Call-graph-based Optimizations in Scala

Romain Beguet
Specialization

class ArrayBuffer[@specialized T] {
  def append(x: T) = ...
}

Specialization

class ArrayBuffer[@specialized T] {
    def append(x: T) = ...
}

… and 7 more
Open-World Compilation

arraybuffer.scala

```scala
class ArrayBuffer[@specialized T] {
  def append(x: T) = ...
}
def foo = new ArrayBuffer[Int]
```

output

```scala
class IntArrayBuffer {
  def append(x: Int) = ...
}
```
Open-World Compilation

arraybuffer.scala

```scala
class ArrayBuffer[@specialized T] {
  def append(x: T) = ...
}
def foo = new ArrayBuffer[Int]
```

compilation

class IntArrayBuffer {
  def append(x: Int) = ...
}

output

```
class IntArrayBuffer {
  def append(x: Int) = ...
}
```

someprog.scala

```scala
def bar = new ArrayBuffer[Double]
```

compilation

Missing specialization Double ArrayBuffer
Dotty-Linker

- Dotty with additional link-time features
- Can take as input scala or TASTY source files
Dotty vs Dotty-Linker

- .scala
- .tasty
  → dotty
  → .jar
  library ?
  executable ?

- .scala
- .tasty
  → dotty-linker
  → final executable .jar
Dotty-Linker

- Dotty with additional link-time features
- Can take as input scala or TASTY source files
- Used as the final step towards releasing an executable
Dotty-Linker

- Dotty with additional link-time features
- Can take as input scala or TASTY source files
- Used as the final step towards releasing an executable
- Performs compilation under a closed-world assumption
Dotty-Linker

- Dotty with additional link-time features
- Can take as input scala or TASTY source files
- Used as the final step towards releasing an executable
- Performs compilation under a closed-world assumption
  - What does this assumption unlock?
Open-World Compilation

csomelib.scala

```scala
class ArrayBuffer[@specialized T] {
  def append(x: T) = ...
}

def foo = new ArrayBuffer[Int]
```

```
someprog.scala

def bar = new ArrayBuffer[Double]
```
Open-World Compilation

somelib.scala

```scala
class ArrayBuffer[@specialized T] {
  def append(x: T) = ...
}

def foo = new ArrayBuffer[Int]
```

compilation

```scala
class ObjectArrayBuffer {
  def append(x: Object) = ...
}

class IntArrayBuffer {
  def append(x: Int) = ...
}

class FloatArrayBuffer {
  def append(x: Float) = ...
}

... and 7 more
```

someprog.scala

```scala
def bar = new ArrayBuffer[Double]
```
Closed-World Compilation

somelib.scala

class ArrayBuffer[@specialized T] {
  def append(x: T) = ...
}
def foo = new ArrayBuffer[Int]

someprog.scala

def bar = new ArrayBuffer[Double]
Closed-World Compilation

```
somelib.scala

class ArrayBuffer[@specialized T] {
  def append(x: T) = ...
}
def foo = new ArrayBuffer[Int]

output

class IntArrayBuffer {
  def append(x: Int) = ...
}
class DoubleArrayBuffer {
  def append(x: Int) = ...
}
```

```
someprog.scala

def bar = new ArrayBuffer[Double]
```
Closed-World Compilation

class ArrayBuffer[T] {
  def append(x: T) = ... 
}

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

def main(...) = {
  val x = foo[Int]
  ...
}

output

compilation ?
class ArrayBuffer[T] {
  def append(x: T) = ...
}

def foo[U] = new ArrayBuffer[U]
def bar = new ArrayBuffer[Double]
def main(...) = {
  val x = foo[Int]
  ...
}
class ArrayBuffer[T] {
  def append(x: T) = ...
}

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

def main(...) = {
  val x = foo[Int]
  ...
}

Call graph

main → foo[Int] → ArrayBuffer[Int]
class ArrayBuffer[T] {
  def append(x: T) = ...
}

