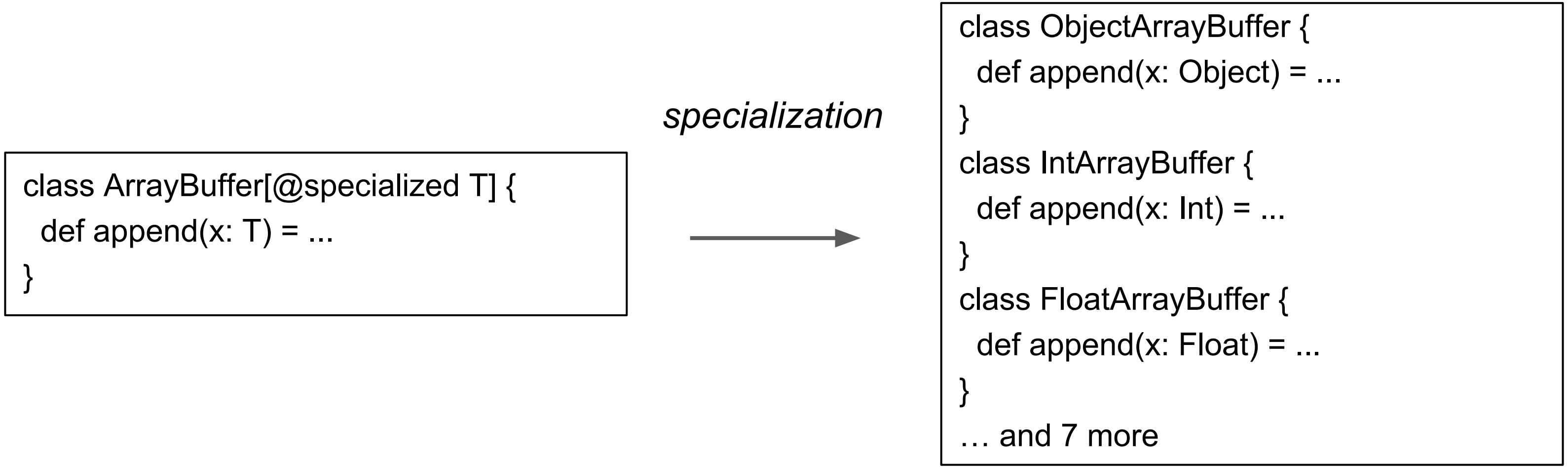
Call-graph-based Optimizations in Scala

Romain Beguet

Specialization

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

Specialization



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

specialization

```
class ObjectArrayBuffer {
  def append(x: Object) = ...
}
class IntArrayBuffer {
  def append(x: Int) = ...
}
class FloatArrayBuffer {
  def append(x: Float) = ...
}
… and 7 more
```

Open-World Compilation

arraybuffer.scala

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

compilation

output

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

Open-World Compilation

arraybuffer.scala

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

compilation →

output

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

...

someprog.scala

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

compilation →

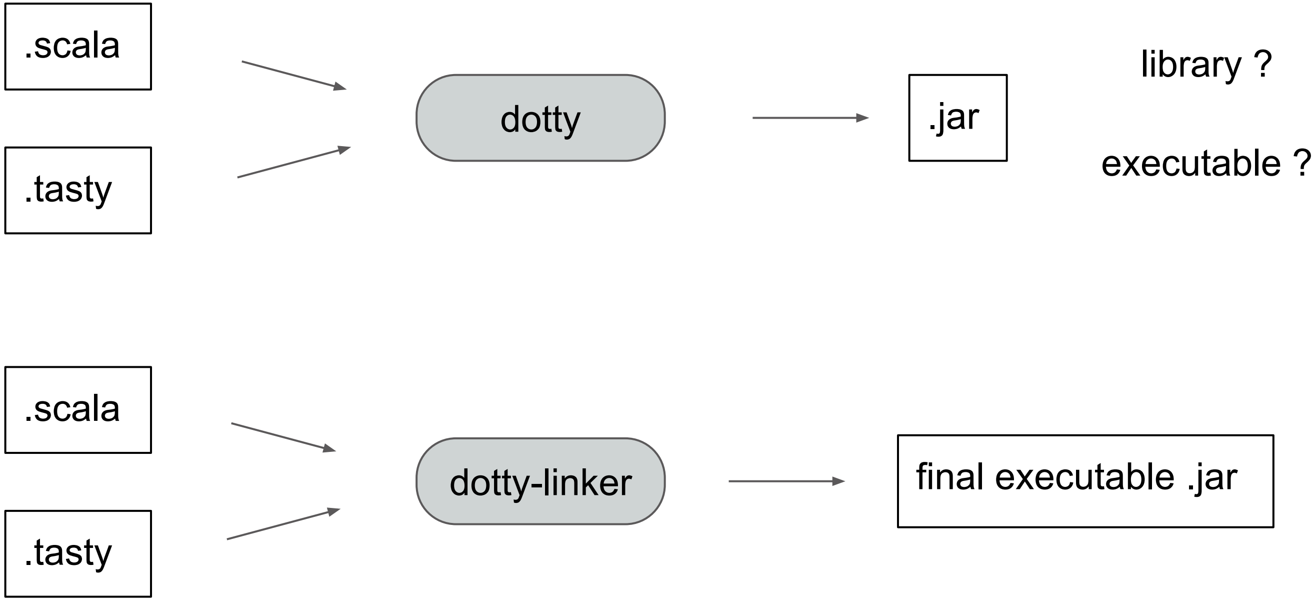
Missing specialization
DoubleArrayBuffer

