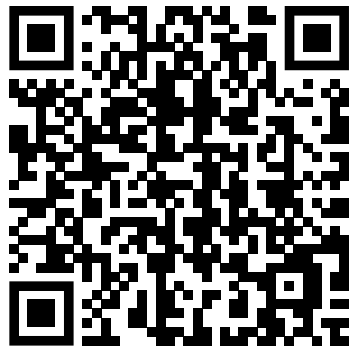


FIRST-CLASS LOGICAL REFINEMENT TYPES FOR SCALA

Matt Bovel, EPFL
matt.bovel.net



2026
ScalaDays
BERLIN

WHOAMI

PhD student at EPFL, co-advised by:

- Martin Odersky (LAMP/Scala)
- Viktor Kunčák (LARA/Stainless)

WHAT IS THIS ABOUT?

A *prototype* implementation of refinement types for Scala.

A *refinement type* is a type *refined* with a logical predicate.

For example:

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

First-class: directly integrated into the compiler and Scala type system.

Logical: not to be confused with existing **strutural refined types**.

SYNTAX

Long form, with an explicit binder:

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

Short form, reusing a name already in scope:

```
val x: (Int with x % 2 == 0) = 42  
// desugars to:  
val x: {v: Int with v % 2 == 0} = 42
```

Surveyed and implemented by [Quentin Bernet](#) for his Master thesis.

EXAMPLE 1: SIZED LISTS

```
def zip[A, B](  
  xs: List[A],  
  ys: List[B] with ys.size == xs.size  
): {zs: List[(A, B)] with zs.size == xs.size}
```

```
def concat[T](  
  xs: List[T],  
  ys: List[T]  
): {zs: List[T] with zs.size == xs.size + ys.size}
```

```
val xs: List[Int] = ???
```

```
val ys: List[Int] = ???
```

```
zip(concat(xs, ys), concat(ys, xs)) // 
```

```
zip(concat(xs, ys), concat(xs, xs)) //  found size xs.size + xs.size,  
// required xs.size + ys.size
```



First-Class Refinement Types for Scala

MATT BOVEL[✉], EPFL, Switzerland

VIKTOR KUNČAK, EPFL, Switzerland

MARTIN ODERSKY, EPFL, Switzerland

Refinement types—types qualified with logical predicates—have proven effective for lightweight verification in languages like Liquid Haskell, F*, and Dafny. However, in these systems refinements are either written in a separate specification language or treated as second-class annotations, disconnected from the host language’s type system. This disconnect creates usability barriers: programmers must maintain two mental models, and refinements cannot interact with features like type inference, subtyping, or overloading.

We present the design of first-class refinement types for Scala 3, where refinements are ordinary types that participate in subtyping, inference, and pattern matching alongside existing language features. We prove type soundness of a core, pure calculus mechanized in *Boogie*, combining dependent function types, bounded

REFINEMENT VS. ASSERTIONS

Why not use assertions instead?

```
def zip[A, B](  
  xs: List[A],  
  ys: List[B]  
) : List[(A, B)] = {  
  require(xs.size == ys.size)  
  ???  
}.ensuring(_.size == xs.size)
```

1. **Run-time overhead:** assertions are checked at run time, refinement types are checked at compile time (by default).

2. **No static guarantees:** assertions are only tested with the inputs you run them with. Refinement types hold *for all values*.

3. **Not part of the API:** assertions are not visible in the function type / scaladoc.

4. **Hard to compose:** assertions cannot be passed as type argument:

```
type Pos = {x: Int with x >= 0}  
def foo(l: List[Pos])  
def bar(f: Pos => Pos)
```

EXAMPLE 2: FANSI

From Li Haoyi's **Fansi** library for colored terminal strings:

With assertions:

```
def True(r: Int, g: Int, b: Int) =  
    True0(r, g, b, trueIndex(r, g, b))  
  
def trueIndex(r: Int, g: Int, b: Int) =  
    require(0 <= r && r < 256)  
    require(0 <= g && g < 256)  
    require(0 <= b && b < 256)  
    r << 16 | g << 8 | b
```

`True(256, 0, 0)` // ✨ at run time

With refinement types:

```
type Comp = {v: Int with 0 <= v && v < 256}  
  
def True(r: Comp, g: Comp, b: Comp) =  
    True0(r, g, b, trueIndex(r, g, b))  
  
def trueIndex(r: Comp, g: Comp, b: Comp) =  
    r << 16 | g << 8 | b
```