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

def main(...) = {
  val x = foo[Int]
  ...
}
class ArrayBuffer[T] {
  def append(x: T) = ...
}

def foo[U] = new ArrayBuffer[U]
def bar = new ArrayBuffer[Double]
def main(...) = {
  val x = foo[Int]
  ...
}

class IntArrayBuffer {
  def append(x: Int) = ...
}

...
Call graph in Dotty-Linker

- Collect Method Summaries
class ArrayBuffer[T] {
  def append(x: T) = ...
}

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

def main(...) = {
  val x = foo[Int]
  ...
}

Collect Summaries
class ArrayBuffer[T] {
    def append(x: T) = ... 
}

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

def main(...) = {
    val x = foo[Int]
    ...
}

---

foo:
- ArrayBuffer[U]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]
Call graph in Dotty-Linker

- Collect Method Summaries
- Build the call graph
Build the Call Graph

```plaintext
foo:
  - ArrayBuffer[U]

bar:
  - ArrayBuffer[Double]

main:
  - foo[Int]
```
Build the Call Graph

---

**foo:**
- `ArrayBuffer[U]`

**bar:**
- `ArrayBuffer[Double]`

**main:**
- `foo[Int]`
Build the Call Graph

- foo:
  - ArrayBuffer[U]

- bar:
  - ArrayBuffer[Double]

- main:
  - foo[Int]
Build the Call Graph

foo:
- ArrayBuffer[U]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]
Build the Call Graph

- **foo**: `ArrayBuffer[U]`
- **bar**: `ArrayBuffer[Double]`
- **main**: `foo[Int]`
Build the Call Graph

foo:
- ArrayBuffer[U]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]

Diagram:
- ArrayBuffer
  - foo
    - {U ⇒ Int}
  - main
    - {T ⇒ Int}
Build the Call Graph

**foo:**
- ArrayBuffer\[U\]

**bar:**
- ArrayBuffer[Double]

**main:**
- foo[Int]

\[ E = \{(main, foo, \{U \Rightarrow \text{Int}\}),
(foo, ArrayBuffer, \{T \Rightarrow \text{Int}\})\} \]
Call-graph-based Optimizations

\[ E = \{ 
    (\text{main}, \text{foo}, \{U \Rightarrow \text{Int}\}), 
    (\text{foo}, \text{ArrayBuffer}, \{T \Rightarrow \text{Int}\}) 
\} \]
Call-graph-based Optimizations

- Dead Code Elimination

\[
E = \{ \\
    (main, foo, \{U \Rightarrow \text{Int}\}), \\
    (foo, ArrayBuffer, \{T \Rightarrow \text{Int}\}) \\
\} 
\]
Call-graph-based Optimizations

- Dead Code Elimination
  - Remove methods that do not have any incoming edges

\[ E = \{ (main, foo, \{U \Rightarrow \text{Int}\}), (foo, \text{ArrayBuffer}, \{T \Rightarrow \text{Int}\}) \} \]
Call-graph-based Optimizations

● Dead Code Elimination
  ○ Remove methods that do not have any incoming edges
  ○ Formally, keep a method $\mu$ iff:

\[ \exists \alpha, \Sigma \mid (\alpha, \mu, \Sigma) \in E \]

\[ E = \{ \]
\[ (\text{main, foo, \{U \Rightarrow \text{Int}\}}), \]
\[ (\text{foo, ArrayBuffer, \{T \Rightarrow \text{Int}\}}) \]
\[ \} \]
Call-graph-based Optimizations

- Dead Code Elimination
  - Remove methods that do not have any incoming edges
  - Formally, keep a method $\mu$ iff:

  $$\exists \alpha, \Sigma \mid (\alpha, \mu, \Sigma) \in E$$

  $E = \{
  (main, foo, \{U \Rightarrow \text{Int}\}),
  (foo, ArrayBuffer, \{T \Rightarrow \text{Int}\})
  \}$
Call-graph-based Optimizations

