LOGICALLY QUALIFIED TYPES FOR SCALA 3

Matt Bovel @LAMP/LARA, EPFL

October 13, 2025

INTRODUCTION

I am Matt Bovel (@mbovel).

A PhD student at EPFL in Switzerland, between two labs:

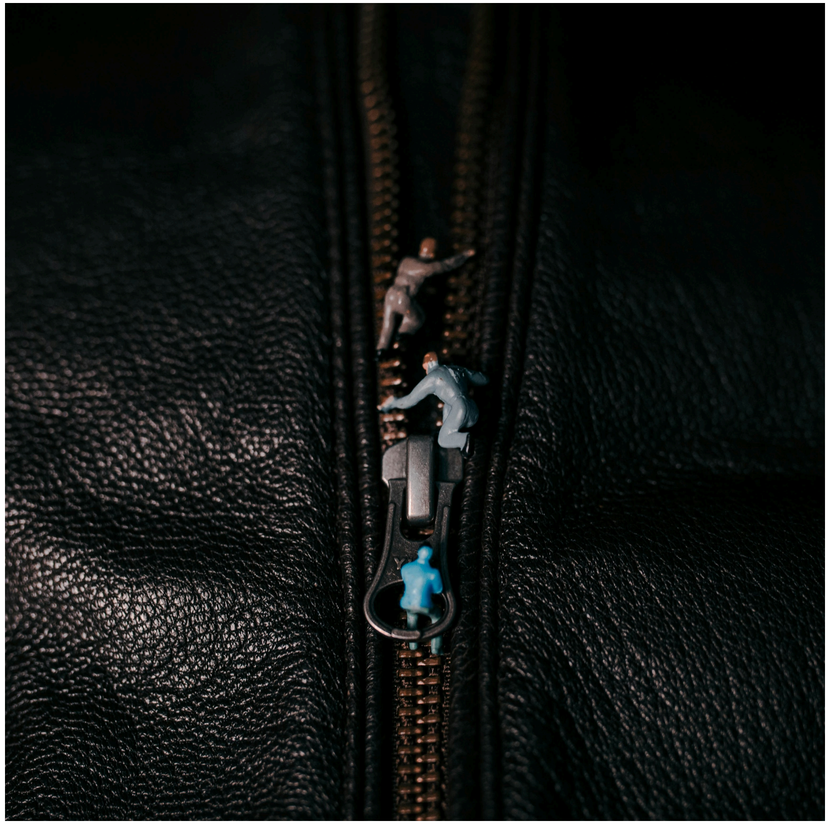
- LAMP: led by Martin Odersky, making the Scala compiler,
- LARA: led by Viktor Kunčak, making the Stainless verifier.

Work done in collaboration with Quentin Bernet and Valentin Schneeberger.

MOTIVATING EXAMPLE: SAFE LIST ZIP

Consider the standard `zip` function:

```
def zip[A, B](
  as: List[A],
  bs: List[B]
): List[(A, B)] =
  ...
```



Black leather zip up jacket, by Todd Pham

SPECIFY USING ASSERTIONS 😕

We can use assertions:

```
def zip[A, B](
  as: List[A],
  bs: List[B]
) : List[(A, B)] = {
  require(as.size == bs.size)
  ...
}.ensure(_.size == as.size)
```

Limitations:

- *Runtime overhead*: checked at runtime, not compile time,
- *No static guarantees*: only checked for specific inputs,
- *Not part of the API*: not visible in function type,
- *Hard to compose*: cannot be passed as type argument.

SPECIFY USING DEPENDENT TYPES 😕

Can we use path-dependent types?

```
def zip[A, B](
  as: List[A],
  bs: List[B] {
    val size: as.size.type
  }
): List[(A, B)] {
  val size: as.size.type
} = ...
```

Limitations:

- *Limited reasoning*: only fields, literals and constant folding,
- *Not inferred*: need manual type annotations, or not typable at all,
- *Different languages*: term-level vs type-level.

