PLRG

Resolving Issues on Dotty Compiler

Jaemin Hong

August 31st, 2018

What is Dotty?

- A next generation compiler for Scala
- Scala 3!
- v0.9.0

- `dotty.epfl.ch`
- `$ brew install lampepfl/brew/dotty`

What's New?

- Intersection types

```
trait A { def foo(): Unit }
trait B { def bar(): Unit }
def f(ab: A & B) = {
  ab.foo()
  ab.bar()
}
```

What's New?

- Union types

```
case class A(foo: String)
case class B(bar: Int)
def f(ab: A | B) = ab match {
  case A(foo) => foo
  case B(bar) => bar.toString
}
```

What's New?

- Literal singleton types

```
val foo: 1 = 1
val bar: "hello world" = "hello world"
def f(x: "hello") = x + " world"
f(“hello")  // "hello world"
f("hi")  // found: String("hi"), required: String("hello")
```

What's New?

- Type lambdas

```
type T[A] = (A, A)

(1 -> 1): T[Int]


trait Foo[X[_]]

class Bar extends Foo[T]
```

What's New?

- Type lambdas

```
type S = [A] => (A, A)
(1 -> 1): S[Int]

trait Foo[X[_]]
class Bar extends Foo[S]
class Bar2 extends Foo[[A] => (A, A)]
```

What's New?

- Implicit function types

```
implicit val ev: Int = 1

def foo(implicit x: Int) = x

bar  // 1
```

What's New?

- Implicit function types

```
def foo(implicit x: Int) = x
def bar(f: implicit Int => Int) = {
  implicit val ev: Int = 2;  f
}
bar(foo)  // 2
bar(implicit (x: Int) => x * x)  // 4
```

What's New?

- Trait parameters

```
trait Foo(x: Int)

class Bar(x: Int) extends Foo(x)
```

New Compiler

- Builds on DOT in the internal data structure.
- Reduce the code size.
    - dotty: 45 KLoc
    - nsc: 75 KLoc
- Twice the speed of nsc.

Dotty Internal

Parse
Type check

Transform AST

Tail recursion
Type erasure
Lambda lift
...

Generate byte code

PLRG

Fix #4466: same ctx for tryInsertApplyOrImplicit ✓ — 7
#4853 by Medowhill was closed 29 days ago

Fix #4653: check method return type is value type ✓ — 8
#4844 by Medowhill was merged on 30 Jul • Approved

Fix #4837: use dealiased type for parent symbol ✓ — 2
#4838 by Medowhill was merged on 31 Jul • Approved

Fix #4725: Expand implicit method to implicit func ✓ — 2
#4835 by Medowhill was merged on 26 Jul • Approved

Fix #4582: use untpd.New in maybeCall ✓ — 10
#4827 by Medowhill was merged on 24 Jul • Approved

Fix #4364: Try SAM type when no candidates found ✓ — 25
#4821 opened on 21 Jul by Medowhill • Changes requested

Fix #4451: Set flag in the correct order ✓ — 5
#4817 by Medowhill was merged on 21 Jul • Approved

Fix #4404: In LambdaLift, modify flags properly ✓ — 21
#4804 by Medowhill was merged on 20 Jul • Approved

Fix #4557: Handle untpd.New with HKTypeLambda ✓ — 16
#4798 by Medowhill was merged on 26 Jul

Issue #4557

```
class Parser[T, Elem]


object Test {
  type P[T] = Parser[T, Char]
  class CustomParser extends P[Unit]
  // Test.P is not a class type
}
```

Issue #4557

```
class Foo[X, Y]

type T[X] = Foo[X, Unit]

class Bar extends T[Unit]  // T is not a class type
```

Issue #4557

```
class Foo[X, Y]

type T[X] = Foo[X, Unit]

class Bar extends Foo[Unit, Unit]

Foo[Unit, Unit]
→ (new Foo).<init>[Unit, Unit]()
```