Dotty-Linker

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

somelib.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

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

someprog.scala

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

compilation →

```
class ObjectArrayBuffer {
  def append(x: Object) = ...
}
class IntArrayBuffer {
  def append(x: Int) = ...
}
class FloatArrayBuffer {
  def append(x: Float) = ...
}
… and 7 more
```

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]
```

someprog.scala

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

compilation →

output

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

Closed-World Compilation

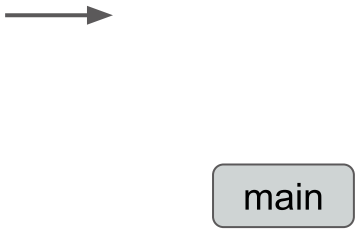
```
class ArrayBuffer[T] {
  def append(x: T) = ...
}

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

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

compilation →

output
?

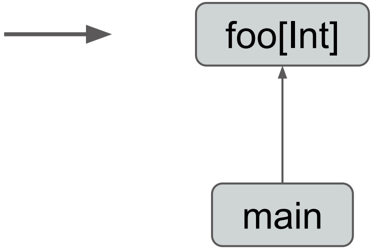
Call graph

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

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

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



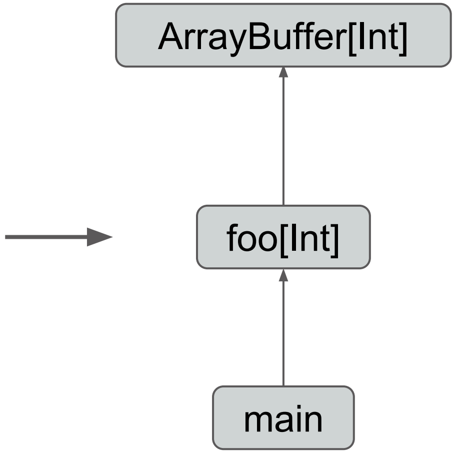
Call graph

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

def foo[U] = new ArrayBuffer[U]

def bar = new ArrayBuffer[Double]

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



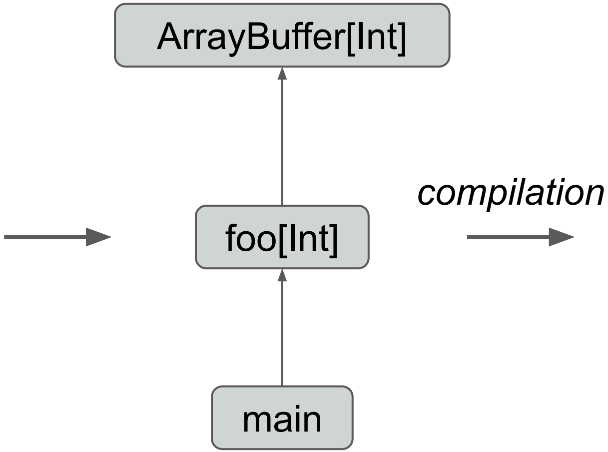
Call graph

```
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