SPECIFY USING LOGICALLY QUALIFIED TYPES ! 🤩

Introducing logically qualified types:

```
def zip[A, B](
  as: List[A],
  bs: List[B] with bs.size == as.size
): {l: List[(A, B)] with l.size == as.size} = ...
```

The return type means
“any value `l` of type `List[(A, B)]` such that `l.size == as.size`”.

IN OTHER LANGUAGES

- “Refinement types for ML” (Freeman & Pfenning, 1991)
- “Liquid Types” (Rondon, Kawaguchi & Jhala, 2008)
- “Refinement Types for Haskell” (Vazou, Seidel, Jhala, Vytiniotis, Peyton-Jones, 2014)
- Liquid Haskell
- Boolean refinement types in F*
- Subset types in Dafny
- Subtypes in Lean

In Scala:

- “SMT-based checking of predicate-qualified types for Scala”, (Schmid & Kunčak, 2016)
- Refined library, Frank Thomas
- Iron library, Raphaël Fromentin

MAIN DIFFERENCE WITH LIQUID HASKELL

Liquid Haskell is a plugin that runs after type checking.

```haskell
module Demo.Hello where

{-@ test2 :: v1:Int -> {it: Int | it == v1} @-}
test2 :: Int -> Int
test2 v1 =
  let v2 = v1 in
  {-@ v2 :: {it: Int | true } @-}
  let v3 = v2 in
  {-@ v3 :: {it: Int | it == v1} @-}
  v3

```

Screenshot from the Liquid Haskell Demo

In contrast, we integrate qualified types directly into the Scala type system and compiler.

SYNTAX

```scala
type NonEmptyList[A] = { l: List[A] with l.nonEmpty }
```

- `l` : binder
- `List[A]` : parent type
- `l.nonEmpty` : qualifier (predicate)

Not to be confused with Scala's existing structural refinement types:

```scala
case class Box(value: Any)
type IntBox = Box { val value: Int }
```

SHORTHAND SYNTAX

When a binder already exists, such as in:

```
def zip[A, B](as: List[A], bs: {bs: List[B] with bs.size == as.size})
```

We can omit it:

```
def zip[A, B](as: List[A], bs: List[B] with bs.size == as.size)
```

The second version is desugared to the first.

MORE LIST API EXAMPLES 🥳

```
def zip[A, B](as: List[A], bs: List[B] with bs.size == as.size):
  {l: List[(A, B)] with l.size == as.size}
```

```
def concat[T](as: List[T], bs: List[T]):
  {rs: List[T] with rs.size == as.size + bs.size}
```

```
val xs: List[Int] = ...
val ys: List[Int] = ...
zip(concat(xs, ys), concat(ys, xs))
zip(concat(xs, ys), concat(xs, xs)) // error
```

WHAT ARE VALID PREDICATES?

```scala
var x = 3
val y: Int with y == 3 = x // ⛔ x is mutable
```

```scala
class Box(val value: Int)
val b: Box with b == Box(3) = Box(3) // ⛔ Box has equality by reference
```

The predicate language is restricted to a fragment of Scala consisting of constants, stable identifiers, field selections over `val` fields, pure term applications, type applications, and constructors of case classes without initializers.

Purity of functions is currently not enforced. Should it be?

HOW TO INTRODUCE QUALIFIED TYPES?

For backward compatibility and performance reasons, qualified types are not inferred from terms by default. The wider type is inferred instead:

```scala
val x: Int = readInt()
val y /* : Int */ = x + 1
```

SELFIFICATION

However, when a qualified type is expected, the compiler attempts to *selfify* the typed expression: that is, to give `e: T` the qualified type `x: T with x == e`:

```
val x: Int = readInt()
val y: Int with (y == x + 1) = x + 1
```

```
def f(i: Int): Int = i * 2
val z: Int with (z == x + f(x)) = x + f(x)
```

RUNTIME CHECKS

When static checking fails, a qualified type can be checked at runtime using pattern matching:

```
val idRegex = "^[a-zA-Z_][a-zA-Z0-9_]*$"
type ID = {s: String with s.matches(idRegex)}
```

```
"a2e7-e89b" match
    case id: ID => // matched: `id` matches idRegex
    case id     => // didn't match
```

RUNTIME CHECKS: `.runtimeChecked`

You can also use `.runtimeChecked` (SIP-57) when the check must always pass:

```
val id: ID = "a2e7-e89b".runtimeChecked
```

Desugars to:

```
val id: ID =
  if ("a2e7-e89b".matches(idRegex)) "a2e7-e89b".asInstanceOf[ID]
  else throw new IllegalArgumentException()
```

Note: like with other types, you can also use `.asInstanceOf[ID]` directly to skip the check altogether.