Issue #4557

```
def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  val (tycon, targs) = tpt match {
    case AppliedTypeTree(tycon, targs) => (tycon, targs)
    case TypedSplice(AppliedTypeTree(tycon, targs)) =>
      (TypedSplice(tycon), targs map (TypedSplice(_)))
    case TypedSplice(tpt1: tpd.Tree) =>
      val tycon = tpt1.tpe.typeConstructor
      val argTypes = tpt1.tpe.argTypesLo
      def wrap(tpe: Type) = TypeTree(tpe) withPos tpt.pos
      (wrap(tycon), argTypes map wrap)
    case _ =>
      (tpt, Nil)
  }
  var prefix: Tree = Select(New(tycon), nme.CONSTRUCTOR)
  if (targs.nonEmpty) prefix = TypeApply(prefix, targs)
  ensureApplied((prefix /: argss)(Apply(_, _)))
}
```

Issue #4557

```
tpt   = Foo[Unit, Unit]
argss = List()

def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  …
}
```

Issue #4557

```
tpt   =              AppliedTypeTree(Foo, List(Unit, Unit))
argss = List()

def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  …
}
```

Issue #4557

```
tpt   = TypedSplice(AppliedTypeTree(Foo, List(Unit, Unit)))
argss = List()

def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  …
}
```

Issue #4557

```
tpt   = TypedSplice(AppliedTypeTree(Foo, List(Unit, Unit)))
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    (TypedSplice(tycon), targs map (TypedSplice(_)))
}
```

Issue #4557

```
tpt   = TypedSplice(AppliedTypeTree(Foo, List(Unit, Unit)))
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    (TypedSplice(tycon), targs map (TypedSplice(_)))
}

tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

Issue #4557

```
argss = List()
tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))

var prefix: Tree = Select(New(tycon), nme.CONSTRUCTOR)
```

Issue #4557

```
argss = List()
tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))

var prefix: Tree = Select(New(tycon), nme.CONSTRUCTOR)

prefix = Select(New(TypedSplice(Foo)), <init>)
```

Issue #4557

```
argss = List()
tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))

var prefix: Tree = Select(New(tycon), nme.CONSTRUCTOR)

prefix = Select(New(TypedSplice(Foo)), <init>)
       ≡ (new Foo).<init>
```

Issue #4557

```
argss  = List()
prefix = Select(New(TypedSplice(Foo)), <init>)

if (targs.nonEmpty) prefix = TypeApply(prefix, targs)
```

PLRG

Issue #4557

```
argss  = List()
prefix = Select(New(TypedSplice(Foo)), <init>)

if (targs.nonEmpty) prefix = TypeApply(prefix, targs)

prefix = TypeApply(
           Select(New(TypedSplice(Foo)), <init>),
           List(TypedSplice(Unit), TypedSplice(Unit))
         )
```

Issue #4557

```
argss  = List()
prefix = Select(New(TypedSplice(Foo)), <init>)

if (targs.nonEmpty) prefix = TypeApply(prefix, targs)

prefix = TypeApply(
           Select(New(TypedSplice(Foo)), <init>),
           List(TypedSplice(Unit), TypedSplice(Unit))
         )
       ≡ (new Foo).<init>[Unit, Unit]
```

Issue #4557

```
argss  = List()
prefix = TypeApply(Select(New(TypedSplice(Foo)), <init>),
           List(TypedSplice(Unit), TypedSplice(Unit)))

ensureApplied((prefix /: argss)(Apply(_, _)))
```

Issue #4557

```
argss  = List()
prefix = TypeApply(Select(New(TypedSplice(Foo)), <init>),
           List(TypedSplice(Unit), TypedSplice(Unit)))

ensureApplied((prefix /: argss)(Apply(_, _)))

Apply(
  TypeApply(Select(New(TypedSplice(Foo)), <init>),
    List(TypedSplice(Unit), TypedSplice(Unit))),
  List()
)
```

Issue #4557

```
argss  = List()
prefix = TypeApply(Select(New(TypedSplice(Foo)), <init>),
           List(TypedSplice(Unit), TypedSplice(Unit)))

ensureApplied((prefix /: argss)(Apply(_, _)))

Apply(
  TypeApply(Select(New(TypedSplice(Foo)), <init>),
    List(TypedSplice(Unit), TypedSplice(Unit))),
  List()
) ≡ (new Foo).<init>[Unit, Unit]()
```

Issue #4557