```
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

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

Call graph in Dotty-Linker

- Collect Method Summaries

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]
  ...
}
```

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

foo:
- ArrayBuffer[*U*]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]

Build the Call Graph

foo:
- ArrayBuffer[*U*]

bar:
- ArrayBuffer[Double]

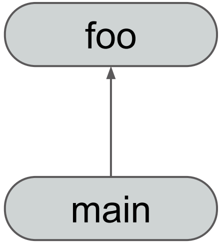
main:
- foo[Int]

Build the Call Graph

foo:
- ArrayBuffer[*U*]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]



Build the Call Graph

foo:

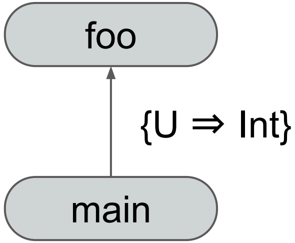
- ArrayBuffer[*U*]

bar:

- ArrayBuffer[Double]

main:

- foo[Int]

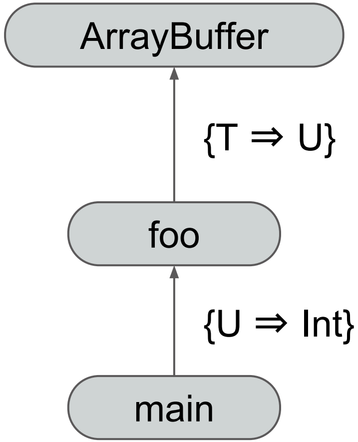


Build the Call Graph

foo:
- ArrayBuffer[*U*]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]

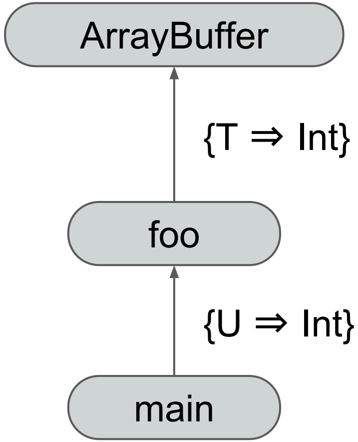


Build the Call Graph

foo:
- ArrayBuffer[*U*]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]

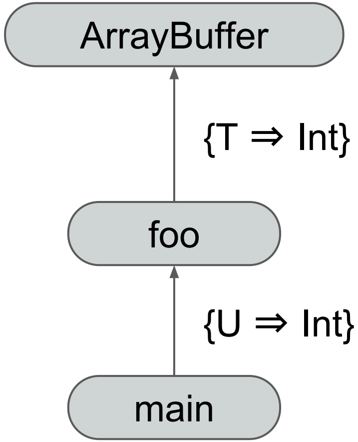


Build the Call Graph

foo:
- ArrayBuffer[*U*]

bar:
- ArrayBuffer[Double]

main:
- foo[Int]



E = {
  (*main*, *foo*, {U ⇒ Int}),
  (*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

E = {
    (*main*, *foo*, {U ⇒ Int}),
    (*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

- Dead Code Elimination

E = {
    (*main*, *foo*, {U ⇒ Int}),
    (*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

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

E = {
    (*main*, *foo*, {U ⇒ Int}),
    (*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

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

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

foo ?

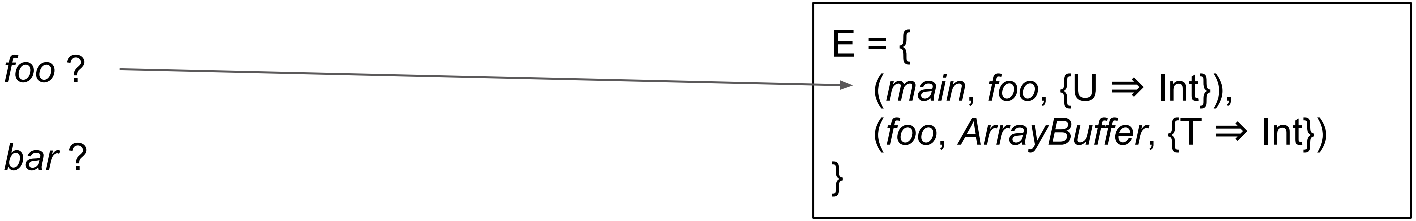
bar ?

E = {
    (*main*, *foo*, {U ⇒ Int}),
    (*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

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

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



foo ?

bar ?

E = {
(*main*, *foo*, {U ⇒ Int}),
(*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

- Dead Code Elimination
- Auto Specialization for types

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

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 μ for all contexts in:

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

E = {
  (*main*, *foo*, {U ⇒ Int}),
  (*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

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

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

foo ?

ArrayBuffer ?

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 μ for all contexts in:

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

foo ? { {U ⇒ Int} }

ArrayBuffer ? { {T ⇒ Int} }

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 *μ* for all contexts in:

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

foo ? { {U ⇒ Int} }

ArrayBuffer ? { {T ⇒ Int}
}

→

def foo_Int = …

def ArrayBuffer_Int = ...

E = {
(*main*, *foo*, {U ⇒ Int}),
(*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Call-graph-based Optimizations

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

E = {
  (*main*, *foo*, {U ⇒ Int}),
  (*foo*, *ArrayBuffer*, {T ⇒ 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(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)
}
```

bar:

- x.foo

main:

- bar(new B)
- bar(x)

Auto Specialization for Terms

bar:

- x.foo

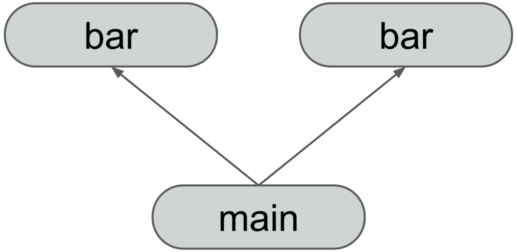
main:

- bar(new B)
- bar(x)

Auto Specialization for Terms

bar:
- x.foo

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



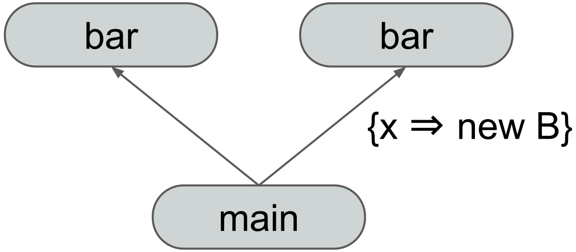
Auto Specialization for Terms

bar:

- x.foo

main:

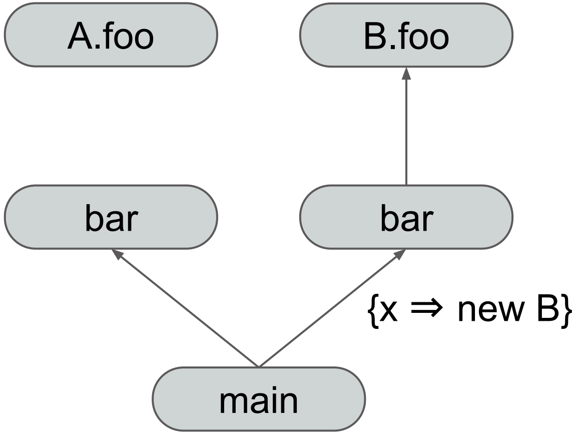
- bar(new B)
- bar(x)