RUNTIME CHECKS: `List.collect`

Scala type parameters are *erased* at runtime, so we cannot match on a `List[T]`.

However, we can use `.collect` to filter and convert a list:

```
type Pos = { v: Int with v >= 0 }

val xs = List(-1,2,-2,1)
xs.collect { case x: Pos => x } : List[Pos]
```

SUBTYPING

How does the compiler check `{x: T with p(x)} <: {y: S with q(y)}`?

1. Check `T <: S`
2. Check `p(x)` implies `q(x)` for all `x`

A solver is needed to check logical implication (2.).

We developed a lightweight custom solver that combines several techniques:

- constant folding,
- normalization,
- unfolding,
- and equality reasoning.

SUBTYPING: CONSTANT FOLDING

```
{v: Int with v == 1 + 1}      <: {v: Int with v == 2}
```

SUBTYPING: NORMALIZATION

Arithmetic expressions are normalized using standard algebraic properties, for example commutativity of addition:

```
{v: Int with v == x + 1}     <: {v: Int with v == 1 + x}
```

```
{v: Int with v == y + x}     <: {v: Int with v == x + y}
```

Or grouping operands with the same constant factor in sums of products:

```
{v: Int with v == x + 3 * y} <: {v: Int with v == 2 * y + (x + y)}
```

SUBTYPING: UNFOLDING

Remember: qualified types are not inferred from terms by default. However, the solver can unfold definitions of local `val` (only), even when they have an imprecise type:

```
val x: Int = ...
val y: Int = x + 1

{v: Int with v == y} =:= {v: Int with v == x + 1}
```

SUBTYPING: EQUALITY REASONING

Transitivity of equality:

```
{v: Int with v == a && a == b} <: {v: Int with v == b}
```

Congruence of equality:

```
{v: Int with a == b}              <: {v: Int with f(a) == f(b)}
```

This is implemented using an E-Graph-like data structure.

SUBTYPING WITH OTHER SCALA TYPES

Singleton qualified types are subtypes of literal types:

```scala
{v: Int with v == 3} <: 3
```

We plan to support subtyping with other Scala types in the future.

FUTURE WORK: SIP

Some work remains on UX (error messages, IDE support, documentation).

Then we’ll make a pre-SIP to get feedback from the community.

Then a full SIP to standardize qualified types in Scala! 🚀

FUTURE WORK: TERM-PARAMETERIZED TYPES

```
extension [T](list: List[T])
  def get(index: Int with index >= 0 && index < list.size): T = ...
```

To modularize the “range” concept, we could introduce term-parameterized types:

```
type Range(from: Int, to: Int) = {v: Int with v >= from && v < to}
extension [T](list: List[T])
  def get(index: Range(0, list.size)): T = ...
```

FUTURE WORK: FLOW-SENSITIVE TYPING

Works with pattern matching:

```
x match
  case x: Int with x > 0 =>
    x: {v: Int with v > 0}
```

Could also work with `if` conditions:

```
if x > 0 then
  x: {v: Int with v > 0}
```

FUTURE WORK: FLOW-SENSITIVE TYPING

Crucially, this would be required for “GADT-like” reasoning with qualified types:

```
enum MyList[+T]:
  case Cons(head: T, tail: MyList[T])
  case Nil

def myLength(xs: MyList[Int]): Int =
  xs match
    case MyList.Nil =>
      // Add assumption xs == MyList.Nil
      0
    case MyList.Cons(_, xs1) =>
      // Add assumption xs == MyList.Cons(?, xs1)
      1 + myLength(xs1)
```

FUTURE WORK: INTEGRATION WITH SMT SOLVERS

Our solver is lightweight 👍 but incomplete 👎.