```
tpt   = TypedSplice(AppliedTypeTree(Foo, List(Unit, Unit)))
argss = List()

def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  …
}

Apply(
  TypeApply(Select(New(TypedSplice(Foo)), <init>),
    List(TypedSplice(Unit), TypedSplice(Unit))),
  List()
)
```

Issue #4557

```
tpt   = Foo[Unit, Unit]
argss = List()

def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  …
}

(new Foo).<init>[Unit, Unit]()
```

Issue #4557

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  …
}
```

Issue #4557

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

def New(tpt: Tree, argss: List[List[Tree]])(implicit ctx: Context): Tree = {
  …
}

(new T).<init>[Unit]()  where T[X] = Foo[Unit, X]
// T is not a class type
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
```

```
tycon = TypedSplice(T)
targs = List(TypedSplice(Unit))
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
```

```
tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe



tycon             = TypedSplice(T)
tycon.tpe         = T
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe.dealias
```

```
tycon              = TypedSplice(T)
tycon.tpe          = T
tycon.tpe.dealias = [X] => Foo[Unit, X]
                   ≡ HKTypeLambda(List(X), AppliedType(Foo, List(Unit, X)))
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe.dealias match {
      case tp1: HKTypeLambda =>


tp1               = [X] => Foo[Unit, X]
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe.dealias match {
      case tp1: HKTypeLambda =>     tp1.appliedTo(targs.map(_.tpe))



tp1               = [X] => Foo[Unit, X]
targs.map(_.tpe) = List(Unit)
tp1.appliedTo …  = Foo[Unit, Unit]
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe.dealias match {
      case tp1: HKTypeLambda =>     tp1.appliedTo(targs.map(_.tpe))

  case TypedSplice(tpt1: tpd.Tree) =>
    val tycon = tpt1.tpe.typeConstructor
    val argTypes = tpt1.tpe.argTypesLo
    def wrap(tpe: Type) = TypeTree(tpe) withPos tpt.pos
    (wrap(tycon), argTypes map wrap)
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe.dealias match {
      case tp1: HKTypeLambda => aux(tp1.appliedTo(targs.map(_.tpe)))

  case TypedSplice(tpt1: tpd.Tree) => aux(tpt1.tpe)
def aux(tp: Type) = { val tycon = tp.typeConstructor
                      val argTypes = tp.argTypesLo
                      def wrap(tpe: Type) = TypeTree(tpe) withPos tpt.pos
                      (wrap(tycon), argTypes map wrap) }
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe.dealias match {
      case tp1: HKTypeLambda => aux(tp1.appliedTo(targs.map(_.tpe)))
      case _ => (TypedSplice(tycon), targs map (TypedSplice(_))) }
  case TypedSplice(tpt1: tpd.Tree) => aux(tpt1.tpe)
}
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    tycon.tpe.dealias match {
      case tp1: HKTypeLambda => aux(tp1.appliedTo(targs.map(_.tpe)))
      case _ => (TypedSplice(tycon), targs map (TypedSplice(_))) }
  case TypedSplice(tpt1: tpd.Tree) => aux(tpt1.tpe)
}

tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

PR #4789

```
class Foo[X, Y]

type T[X] = Foo[X, Unit]

class Bar extends T[Unit]
```

PR #4789

```
class Foo[X, Y]


type T[X] = Foo[X, Unit]
type S[X] = T[X]
class Bar extends S[Unit]  // T is not a class type
```

PR #4789

```
tpt   = S[Unit]  where S[X] = T[X], T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>

    tycon.tpe.dealias match {
      case tp1: HKTypeLambda => aux(tp1.appliedTo(targs.map(_.tpe)))
      case _ => (TypedSplice(tycon), targs map (TypedSplice(_))) }
}

tycon = TypedSplice(T)
targs = List(TypedSplice(Unit))
```