`True(256, 0, 0)` // 🚫 at compile time

ERROR MESSAGES

Predicates are type-checked like other Scala expressions:

```
def f[A](l: List[A] with l.nonEmpty) = () // error
```

```
-- [E008] Not Found Error: 03_error_messages.scala:9:27
```

```
9 |def f[A](l: List[A] with l.nonEmpty) = () // error
```

```
|           ^^^^^^^^^^^
```

```
|           value nonEmpty is not a member of List[A] - did you mean l.nonEmpty?
```

ERROR MESSAGES AND INFERENCE

Same inference and error reporting as for other Scala types:

```
type PosInt = {v: Int with v > 0}
def g[T](f: T => Unit, x: T) = f(x)
g((x: PosInt) => x * 2, -2) // error
```

```
-- [E007] Type Mismatch Error: 03_error_messages.scala:17:26
17 |   g((x: PosInt) => x * 2, -2) // error
    |                               ^^
    |                               Found:      (-2 : Int)
    |                               Required: {x: Int with x > 0}
```

EXAMPLE 3: OVERLOAD RESOLUTION

```
@targetName("minSorted")  
def min(l: List[Int] with l.isSorted) =  
    l.head // O(1)  
  
def min(l: List[Int]) =  
    l.min // O(n)  
  
def ex1(l: List[Int] with l.isSorted) =  
    min(l) // calls the first overload
```

REFINEMENT TYPES VS. LIBRARIES

With **Iron**, example 3 also works, but predicates are type-level constraints:

```
final class IsSorted
given Constraint[List[Int], IsSorted] with
  override inline def test(inline l: List[Int]): Boolean = l.isSorted
  override inline def message: String = "List must be sorted"

def min(l: List[Int] :| IsSorted): Int = l.head
```

WHAT WE WANT

Like libraries, unlike assertions:

1. **Compile-time guarantees** for *all* values, with no run-time cost (by default).

2. **Composability:**

```
List[{x: Int with x > 0}],  
{x: Int with x > 0} => String.
```

3. **Self-documenting APIs:** pre- and post-conditions live in signatures.

Unlike libraries:

4. **Term-level predicates:** plain Scala expressions, no type-level copy of each method (like `IsSorted` and its `Constraint`).

5. **More reasoning:** a specialized solver, beyond **type-level programming**.

6. **Better inference:** refinements introduced when needed, and from `if` conditions.

STRONG SPECIFICATIONS FOR AGENTS

Please implement `def foo.`

Done.

But this is wrong!

You're absolutely right!

STRONG SPECIFICATIONS FOR AGENTS

Please implement `def foo(): {res: Any with correct(res)}.`

Done and verified.

Refinement types give precise, checked interfaces: with agents, and between modules.

WHAT ARE VALID PREDICATES?

```
var x = 3  
val y: (Int with y == x) = x //  x is mutable
```

```
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 PREDICATES

By default, functions called in predicates are *assumed* to be pure:

```
class Cell { var field: Int = 0 }  
def test(c: Cell) =  
  def getAndBump(): Int = { c.field += 1; c.field }  
  val v: Int with v == getAndBump() = getAndBump() //  accepted
```

With **capture** and **separation checking**, predicates are required to be pure:

```
import language.experimental.{captureChecking, separationChecking}  
class Cell extends caps.Stateful { var field: Int = 0 }  
def test(c: Cell^) =  
  def getAndBump(): Int = { c.field += 1; c.field }  
  val v: Int with v == getAndBump() = getAndBump() //  not pure: uses {c}
```

WHEN ARE REFINEMENTS INFERRED?

Refinement types are **not** inferred from terms by default; the wider type is:

```
val x = 42
// x: Int, not {v: Int with v == 42}
```

Why not always infer the precise type?

1. **Backward compatibility:** implicit and overload resolution depend on inferred types; a more precise type changes which instances are found.

2. **Performance:** bigger types, slower comparisons.

3. **Developer experience:** users would be flooded with complex types.

Instead, precision is recovered **on demand**, when a refinement type is expected:

```
val x: (Int with x == 42) = 42
```

FLOW-SENSITIVE TYPING

The condition of an `if` is assumed in its branches:

```
def pos(x: Int): {v: Int with v > 0} =  
  if x > 0 then x else 1
```

In the **bound-checked merge sort**, `head` and `tail` require a non-empty sequence (`SafeSeq[T] with a.len > 0`):