In particular, it cannot handle ordering relations yet, for example it cannot prove:

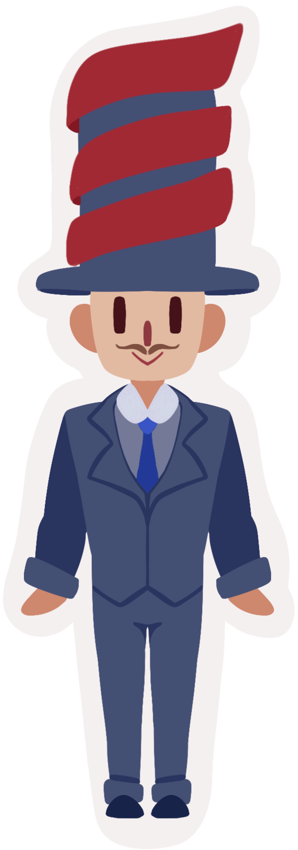
```
{v: Int with v > 2} <: {v: Int with v > 0}
```

For this and for more complex predicates, we could integrate with an external SMT solver like Z3, CVC5, or Princess *for casting only*, so that we don't pay the potential performance cost everywhere.

CONCLUSION

- Syntax: `{x: T with p(x)}`,
- Selfification: `e: T` becomes `x: T with x == e` when needed,
- Runtime checks: pattern matching and `.runtimeChecked`,
- Subtyping: custom lightweight solver,
- Future work: SIP, term-parameterized types, flow-sensitive typing, SMT integration.

- Two-page summary
- Prototype (dotty#21586)



Un type qualifié, by Marina Granados Castro

BACKUP/OUTDATED SLIDES

BONUS: WORKS WITH IMPLICIT RESOLUTION

```
type Pos = { v: Int with v >= 0 }
type Neg = { v: Int with v < 0 }

trait Show[-A]:
  def apply(a: A): String
given show1: Show[Pos] with
  def apply(a: Pos): String = "I am a positive integer!"
given show2: Show[Neg] with
  def apply(a: Neg): String = "I am a negative integer!"
def show[A](a: A)(using s: Show[A]): String = s.apply(a)

def f(x: Int with x == 42, y: Int with y == -42): Unit =
  println(show(x)) // I am a positive integer!
  println(show(y)) // I am a negative integer!
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL

```scala
def checkSame(dimA: Int, dimB: dimA.type): Unit = ()
checkSame(3, 3) // ok
checkSame(3, 4) // error
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL 🥲

```scala
def checkSame(dimA: Int, dimB: dimA.type): Unit = ()
val x = 3
val y = 3
checkSame(x, y) // error
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL 😄

```scala
def checkSame(dimA: Int, dimB: dimA.type): Unit = ()
val x: 3 = 3
val y: 3 = 3
checkSame(x, y) // ok
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL 😢

```scala
def checkSame(dimA: Int, dimB: dimA.type): Unit = ()
def readInt(): Int = ...
val x: Int = readInt()
val y = x
val z = y
checkSame(y, z) // error
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL 😁

```scala
def checkSame(dimA: Int, dimB: dimA.type): Unit = ()
val x: Int = readInt()
val y: x.type = x
val z: x.type = x
checkSame(y, z) // okay
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL 😓

```scala
def checkSame(dimA: Int, dimB: dimA.type): Unit = ()
val x: Int = readInt()
val y: Int = readInt()
val z = x + y
val a = y + x
checkSame(z, a) // error
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL 😲

```scala
def checkSame(dimA: Int, dimB: dimA.type): Unit = ()
import scala.compiletime.ops.int.+
val x: 3 = 3
val y: 5 = 5
val z: x.type + y.type = x + y
val a: y.type + x.type = y + x
checkSame(z, a) // error
```

CHECKING INTEGER EQUALITY AT THE TYPE LEVEL 😭

```scala
import scala.compiletime.ops.int.+
val x: Int = readInt()
val y: Int = readInt()
val z: x.type + y.type = x + y // error
val a: y.type + x.type = y + x // error
checkSame(z, a) // error
```