PR #4789

```
tpt   = S[Unit]  where S[X] = T[X], T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    @tailrec def normalize(tp: HKTypeLambda) = …
    tycon.tpe.dealias match {
      case tp1: HKTypeLambda => aux(normalize(tp1))
      case _ => (TypedSplice(tycon), targs map (TypedSplice(_))) }
}

tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(tpt1: AppliedTypeTree) =>
        tpt1.tpe.dealias
```

```
tpt1.tpe.dealias = Foo[Unit, Unit]
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(tpt1: AppliedTypeTree) =>
    aux(tpt1.tpe.dealias)
```

```
tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(tpt1: AppliedTypeTree) =>
    aux(tpt1.tpe.dealias)
  case TypedSplice(tpt1: tpd.Tree) =>
    aux(tpt1.tpe)



tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(tpt1: AppliedTypeTree) =>
    aux(tpt1.tpe.dealias)
  case TypedSplice(tpt1: tpd.Tree) =>
    aux(tpt1.tpe.dealias)



tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {


  case TypedSplice(tpt1: tpd.Tree) =>
    aux(tpt1.tpe.dealias)



tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

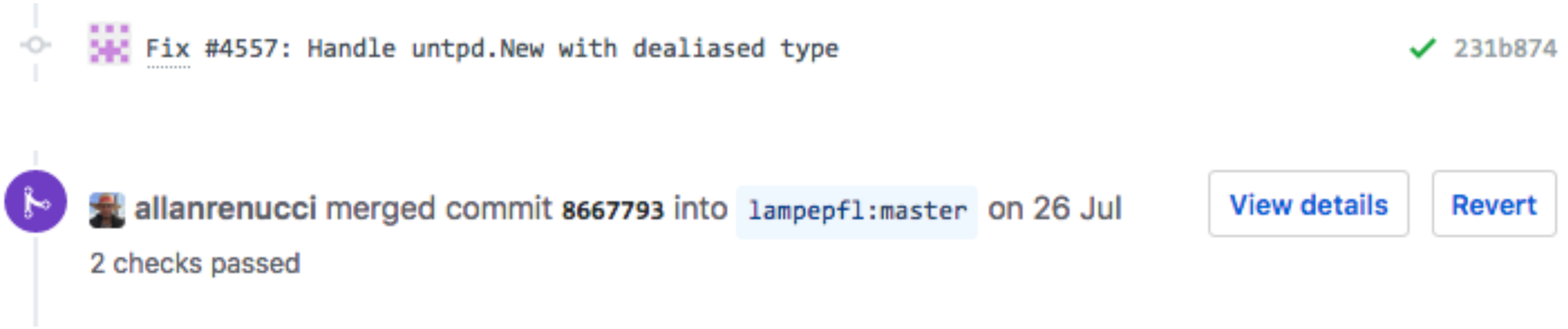
PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(tpt1: tpd.Tree) =>
    val tp = tpt1.tpe.dealias
    val tycon = tp.typeConstructor
    val argTypes = tp.argTypesLo
    def wrap(tpe: Type) = TypeTree(tpe) withPos tpt.pos
    (wrap(tycon), argTypes map wrap)

tycon = TypedSplice(Foo)
targs = List(TypedSplice(Unit), TypedSplice(Unit))
```

PR #4789

```
val (tycon, targs) = tpt match {
  case AppliedTypeTree(tycon, targs) =>
    (tycon, targs)
  case TypedSplice(AppliedTypeTree(tycon, targs)) =>
    (TypedSplice(tycon), targs map (TypedSplice(_)))
  case TypedSplice(tpt1: tpd.Tree) =>
    val tycon = tpt1.tpe.typeConstructor
    val argTypes = tpt1.tpe.argTypesLo
    val tp = tpt1.tpe.dealias
    val tycon = tp.typeConstructor
    val argTypes = tp.argTypesLo
    def wrap(tpe: Type) = TypeTree(tpe) withPos tpt.pos
    (wrap(tycon), argTypes map wrap)
```

PR #4789


Fix #4557: Handle untpd.New with dealiased type
231b874
allanrenucci merged commit 8667793 into lampepfl:master on 26 Jul
View details
Revert
2 checks passed

