
        at Foo.main(Test.scala)
```

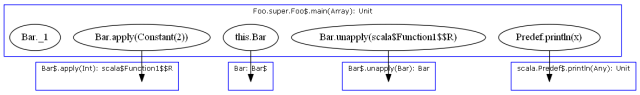
Indeed, since calls to `unapply` are implicitly done upon checking against the different cases during pattern matching, our current call graph implementation did not catch them and they were consequently eliminated. Therefore, the way I chose to address this issue was to explicitly add in our method summary every possible implicit call that can happen during a pattern matching. In order to do this, I first identified exactly all the possible forms that the `unapply` function can take:

1. `unapply` returning a `Boolean`
2. `unapply` returning an `Option[T]`
3. `unapply` returning an `Option[(T1, ..., Tn)]`
4. `unapply` returning a `ProductN[T1, ..., Tn]`
5. `unapplySeq` returning an `Option[Seq[T]]`

Then, when transforming a `Match` tree, simply fetch the top-level cases that are `Unapply` trees and register a call to their corresponding `unapply` function (given through `Unapply.fun`), with the `Match` selector as argument (given by `Match.sel`). However, there may be more to do depending on the return value of the `unapply` call:

1. For the first case, there are no more calls to register since the returned boolean will simply be checked in a branch.
2. In the second case, the returned value is an `Option[T]`. In this case we need to register an additional call to its `empty` method followed by a call to `get`.
3. In the third case, an `Option[(T1, ..., Tn)]` is returned. Thus we must first register the same two calls as for the second case, with additional calls to the product elements' getters (`_1`, ..., `._n`).
4. In the fourth case, like for the third one, we must register calls to the product element's getters. However, the return value is not an option which is checked for emptiness, but a reference which is checked for nullity, so we don't need to generate calls to `empty` or `get`.
5. The fifth case is a bit more tricky. We first generate the calls to `Option`'s `empty` and `get` as in the second case, but we must as well register the calls to access the elements of the resulting sequence. After inspecting the disassembly of a simple test, we find out that: first of all, the length of the resulting sequence is checked against the number of arguments given in the pattern, which means we must register a call to `Seq`'s `lengthCompare` method. Then, we notice that the first element is retrieved with the `head` method. Also, the subsequent ones (if any) are accessed with the `apply` method. We also generate these calls and we are done.

Figure 6. Call graph with pattern matching support



With this implemented, we can safely compile and run our example code with dead code elimination enabled, or any pattern matching constructs which contains `Unapply` trees in the top-level cases. The new call graph looks like in figure 6.

In order to completely support pattern matching however, we must also handle nested patterns. Consider the following code:

```
object Foo {
  object Twice {
    def unapply(x: Int) = if (x % 2 == 0) Some(x / 2)
      else None
  }
  object Thrice {
    def unapply(x: Int) = if (x % 3 == 0) Some(x / 3)
      else None
  }
  def main(args: Array[String]): Unit = {
    6 match {
      case Twice(Thrice(x)) => println(x)
      case _ =>
    }
  }
}
```

If we use the implementation described so far to compile and run our code, we get:

```
Exception in thread "main"
    dotty.runtime.DeadCodeEliminated
        at Foo$Thrice$.unapply(Test.scala:126)
        at Foo$.main(Test.scala:130)
        at Foo.main(Test.scala)
```

The reason is that although we correctly registered the implicit calls for out top-level case `Twice.unapply`, we did not do it for `Thrice.unapply`. To fix this, we have to consider the deconstructed values of the top-level cases as the new selectors for the nested patterns, and therefore recursively register the implicit calls on these new expressions. Looking at our example, we can see this process as transforming the nested pattern into:

```
Twice.unapply(6).get match {
  case Thrice(x) => ...
}
```

and performing the exact same implicit call generation described above on this new `Match` tree. We do this recursively, to finally obtain a complete support for pattern matching in the call graph construction algorithm.

Algorithm 3 Call graph construction algorithm handling closures

edges ← {}
reachableMethods ← {}
reachableTypes ← {}
for all e ∈ entryPoints **do**
  reachableMethods ← reachableMethods ∪ (e, {})
end for
while new reachableMethods **or** new reachableTypes **do**
  for all (caller, ctx) ∈ reachableMethods **do**
    for all call c ∈ $\text{summary}_{\text{caller}}$ **do**
      case c **is** "new T"
        reachableTypes ← reachableTypes ∪ T
        edges ← edges ∪ caller → T.<*init*>
        reachableMethods ← reachableMethods ∪ T.<*init*>
      case c **is** "$rec.callee[X_1, ..., X_n]$"

$$Y_i \leftarrow \begin{cases} S & \text{if } \exists S : (X_i \rightarrow S) \in \text{ctx} \\ X_i & \text{otherwise} \end{cases} \quad \forall 1 \leq i \leq n$$

        for all T ∈ reachableTypes, T <:< $type$(rec) **do**
          if ∃ method $o \in T : sig(\text{o}) = sig(\text{callee})$ **then**
            let $O_1, ..., O_n$ be the type parameters or o
            sub ← $\{O_i \rightarrow Y_i\}_{1 \leq i \leq n}$
            edges ← edges ∪ (caller, ctx) → (callee, sub)
            reachableMethods ← reachableMethods ∪ (callee, sub)
          end if
        end for
    end for
    for all closure C ∈ $\text{summary}_{\text{caller}}$ **do**
      reachableTypes ← reachableTypes ∪ C
    end for
  end for
end while

Besides, we do not need to handle specially constructs like those presented in the code below because they are transformed into `Match` trees before the call graph is built:

```
val Twice(x) = 4
val f: PartialFunction[Int, Int] = { case x => 2 * x }
```

However, patterns in cases of a `try catch` construct which require `unapply` calls must be dealt with specially. Since the selector is not explicitly written, we must transform our original tree:

```
try {
  throw ...
} catch {
  case Extractor(x) => ...
}
```

into the following pattern matching construct, where `ex` is a synthetically created selector of `Throwable` type representing the thrown exception.

```
ex match {
  case Extractor(x) => ...
}
```

We then simply register implicit calls as described above on this new tree.

4.2 Adding complete support for Closures

Before this contribution, closures were partially supported. They would only be tracked by the call graph algorithm when passed as argument in a call, such as in `lst.map(x => 2 * x)`. However, consider the following code:

Figure 7. Call graph with no closure support

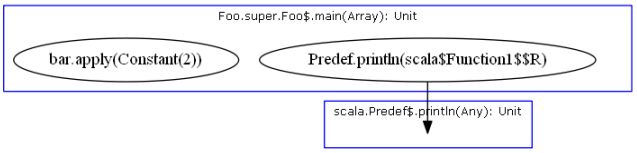
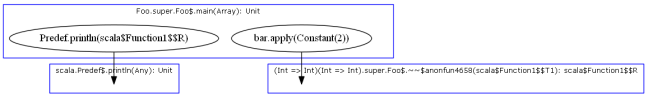


Figure 8. Call graph with closure support




```
object Foo {
  def main(args: Array[String]): Unit = {
    val bar = (x: Int) => x
    println(bar(2))
  }
}
```

The call graph for this example is as shown in figure 7. When compiling the program with dead code elimination activated, we get:

```
Exception in thread "main"
   dotty.runtime.DeadCodeEliminated
       at Foo$.$anonfun$1(Test.scala:137)
       at ...
       at Foo$.main(Test.scala:138)
       at Foo.main(Test.scala)
```

Unlike for the pattern matching issues, here the call is explicitly done in `bar(2)`. The reason our closure is not found reachable is simply because it looks as if it was never instantiated to the call graph algorithm. Recall from algorithm 2 that a type is considered reachable if and only if there is an explicit call to its constructor somewhere (this is a slightly simplified, i.e. things are done differently for modules). Since at this point in the compilation closures are still not in their final form, they do not have a constructor available to be called, and therefore it is not possible to simply add a call to a constructor in the method's summary. Instead, I chose to modify the `MethodSummary` class so that it can store a collection of closures that are defined in the method. Then it required handling these closures specially when generating the call graph, as can be seen in algorithm 3. Basically we assume that a closure becomes reachable as soon as its enclosing method becomes reachable. We could be more precise about this in the future but it is clearly not a priority right now. The reason it solves the problem is simply that since the closure type becomes part of the $reachableTypes$ at some point, all the conditions required for its implemented method to be considered reachable when processing a call to the closure will be satisfied. At this point, the example program runs correctly and the call graph is as shown in figure 8.

Unfortunately, there is a flow with this simple fix which is illustrated by the following example:

```
object Foo {
  def foo[T] = (x: T) => x
  def main(args: Array[String]): Unit = {
    val bar = foo[Int]
    println(bar(2))
  }
}
```

Once more, a `DeadCodeEliminated` exception will be thrown when trying to run it. The reason is not obvious: here, the method

`apply` implemented by the closure defined in `foo` has signature `T => T`, but the `apply` call done in `bar(2)` expects a method with signature `Int => Int`. Taking a look at algorithm 3, we can notice that the condition $sig(\text{o}) = sig(\text{callee})$ is not satisfied for this reason exactly, preventing our closure from becoming reachable. To fix this, we should use the context `ctx` of the enclosing method when adding our closure to the set of reachable types. In our example, the context at this moment is $\{T \rightarrow Int\}$. Therefore, we should simply apply this context to the type of our closure so that its implemented method has type `Int => Int` instead of `T => T` when added to $reachableTypes$. More formally, assuming the implemented method of the closure has type $(T_1, ..., T_{n-1}) \Rightarrow T_n$, we construct a new closure type which implemented method has type $(R_1, ..., R_{n-1}) \Rightarrow R_n$ where $R_1, ..., R_n$ is defined as:

$$R_i = \begin{cases} S & \text{if } \exists S : (T_i \rightarrow S) \in \text{ctx} \\ T_i & \text{otherwise} \end{cases} \qquad \forall 1 \leq i \leq n$$

Finally, we simply add this new closure to the set of reachable types instead of the original one and it will successfully become reachable since the signatures will match.

5. Future Work

Adding to the compiler toolbox a way of generating precise call graphs for Scala programs lays the foundations for further analyses and new specific optimizations based on it. On top of the few optimizations mentioned in section 2, we can cite inlining and devirtualization. Most importantly, it provides a strong basis upon which brand-new, Scala-specific optimizations can be thought of.

Hence, the current goal is to have a stable call graph construction algorithm which can handle all Scala features while being as precise as possible. As a matter of fact, there is a lot of room for improvement to make it the most precise possible. For example, some specific flow sensitive analyses could help further reduce the size of the call graph such as pointer analysis, which could be used to limit the number of out-edges from a callsite subject to dynamic dispatch by using our knowledge of the pointer and discarding infeasible calls.

Additionally, dead code elimination and auto specialization are two optimizations that are being developed in parallel to the call graph algorithm. Paper [11] already presents very motivating results.

6. Conclusion

Working with Dotty was a very interesting and fulfilling experience. I could discover the internals of the compiler and understand many concepts while working on a very ambitious project doing concrete work. I was able to discover many bugs and edge cases throughout my journey, which combined with hours of diving into the complex code-base of the call graph construction algorithm and extensive periods of debugging finally allowed me to get a solid enough grasp of its mechanisms, thanks to which I could successfully implement support for pattern matching and complete support for closures.

References

[1] Dotty-linker project, . URL `https://github.com/dotty-linker/dotty`.

[2] Call graph with dce, pull request, . URL `https://github.com/lampepfl/dotty/pull/1840`.

[3] Scala Programming Language. URL `http://scala-lang.org/`.

[4] I. Dragos and M. Odersky. Compiling Generics Through User-Directed Type Specialization. In *ICOOOLPS*, Genova, Italy, 2009.

[5] B. Emir. *Object-oriented pattern matching*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2007. URL `https://infoscience.epfl.ch/record/109881/`.

[6] S. Milton and H. W. Schmidt. Dynamic dispatch in object-oriented languages. Technical report, 1994. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.5597&rep=rep1&type=pdf`.

[7] A. Moors. *Type Constructor Polymorphism for Scala: Theory and Practice (Type constructor polymorfisme voor Scala: theorie en praktijk)*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, 2009. URL `https://lirias.kuleuven.be/handle/1979/2642`. Joosen, Wouter and Piessens, Frank (supervisors).

[8] M. Odersky. *The Scala Language Specification Version 2.9*. 2014. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[9] D. Petrashko. Dotty linker: Making your scala applications smaller and faster, 2015. URL `https://d-d.me/talks/scaladays2015/#/`.

[10] D. Petrashko. Autospecialization in dotty, 2016. URL `https://d-d.me/talks/flatmap2016/#/`.

[11] D. Petrashko, V. Ureche, O. Lhoták, and M. Odersky. Call graphs for languages with parametric polymorphism. Technical report, EPFL, 2016. https://infoscience.epfl.ch/record/217276.

[12] M. Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005.

[13] V. Ureche, C. Talau, and M. Odersky. Miniboxing: Improving the Speed to Code Size Tradeoff in Parametric Polymorphism Translations. In *OOPSLA*, 2013.