Auto Specialization for Terms

bar:
- x.foo

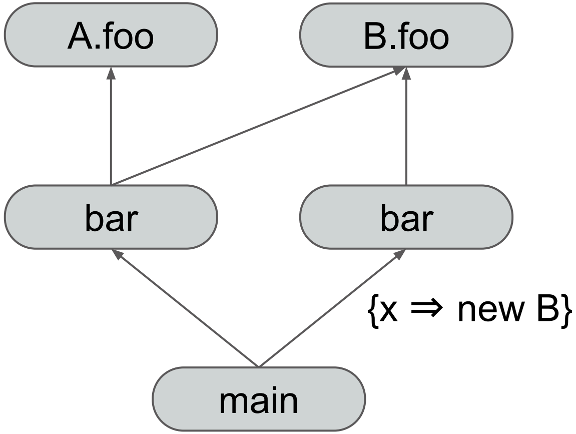
main:
- bar(new B)
- bar(x)



Auto Specialization for Terms

bar:
- x.foo

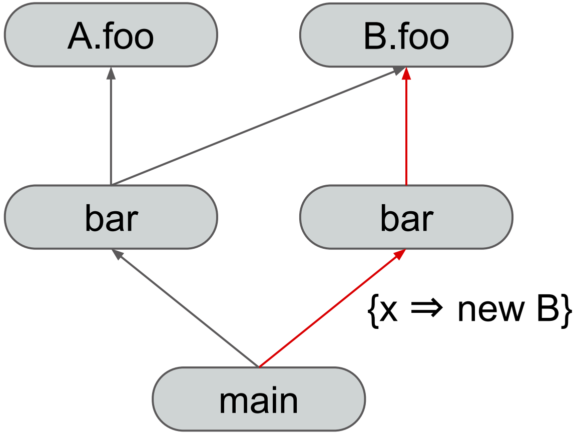
main:
- bar(new B)
- bar(x)



Auto Specialization for Terms

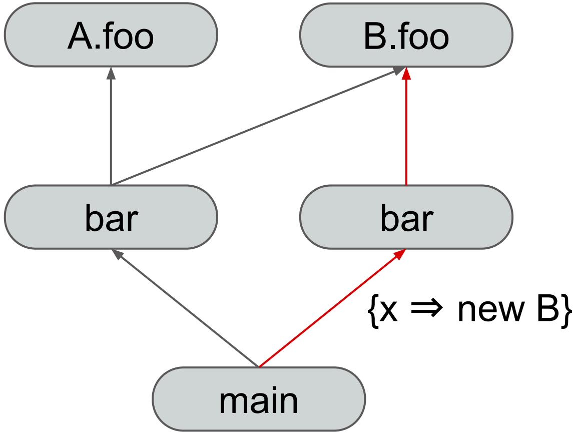
bar:
- x.foo

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



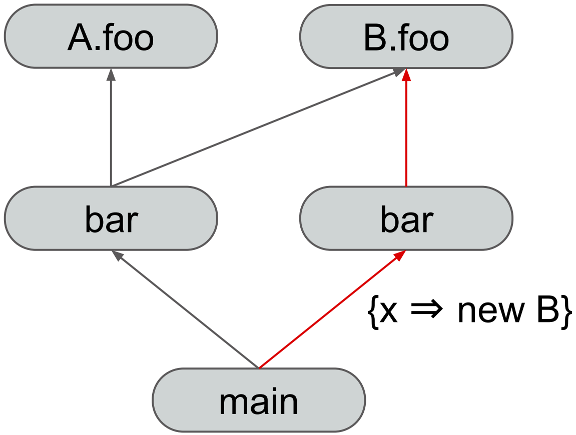
Auto Specialization for Terms

```
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

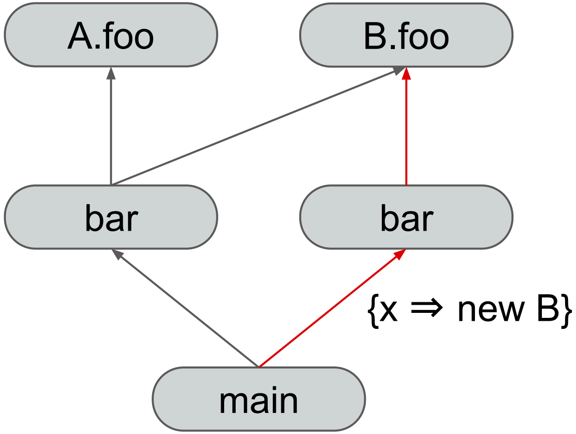
```
class A {
  def foo: Int = 1
}
class B extends A {
  override def foo: Int = 2
}

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) = B.foo_impl(x)

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

Auto Specialization for Terms

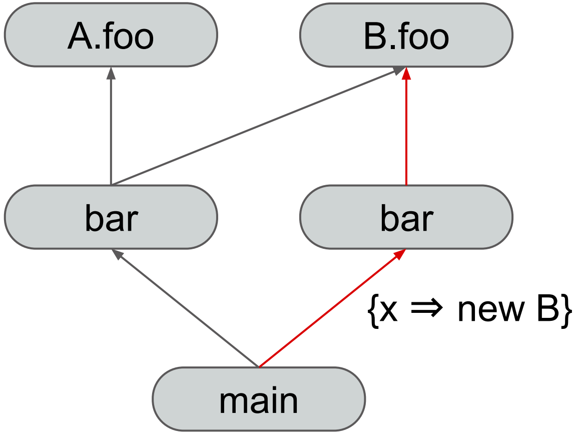
```
class A {
  def foo: Int = 1
}
class B extends A {
  override def foo: Int = 2
}

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
  bar_B(new B)
  bar(x)
}
```