- Dead Code Elimination
  - Remove methods that do not have any incoming edges
  - Formally, keep a method $\mu$ iff:

\[
\exists \alpha, \Sigma \mid (\alpha, \mu, \Sigma) \in E
\]

```
E = {
  (main, foo, {U ⇒ Int}),
  (foo, ArrayBuffer, {T ⇒ Int})
}
```
Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types

\[
E = \{(\text{main, foo}, \{U \Rightarrow \text{Int}\}),
(\text{foo, ArrayBuffer}, \{T \Rightarrow \text{Int}\})\}
\]
Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types
  - Generate only needed specializations
  - Without any manual annotation

$$E = \{ (main, foo, \{U \Rightarrow \text{Int}\}),
\ (foo, \text{ArrayBuffer}, \{T \Rightarrow \text{Int}\}) \}$$
Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types
  - Generate only needed specializations
  - Without any manual annotation
  - Formally, generate variants of $\mu$ for all contexts in:

$$\left\{ \Sigma \mid \exists \alpha : (\alpha, \mu, \Sigma) \in E \right\}$$

E = {
  (main, foo, {U ⇒ Int}),
  (foo, ArrayBuffer, {T ⇒ Int})
}
Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types
  - Generate only needed specializations
  - Without any manual annotation
  - Formally, generate variants of $\mu$ for all contexts in:

\[
\left\{ \sum \mid \exists \alpha : (\alpha, \mu, \Sigma) \in E \right\}
\]

E = {
  (main, foo, {U ⇒ Int}),
  (foo, ArrayBuffer, {T ⇒ Int})
}
Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types
  - Generate only needed specializations
  - Without any manual annotation
  - Formally, generate variants of $\mu$ for all contexts in:
    \[ \{ \Sigma \mid \exists \alpha : (\alpha, \mu, \Sigma) \in E \} \]
    
    \[
    \begin{align*}
    foo \ ? & \{ \{U \Rightarrow \text{Int}\} \} \\
    ArrayBuffer \ ? & \{ \{T \Rightarrow \text{Int}\} \}
    \end{align*}
    \]

\[ E = \{ \\
  (main, foo, \{U \Rightarrow \text{Int}\}), \\
  (foo, ArrayBuffer, \{T \Rightarrow \text{Int}\})
\} \]
Call-graph-based Optimizations

● Dead Code Elimination
● Auto Specialization for types
  ○ Generate only needed specializations
  ○ Without any manual annotation
  ○ Formally, generate variants of $\mu$ for all contexts in:

\[ \{ \Sigma \mid \exists \alpha : (\alpha, \mu, \Sigma) \in E \} \]

\[
\begin{align*}
  &\text{foo} \, ? \{ \{U \Rightarrow \text{Int}\} \} \quad \text{def foo_int = ...} \\
  &\text{ArrayBuffer} \, ? \{ \{T \Rightarrow \text{Int}\} \} \quad \text{def ArrayBuffer_int = ...}
\end{align*}
\]

\[
E = \{ \\
  (\text{main, foo, } \{U \Rightarrow \text{Int}\}), \\
  (\text{foo, ArrayBuffer, } \{T \Rightarrow \text{Int}\})
\} \]
Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types
- Auto Specialization for terms

\[ E = \{(\text{main}, \text{foo}, \{U \Rightarrow \text{Int}\}),
(\text{foo}, \text{ArrayBuffer}, \{T \Rightarrow \text{Int}\})\} \]
Auto Specialization for Terms

class A {
    def foo: Int = 1
}
class B extends A {
    override def foo: Int = 2
}

def bar(x: A) = x.foo

def main(...) = {
    val x = new A
    bar(new B)
    bar(new B)
    bar(x)
}
Auto Specialization for Terms

class A {
    def foo: Int = 1
}
class B extends A {
    override def foo: Int = 2
}
def bar(x: A) = x.foo
def main(...) = {
    val x = new A
    bar(new B)
    bar(x)
}
Auto Specialization for Terms

bar:
- x.foo

main:
- bar(new B)
- bar(x)
Auto Specialization for Terms

bar:
- \( x.\text{foo} \)

main:
-\(\text{bar}(\text{new}\ B)\)
-\(\text{bar}(x)\)
Auto Specialization for Terms

bar:
  - x.foo

main:
  - bar(new B)
  - bar(x)

\[
\text{bar} \Rightarrow \{x \Rightarrow \text{new B}\}
\]
Auto Specialization for Terms

bar:
- x.foo

main:
- bar(new B)
- bar(x)