PR #4789


Fix #4557: Handle untpd.New with dealiased type
231b874
allanrenucci merged commit 8667793 into lampepfl:master on 26 Jul
View details
Revert
2 checks passed
Merge pull request #4798 from Medowhill/issue4557
allanrenucci committed on 26 Jul
Verified
8667793

PR #4789

```
val (tycon, targs) = tpt match {
  case AppliedTypeTree(tycon, targs) =>
    (tycon, targs)
  case TypedSplice(tpt1: tpd.Tree) =>
    val tp = tpt1.tpe.dealias
    val tycon = tp.typeConstructor
    val argTypes = tp.argTypesLo
    def wrap(tpe: Type) = TypeTree(tpe) withPos tpt.pos
    (wrap(tycon), argTypes map wrap)
    val argTypes = tpt1.tpe.dealias.argTypesLo
    def wrap(tpe: Type) = TypeTree(tpe).withPos(tpt.pos)
    (tpt, argTypes.map(wrap))
```

We need to keep the original type

- to allow IDE navigation
- to make PrepareTransparent work correctly

PR #4789

```
tpt   = T[Unit]  where T[X] = Foo[Unit, X]
argss = List()

val (tycon, targs) = tpt match {
  case TypedSplice(tpt1: tpd.Tree) =>
    val argTypes = tpt1.tpe.dealias.argTypesLo
    def wrap(tpe: Type) = TypeTree(tpe).withPos(tpt.pos)
    (tpt, argTypes.map(wrap))

tycon = TypedSplice(AppliedTypeTree(T, List(Unit)))
targs = List(TypedSplice(Unit))

(new T[Unit]).<init>[Unit]()
```

Issue #4725

```
import language.higherKinds

object Main {
  trait Eq[A]
  case class Cofree[S[_], A](head: A, tail: Cofree[S, A])
  implicit val intEq: Eq[Int] = null
  implicit def forList[A : Eq]: Eq[List[A]] = ???
  implicit def eqCofree[S[_], A : Eq](
    implicit S: implicit Eq[A] => Eq[S[A]]): Eq[Cofree[S, A]] = ???
  val a = eqCofree[List, Int]
  // no implicit argument of type implicit Main.Eq[Int] => Main.Eq[List[Int]]
  // was found for parameter S of method eqCofree in object Main.
}
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: implicit T => S): Unit = ()
implicit def bar(implicit ev: T): S = ???

foo
// no implicit argument of type implicit T => S was found for
// parameter ev of method foo
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: T => S): Unit = ()
implicit def bar(ev: T): S = ???

foo
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: T => S): Unit = ()
implicit def bar(ev: T): S = ???

foo(bar)
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: T => S): Unit = ()
implicit def bar(ev: T): S = ???

foo(
  (ev: T) => bar(ev)
)
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: T => S): Unit = ()
implicit def bar(ev: T): S = ???

foo({
  def $anonfun(ev: T): S = bar(ev)
  closure($anonfun)
})
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: implicit T => S): Unit = ()
implicit def bar(implicit ev: T): S = ???

foo(bar)
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: implicit T => S): Unit = ()
implicit def bar(implicit ev: T): S = ???

foo(
  implicit (evidence$1: T) => bar(evidence$1)
)
```

Issue #4725

```
trait T
trait S

def foo(implicit ev: implicit T => S): Unit = ()
implicit def bar(implicit ev: T): S = ???

foo({
  def $anonfun(implicit evidence$1: T): S = bar(evidence$1)
  closure($anonfun)
})
```

Issue #4725

Type `foo`:

`foo` is an implicit method.

function type is not expected.

```
formal = implicit T => S
       = ImplicitFunction1[T, S]
       ≡ AppliedType(ImplicitFunction1, List(T, S))

def inferImplicitArg(formal: Type, pos: Position)
  (implicit ctx: Context): Tree
```

Issue #4725

```
formal   = implicit T => S
eligible = List(Candidate(bar))

def searchImplicits(eligible: List[Candidate], contextual: Boolean):
  SearchResult
```

Issue #4725

```
formal   = implicit T => S
cand     =       Candidate(bar)

def typedImplicit(cand: Candidate, contextual: Boolean)
  (implicit ctx: Context): SearchResult
```