Auto Specialization for Terms

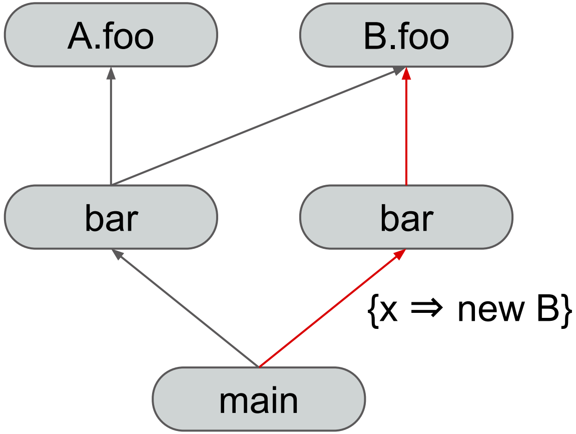
```
class A {
  def foo: Int = 1
}
class B extends A {
  override def foo: Int = 2
}

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)
}
```



```
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
  2
  bar(x)
}
```

Call-graph-based Optimizations

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

E = {
(*main*, *foo*, {U ⇒ Int}),
(*foo*, *ArrayBuffer*, {T ⇒ Int})
}

Contributions

Contributions

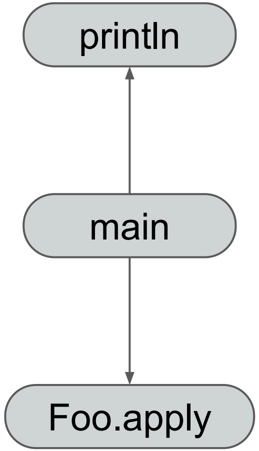
- Adding support for pattern matching in the callgraph

Pattern Matching

```
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)
  }
}
```



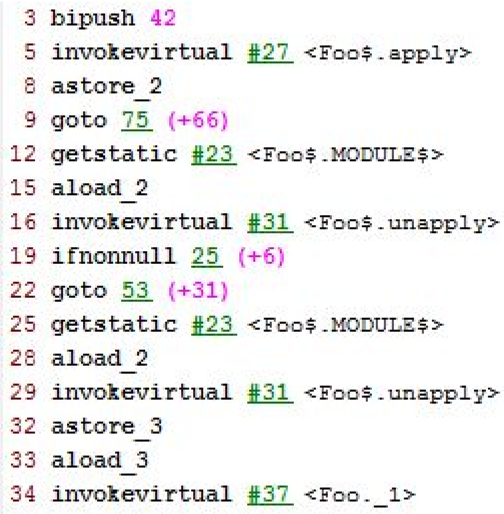
println

main

Foo.apply

Pattern Matching

```
case class Foo(x: Int)

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



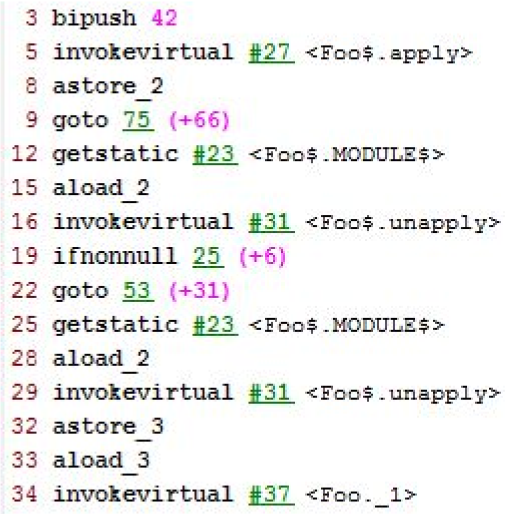
```
 3 bipush 42
 5 invokevirtual #27 <Foo$.apply>
 8 astore_2
 9 goto 75 (+66)
12 getstatic #23 <Foo$.MODULE$>
15 aload_2
16 invokevirtual #31 <Foo$.unapply>
19 ifnonnull 25 (+6)
22 goto 53 (+31)
25 getstatic #23 <Foo$.MODULE$>
28 aload_2
29 invokevirtual #31 <Foo$.unapply>
32 astore_3
33 aload_3
34 invokevirtual #37 <Foo._1>
```

Contributions

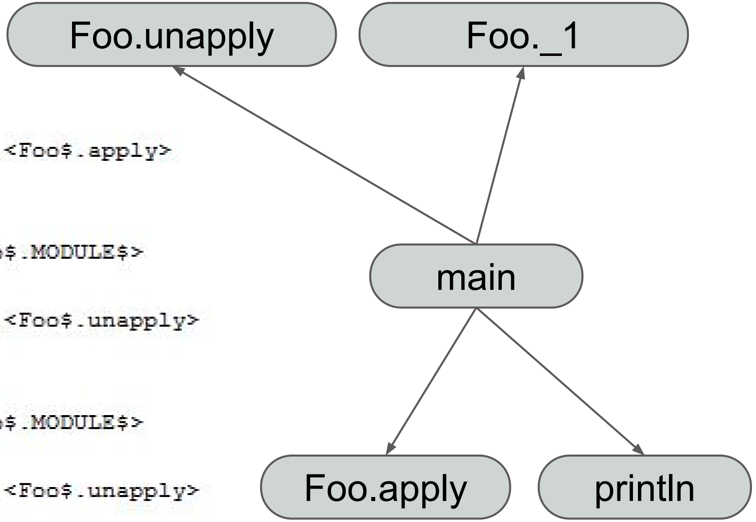
- Adding support for pattern matching in the callgraph
    - Make a procedure to generate unapply calls
    - Need to differentiate 5 cases
        1. unapply returns a Boolean
        2. unapply returns Option[T]
        3. unapply returns Option[(T_1, …, T_n)]
        4. unapply returns ProductN[T_1, …, T_n]
        5. 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)
  }
}
```



```
 3 bipush 42
 5 invokevirtual #27 <Foo$.apply>
 8 astore_2
 9 goto 75 (+66)
12 getstatic #23 <Foo$.MODULE$>
15 aload_2
16 invokevirtual #31 <Foo$.unapply>
19 ifnonnull 25 (+6)
22 goto 53 (+31)
25 getstatic #23 <Foo$.MODULE$>
28 aload_2
29 invokevirtual #31 <Foo$.unapply>
32 astore_3
33 aload_3
34 invokevirtual #37 <Foo._1>
```