A.foo

B.foo

bar

{x ⇒ new B}

main
Auto Specialization for Terms

**bar:**
- `x.foo`

**main:**
- `bar(new B)`
- `bar(x)`

```
main
  ↓
 bar
  ↓
̂  
 bar
  ↓
̂  
 A.foo
  ↓
̂  
 B.foo
```

```
main         {x ⇒ new B}
  ↓
 bar
  ↓
̂  
 bar
  ↓
̂  
 x.foo
  ↓
̂  
 B.foo
```

```
main
  ↓
̂  
 bar
  ↓
̂  
 x.foo
  ↓
̂  
 B.foo
```

```
main
  ↓
̂  
 bar
  ↓
̂  
 x.foo
  ↓
̂  
 B.foo
```
Auto Specialization for Terms

**bar**:  
- x.foo

**main**:  
- bar(new B)  
- bar(x)
class A {
    def foo: Int = 1
}
class B extends A {
    override def foo: Int = 2
}
def bar(x: A) = x.foo
def main(...) = {
    val x = new A
    bar(new B)
    bar(x)
}
class A {
    def foo: Int = 1
}

class B extends A {
    override def foo: Int = 2
}

def bar(x: A) = x.foo

def main(...) = {
    val x = new A
    bar(new B)
    bar(x)
}

class A {
    def foo: Int = 1
}

class B extends A {
    override def foo: Int = 2
    static def foo_impl(x: B): Int = 2
}

def bar_B(x: B) = B.foo_impl(x)

def main(...) = {
    val x = new A
    bar_B(new B)
    bar(x)
}
class A {
  def foo: Int = 1
}
class B extends A {
  override def foo: Int = 2
}
def bar(x: A) = x.foo
def main(...) = {
  val x = new A
  bar_B(new B)
  bar(x)
}

class A {
  def foo: Int = 1
}
class B extends A {
  override def foo: Int = 2
  static def foo_impl(x: B): Int = 2
}
def bar_B(x: B) = 2
def main(...) = {
  val x = new A
  bar_B(new B)
  bar(x)
}
Auto Specialization for Terms

class A {
  def foo: Int = 1
}
class B extends A {
  override def foo: Int = 2
}
def bar(x: A) = x.foo
def main(...) = {
  val x = new A
  bar(new B)
  bar(x)
}

class A {
  def foo: Int = 1
}
class B extends A {
  override def foo: Int = 2
  static def foo_impl(x: B): Int = 2
}
def bar(x: A) = x.foo
def bar_B(x: B) = 2
def main(...) = {
  val x = new A
  2
  bar(x)
}
Auto Specialization for Terms

class A {
    def foo: Int = 1
}
class B extends A {
    override def foo: Int = 2
}
def bar(x: A) = x.foo
def main(...) = {
    val x = new A
    bar(new B)
    bar(x)
}
Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types
- Auto Specialization for terms

\[ E = \{(main, foo, \{U \Rightarrow \text{Int}\}), (foo, ArrayBuffer, \{T \Rightarrow \text{Int}\})\} \]
Contributions
Contributions

- Adding support for pattern matching in the callgraph
case class Foo(x: Int)

def main(args: Array[String]): Unit = {
  Foo(42) match {
    case Foo(x) => println(x)
  }
}
case class Foo(x: Int)

def main(args: Array[String]): Unit = {
  Foo(42) match {
    case Foo(x) => println(x)
  }
}

Pattern Matching
case class Foo(x: Int)

def main(args: Array[String]): Unit = {
  Foo(42) match {
    case Foo(x) => println(x)
  }
}
Contributions

- Adding support for pattern matching in the callgraph
  - Make a procedure to generate unapply calls
  - Need to differentiate 5 cases
    i. unapply returns a Boolean
    ii. unapply returns Option[T]
    iii. unapply returns Option[(T₁, ..., Tₙ)]
    iv. unapply returns ProductN[T₁, ..., Tₙ]
    v. unapplySeq returns Option[Seq[T]]
Pattern Matching

case class Foo(x: Int)

def main(args: Array[String]): Unit = {
  Foo(42) match {
    case Foo(x) => println(x)
  }
}
Contributions