Issue #4725

```
pt       = implicit T => S
tree     =                 bar

def adapt1(tree: Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree
```

Issue #4725

```
pt       = implicit T => S
tree     =                 bar

def adapt1(tree: Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree

wtp      = tree.tpe.widen
         = (ev: implicit T): S
         ≡ MethodType(List(ev), List(T), S)

  typed(etaExpand(tree, wtp, arity), pt)
```

Issue #4725

```
pt       = implicit T => S
initTree = ev => bar(ev)

def typedUnadapted(initTree: untpd.Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree
```

Issue #4725

```
pt       = implicit T => S
initTree = ev => bar(ev)

def typedUnadapted(initTree: untpd.Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree

xtree    = ev => bar(ev)

  val ifpt = defn.asImplicitFunctionType(pt)
  val result = if (ifpt.exists && !untpd.isImplicitClosure(xtree) && …)
                 makeImplicitFunction(xtree, ifpt) else …
```

Issue #4725

```
pt       = implicit T => S
initTree = ev => bar(ev)

def typedUnadapted(initTree: untpd.Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree

xtree    = ev => bar(ev)

  val ifpt = defn.asImplicitFunctionType(pt)
  val result = if (ifpt.exists && !untpd.isImplicitClosure(xtree) && …)
                 makeImplicitFunction(xtree, ifpt) else …

           implicit (evidence$1: T) => (ev => bar(ev)): S
```

PR #4835

```
var paramFlag = Synthetic | Param
if (mt.isImplicitMethod) paramFlag |= Implicit
val params = (mt.paramNames, paramTypes).zipped.map((name, tpe) =>
  ValDef(name, tpe, EmptyTree).withFlags(Synthetic | Param)
    .withPos(tree.pos.startPos))
  ValDef(name, tpe, EmptyTree).withFlags(paramFlag)
    .withPos(tree.pos.startPos))
```

…

```
val fn = untpd.Function(params, body)
val fn =
  if (mt.isImplicitMethod)
    new untpd.FunctionWithMods(params, body, Modifiers(Implicit))
  else untpd.Function(params, body)
```

PR #4835

```
pt       = implicit T => S
tree     =                 bar

def adapt1(tree: Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree

wtp      = tree.tpe.widen
         = (ev: implicit T): S
         ≡ MethodType(List(ev), List(T), S)

  typed(etaExpand(tree, wtp, arity), pt)
```

PR #4835

```
pt       = implicit T => S
initTree = implicit ev => bar(ev)

def typedUnadapted(initTree: untpd.Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree
```

PR #4835

```
pt       = implicit T => S
initTree = implicit ev => bar(ev)

def typedUnadapted(initTree: untpd.Tree, pt: Type, locked: TypeVars)
  (implicit ctx: Context): Tree

xtree    = implicit ev => bar(ev)

  val ifpt = defn.asImplicitFunctionType(pt)
  val result = if (ifpt.exists && !untpd.isImplicitClosure(xtree) && …)
                 makeImplicitFunction(xtree, ifpt) else …

           implicit (ev: T) => bar(ev): S
```

Issue #4466

```
object Foo {
  case class Bar(map: Map[String, String])
  object Bar {
    def apply(str: String): Bar = ???
  }
  Bar(Map("A" -> “B”))  // object Bar in object Foo does not take parameters
}
```

Issue #4466

```
class Foo {
  def apply(bar: Bar[Int]): Unit = ()
  def apply(i: Int): Unit = ()
}

class Bar[X](x: X)

val foo = new Foo
foo(new Bar(0))  // value foo does not take parameters
```

Issue #4466

Type `foo(new Bar(0))`

Type `foo` against `FunProto(List(new Bar(0)), _)`

`foo` is not a method.

→ Should insert `apply`.

Issue #4466

Type `foo(new Bar(0))`

  Type `foo` against `FunProto(List(new Bar(0)), _)`

    `foo` is not a method.