Contributions

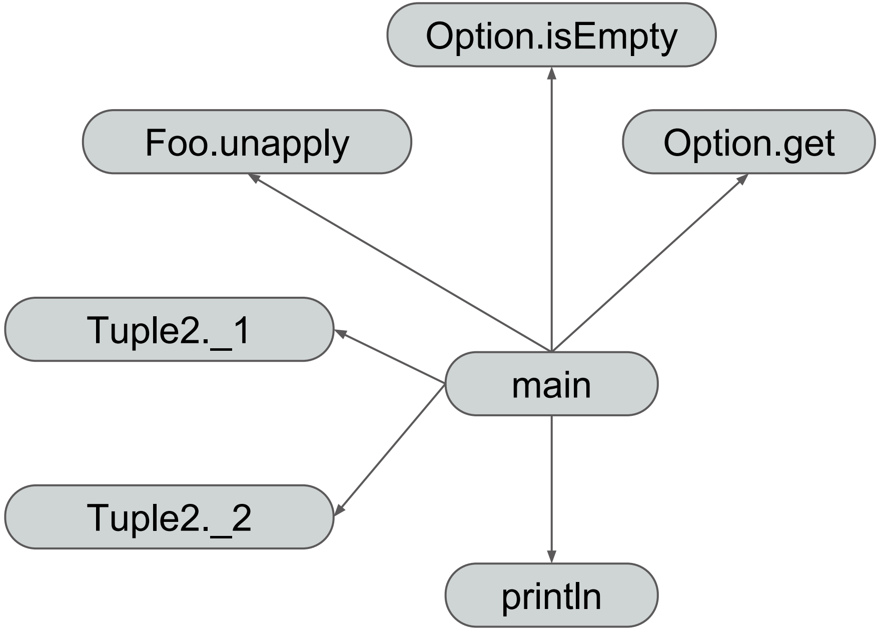
- Adding support for pattern matching in the callgraph
  - Make a procedure to generate unapply calls
  - Need to differentiate 5 cases
  - Need to handle nested patterns

Pattern Matching

```
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)
  }
}
```

Pattern Matching

```
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)
  }
}
```



Pattern Matching

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

Pattern Matching

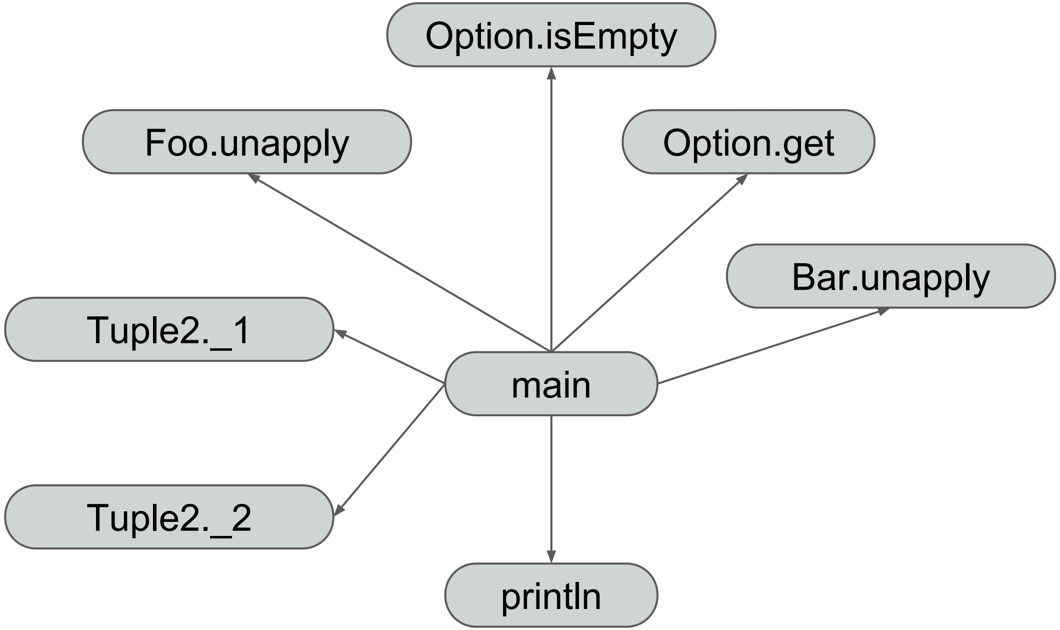
```
def main(args: Array[String]): Unit = {
  42 match {
     case Foo(42, Bar(x)) => println(x)
  }
}
```