- Adding support for pattern matching in the callgraph
  - Make a procedure to generate unapply calls
  - Need to differentiate 5 cases
  - Need to handle nested patterns
Pattern Matching

```scala
object Foo {
  def unapply(x: Int) = Some((x, 2*x))
}

object Bar {
  def unapply(x: Int) = Some(x)
}

def main(args: Array[String]): Unit = {
  42 match {
    case Foo(42, Bar(x)) => println(x)
  }
}
```
object Foo {
    def unapply(x: Int) = Some((x, 2*x))
}
object Bar {
    def unapply(x: Int) = Some(x)
}
def main(args: Array[String]): Unit = {
    42 match {
        case Foo(42, Bar(x)) => println(x)
    }
}
def main(args: Array[String]): Unit = {
  42 match {
    case Foo(42, Bar(x)) => println(x)
  }
}
def main(args: Array[String]): Unit = {
  42 match {
    case Foo(42, Bar(x)) => println(x)
  }
}

val tmp1 = Foo.unapply(42)
if (!tmp1.isEmpty) {
  ...
  tmp1.get._2 match {
    case Bar(x) => println(x)
  }
}
def main(args: Array[String]): Unit = {
  42 match {
    case Foo(42, Bar(x)) => println(x)
  }
}

val tmp1 = Foo.unapply(42)
if (!tmp1.isEmpty) {
  ...
  tmp1.get._2 match {
    case Bar(x) => println(x)
  }
}
def main(args: Array[String]): Unit = {
  42 match {
    case Foo(42, Bar(x)) => println(x)
  }
}

val tmp1 = Foo.unapply(42)
if (!tmp1.isEmpty) {
  ...
  tmp1.get._2 match {
    case Bar(x) => println(x)
  }
}
Contributions

- Adding support for pattern matching in the callgraph
- Completing support for closures
Closures

- Initially only partially supported

```scala
def foo(f: Int=>Int) = {
  f(42)
}

def main(...) = {
  foo(x => 2*x)
}
```
Closures

- Initially only partially supported

```scala
def foo(f: Int=>Int) = {
  f(42)
}
def main(...) = {
  foo(x => 2*x)
}
```
Closures

- Initially only partially supported

```scala
def foo(f: Int=>Int) = {
  f(42)
}

def main(...) = {
  foo(x => 2*x)
}
```

```
main.anonfun$1

foo

{f ⇒ (x => 2*x)}

main
```
Closures

- Initially only partially supported

```scala
def foo(f: Int=>Int) = {
  f(42)
}

def main(...) = {
  val f = (x: Int) => 2 *x
  foo(f)
}
```
Closures

- Initially only partially supported

```scala
def foo(f: Int=>Int) = {
  f(42)
}
def main(...) = {
  val f = (x: Int) => 2 *x
  foo(f)
}
```
Closures

- Initially only partially supported
- In each method summary, store its closures
Closures

- Initially only partially supported
- In each method summary, store its closures

```scala
def foo(f: Int=>Int) = {
  f(42)
}
def main(...) = {
  val f = (x: Int) => 2 *x
  foo(f)
}
```
Closures

- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program

```
main:
  • foo(f)
  □ main.anonfun$1

foo:
  • f(42)
```
Closures

- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program

```
foo:
  • f(42)

main:
  • foo(f)
  ❏ main.anonfun$1
```

Closures
- main.anonfun$1

main
Closures

- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program

```
foo:
  • f(42)

main:
  • foo(f)
  ❑ main.anonfun$1
```

Closures
- main.anonfun$1

- foo

- main
Closures

- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program

```scala
foo:  
  • f(42)

main:  
  • foo(f)  
    □ main.anonfun$1
```

f has static type `Function1[Int, Int]`
Closures

- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program
- On “FunctionX[...].apply”, assume call to our closures

```
foo:
  • f(42)

main:
  • foo(f)
  □ main.anonfun$1

Closures
- main.anonfun$1
```

Diagram:
- foo
- main.anonfun$1
- main
Closures

- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program
- On “FunctionX[...].apply”, assume call to our closures
- Problem: closures defined across the program are considered called
Closures

- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program
- On “FunctionX[...].apply”, assume call to our closures
- Problem: closures defined across the program are considered called
  - Stronger analysis needed!
The End

Thanks!