    → Should insert `apply`.

```
tree = foo
pt   = FunProto(List(new Bar(0)), _)
```

```
def tryInsertApplyOrImplicit(tree: Tree, pt: ProtoType, locked: TypeVars)
  (fallBack: => Tree)(implicit ctx: Context): Tree
```

Issue #4466

```
tree = foo
pt   = FunProto(List(new Bar(0)), _)

def tryInsertApplyOrImplicit(tree: Tree, pt: ProtoType, locked: TypeVars)
  (fallBack: => Tree)(implicit ctx: Context): Tree

  tryEither(tryApply(_))((_, _) => tryImplicit)
```

Issue #4466

```
tree = foo
pt   = FunProto(List(new Bar(0)), _)

def tryInsertApplyOrImplicit(tree: Tree, pt: ProtoType, locked: TypeVars)
  (fallBack: => Tree)(implicit ctx: Context): Tree

op   = tryApply

def tryEither[T](op: Context => T)(fallBack: (T, TyperState) => T)
  (implicit ctx: Context) = {
  val nestedCtx = ctx.fresh.setNewTyperState()
  val result = op(nestedCtx)
```

Issue #4466

```
tree = foo
pt   = FunProto(List(new Bar(0)), _)

def tryApply(implicit ctx: Context)

  val sel =
    typedSelect(untpd.Select(untpd.TypedSplice(tree), nme.apply), pt)

sel  = foo.apply + overloaded
```

Issue #4466

```
tree = foo
pt   = FunProto(List(new Bar(0)), _)

def tryApply(implicit ctx: Context)

  val sel =
    typedSelect(untpd.Select(untpd.TypedSplice(tree), nme.apply), pt)

sel  = foo.apply + overloaded

  adapt(simplify(sel, pt, locked), pt, locked)
```

Issue #4466

```
alts = List(foo.apply, foo.apply)
pt   = FunProto(List(new Bar(0)), _)

def resolveOverloaded(alts: List[TermRef], pt: Type, targs: List[Type])
  (implicit ctx: Context): List[TermRef]
```

Issue #4466

```
alts = List(foo.apply, foo.apply)
pt   = FunProto(List(new Bar(0)), _)

def resolveOverloaded(alts: List[TermRef], pt: Type, targs: List[Type])
  (implicit ctx: Context): List[TermRef]

alts2         = List(foo.apply, foo.apply)
pt.typedArgs = List(new Bar[X](0): Bar[X])
resultType   = _

  narrowByTrees(alts2, pt.typedArgs, resultType)
```

Issue #4466

```
class Foo {
  def apply(bar: Bar[Int]): Unit = ()
  def apply(i: Int): Unit = ()
}

Bar[X] <: Int      ?
Bar[X] <: Bar[Int] ?
```

Issue #4466

```
Bar[X] <: Bar[Int] ?
```

“New” TyperState

FunProto’s TyperState

TypeVar(X)

Literal(0) <: X

Issue #4466

```
Bar[X] <: Bar[Int] ?
```

“New” TyperState

Int <: X & X <: Int

FunProto’s TyperState

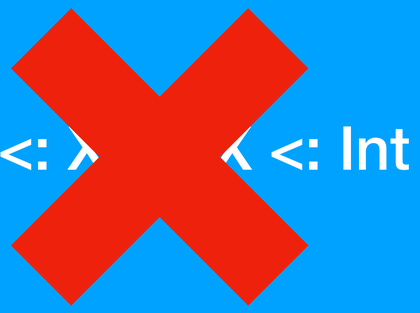
TypeVar(X)

Literal(0) <: X

Issue #4466

```
Bar[X] <: Bar[Int] ?
```

“New” TyperState



FunProto’s TyperState

TypeVar(X)

Literal(0) <: X

PR #4871

```
override def withContext(newCtx: Context) =
  if (newCtx `eq` ctx) this
  else new FunProto(args, resType)(typer, state)(newCtx)
}
```

…

```
val sel = typedSelect(untpd.Select(untpd.TypedSplice(tree), nme.apply), pt)
val pt1 = pt.withContext(ctx)
val sel = typedSelect(untpd.Select(untpd.TypedSplice(tree), nme.apply), pt1)
```

…

```
adapt(simplify(sel, pt, locked), pt, locked)
adapt(simplify(sel, pt1, locked), pt1, locked)
```