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

Pattern Matching

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

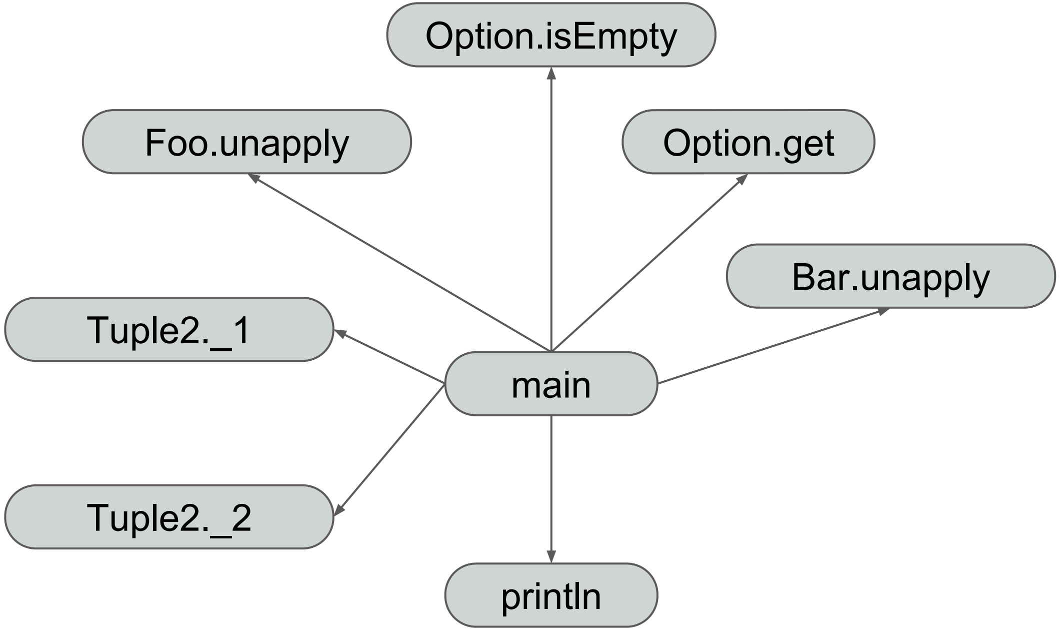
```
val tmp1 = Foo.unapply(42)
if (!tmp1.isEmpty)  {
  ...
  tmp1.get._2 match {
    case Bar(x) => println(x)
  }
}
```



Pattern Matching

```
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

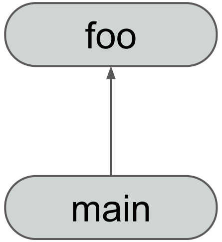
- Initially only partially supported

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

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

Closures

- Initially only partially supported

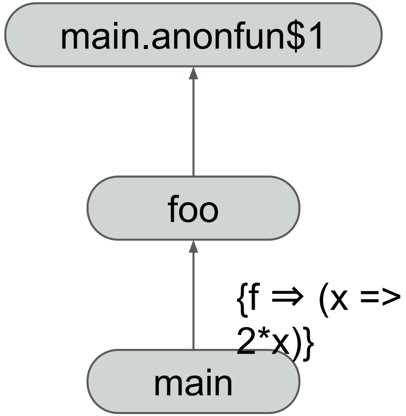
```
def foo(f: Int=>Int) = {
  f(42)
}

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



Closures

- Initially only partially supported

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

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



main.anonfun$1

foo

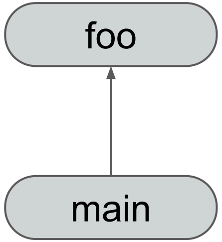
{f ⇒ (x => 2*x)}

main

Closures

- Initially only partially supported

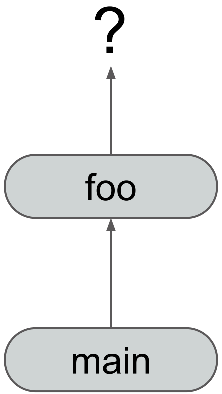
```
def foo(f: Int=>Int) = {
  f(42)
}

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



Closures

- Initially only partially supported

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

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

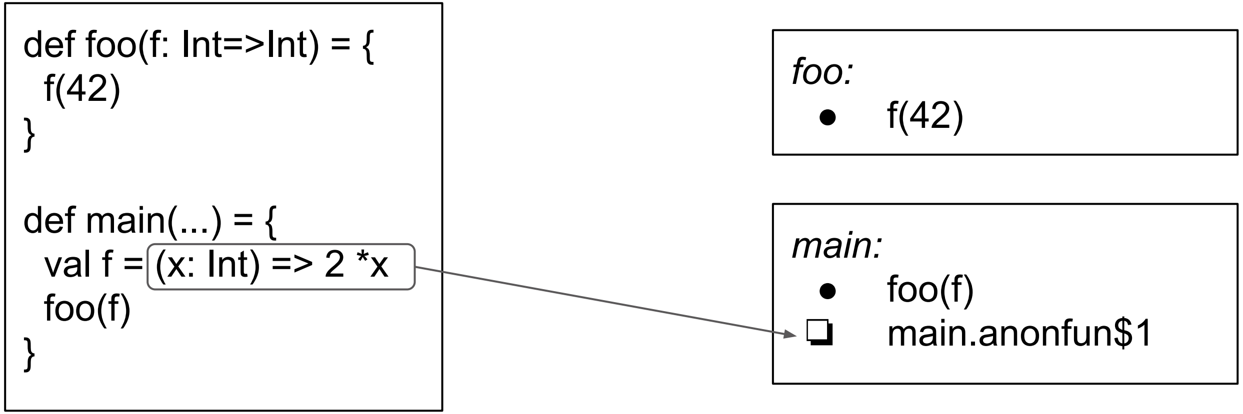


Closures

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

Closures

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



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

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

foo:
- f(42)

main:
- foo(f)

❏ main.anonfun$1

Closures

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

foo:
- f(42)

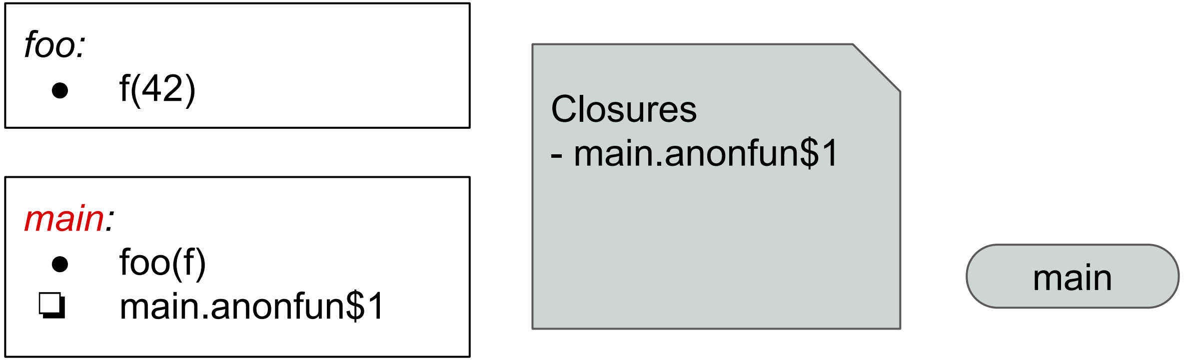
main:
- foo(f)

❏ main.anonfun$1

Closures

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

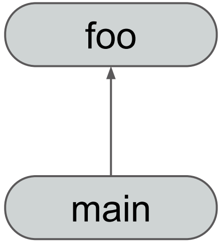
- Initially only partially supported
- In each method summary, store its closures
- Keep track of all reachable closures in the program

foo:
- f(42)

main:
- foo(f)

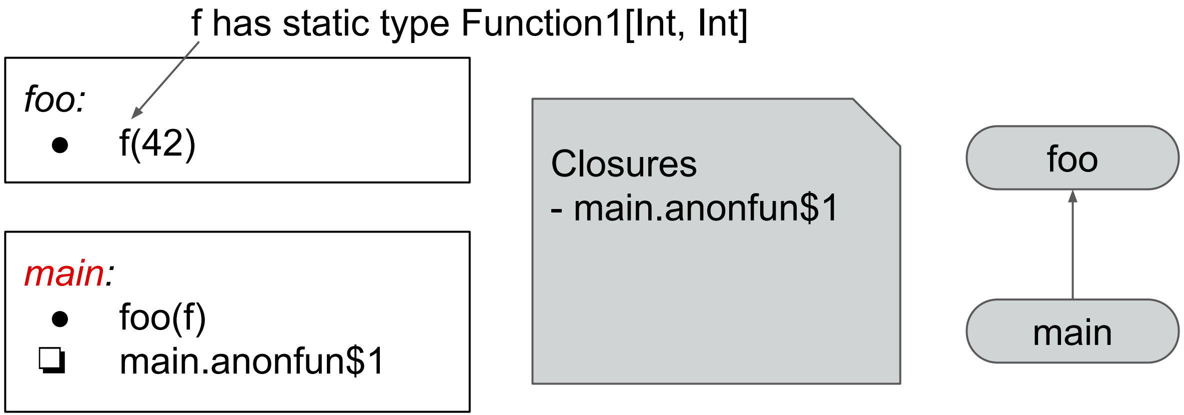
❏ main.anonfun$1

Closures
- main.anonfun$1



Closures

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



f has static type Function1[Int, Int]

foo:
- f(42)

main:
- foo(f)

❏ main.anonfun$1

Closures
- main.anonfun$1

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:

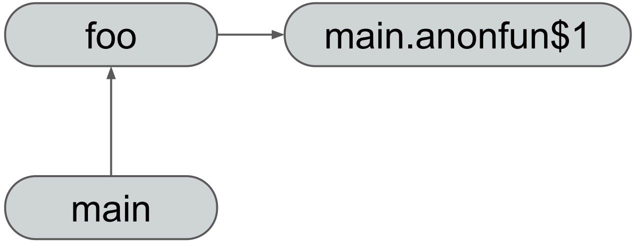
- f(42)

main:

- foo(f)

☐ main.anonfun$1

Closures
- main.anonfun$1



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!