```
def merge[T: Ordering as ord](left: SafeSeq[T], right: SafeSeq[T]): SafeSeq[T] =  
  if left.len > 0 && right.len > 0 then  
    if ord.lt(left.head, right.head) then SafeSeq(left.head) ++ merge(left.tail, right)  
    else SafeSeq(right.head) ++ merge(left, right.tail)  
  else if left.len == 0 then right  
  else left
```

SOLVER

How does the compiler check

```
{x: T with p(x)} <: {x: T with q(x)} ?
```

Need to check that `p(x)` implies `q(x)` for all `x: T`.

Existing implementations in other languages lower to SMT solvers.

Due to packaging and performance concerns, we instead implemented our own **lightweight e-graph-based solver**.

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

And domain-specific normalizations such as:

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

Also beta-reduction, selection of case class fields (`Point(a, b).x` is `a`), and limited reasoning on linear inequalities.

RUN-TIME CHECKS

Pattern matching branches on whether the predicate holds at run time:

```
val idRegex = "[0-9a-f]{4}-[0-9a-f]{4}"  
type ID = {s: String with s.matches(idRegex)}
```

```
"a2e7-e89b" match  
  case id: ID => ... // holds, id has type ID  
  case _      => ... // does not hold
```

RUN-TIME CHECKS: `.runtimeChecked`

You can also use `.runtimeChecked` (SIP-57) when you expect the check to 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]` to skip the check altogether.

`.runtimeChecked` for refinement types was implemented by [Valentin Schneeberger](#).

GRADUAL TYPING

By default, a refinement that cannot be verified statically is a type error:

```
val x: {v: Int with v == 1} = readInt() //  type error
```

A language import turns these errors into something else:

```
import language.experimental.qualifiedTypes.runtimeChecks // or warn, or silent
```

Mode	Unverified refinement becomes
<code>runtimeChecks</code>	run-time check, throws if it fails
<code>warn</code>	warning, no run-time check
<code>silent</code>	accepted, no run-time check

VERIFYING WITH STAINLESS

The **Stainless** verifier can use Dotty's refinement types, thanks to **Kasia Marek**!

It compiles in `silent` mode, then verifies refinements itself with SMT solvers:

```
def sum(n: BigInt with n >= 0): {r: BigInt with 2 * r == n * (n + 1)} =  
  if n == 0 then BigInt(0) else n + sum(n - 1) //  verified
```

```
def abs(x: Int): {r: Int with r >= 0} =  
  if x >= 0 then x else -x //  counter-example: x = -2147483648
```

See also: a verified **merge sort**: sorted, same size, same elements.

EXAMPLE 4: MEMOIZED FIBONACCI

A map whose values are related to their keys by a predicate `p`:

```
case class DepMap[K, V](p: (K, V) => Boolean):  
  def put(n: K, v: V with p(n, v)): Unit = ???  
  def get(n: K): Option[{res: V with p(n, res)}] = ???
```

A function whose result type is its own definition:

```
def fib(n: Int): {r: Int with r == (if n <= 1 then 1 else fib(n - 1) + fib(n - 2))} =  
  if n <= 1 then 1 else fib(n - 1) + fib(n - 2)
```

EXAMPLE 4: MEMOIZED FIBONACCI (2)

A memoized version, checked to always return `fib(n)`:

```
val cache: (DepMap[Int, Int] with (cache == DepMap[Int, Int]((k, v) => v == fib(k)))) =  
  DepMap[Int, Int]((k, v) => v == fib(k))  
  
def fibMemo(n: Int): {res: Int with res == fib(n)} =  
  cache.get(n) match  
    case Some(res) => res  
    case None =>  
      val res: (Int with (res == fib(n))) =  
        if n <= 1 then 1 else fibMemo(n - 1) + fibMemo(n - 2)  
      cache.put(n, res)  
      res
```

EVALUATION

Compilation overhead, against each system's own unchecked baseline:

System	Overhead
First-class (ours)	0–13%
Schmid and Kunčák	20–41%
Stainless	$\geq 53\%$

When the feature is unused, the full Dotty CI passes unchanged: the compiler itself ($\approx 200\,000$ LoC), $\approx 10\,000$ tests, a 50-project community build.

$\leq 4\%$ slowdown on the official benchmark suite.

FUTURE WORK

- **Pre-SIP and SIP** to make refinement types part of Scala.
- **Better solver** for what our lightweight solver cannot do.
- **A termination checker** (Master semester project from [Ozair Faizan](#) to be included).
- **Term-parameterized types**, to modularize predicates:

```
type Range(from: Int, to: Int) =  
  {v: Int with v >= from && v < to}
```

HOW TO USE

Try it online in the [playground](#), and report issues at github.com/epfl-lara/dotty.

Scala CLI: add these directives and run `scala --cli-version 1.18.0 run hello.scala`:

```
//> using scalaOrganization ch.epfl.lara  
//> using scala 3.10.1-RC1-bin-20260903-e1f9361-NIGHTLY  
//> using options -language:experimental.qualifiedTypes
```

sbt: in `build.sbt`:

```
scalaOrganization := "ch.epfl.lara"  
scalaVersion := "3.10.1-RC1-bin-20260903-e1f9361-NIGHTLY"  
scalacOptions += "-language:experimental.qualifiedTypes"
```

THANK YOU!



Slides



Paper



Playground



*Un type raffiné,
by Marina Granados Castro*

BONUS SLIDES



2026

ScalaDays

BERLIN

BACKUP: BOUNDED POLYMORPHISM

Inspired by [Abstract Refinement Types](#) (Vazou et al., ESOP 2013).

```
given Ord[Int] = ...
```

```
def max[T: Ord as ord, U <: T](x: U, y: U): U =  
  if ord.compare(x, y) >= 0 then x else y
```

```
type Even = {v: Int with v % 2 == 0}
```

```
def ex2: Even =  
  maximum(List(2, 4, 6))
```

```
def maximum[T: Ord, U <: T](xs: List[U]): U =  
  xs.reduce(max)
```

REFINEMENT TYPES VS. EXISTING CONSTRUCTS

Scala already has *structural refined types*:

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

And singleton types, and opaque types, and compile-time operations:

```
opaque type SizedList[  
  X <: Int: Singleton,  
  A  
] <: List[A]
```

```
def zip[A, B, Size <: Int: Singleton](  
  as: SizedList[Size, A],  
  bs: SizedList[Size, B]  
): SizedList[Size, (A, B)]
```

SUBTYPING: CONSTANT FOLDING

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

2026

ScalaDays

BERLIN

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: 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

Literal types are subtypes of singleton refinement types:

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

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

BACKUP: RESULT TYPES AS ASSUMPTIONS

The solver uses the result types of called functions as assumptions:

```
def double(x: Int): {res: Int with res == x + x} = x + x
val d: {res: Int with res == 6} = double(3)
```

```
def inc(x: Int): {res: Int with res == x + 1} = x + 1
val w: {res: Int with res == 5} = inc(inc(3))
```

This rule needs the called function to terminate: a result type says nothing about a call that never returns.

BACKUP: *FIRST-CLASS?*

In our prototype, refinement types are ordinary Scala types, participating in inference, **overload resolution** (example 3), subtyping, etc.

In contrast, Liquid Haskell is a plugin that runs after type checking:

```
{-@ x :: {v:Int | v mod 2 == 0} @-}  
let x = 42 :: Int in ...
```

“It’s sort of like you’re doing two things at once [...] you’re also talking to GHC, but you’re also talking to LiquidHaskell.” — a participant in [Usability Barriers for Liquid Types](#) (Gamboa et al., PLDI 2025)

Our implementation is directly integrated into the Scala 3 compiler:

```
val x: (Int with x % 2 == 0) = 42
```

BACKUP: LH USABILITY BARRIERS

From “Usability Barriers for Liquid Types” [1]:

- 4.2 Unclear Divide between Haskell and LiquidHaskell:
 - “comments are usually seen as just optional information in the code and not something that is directly used by the compiler”
“It’s sort of like you’re doing two things at once because you’re implementing in Haskell. But you’re also talking to GHC, but you’re also talking to LiquidHaskell.”
- 4.7 Unhelpful Error Messages
 - “[...] error messages produced from typing errors inside the predicates, seemed indistinguishable from those produced by verification errors.”
- 4.8 Limited IDE Support
 - “[user] tried to use the function `length`, but since it was not imported, it was impossible to use in this case.”

[1] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. Proc. ACM Program. Lang. 9, PLDI, Article 224 (June 2025), 26 pages. [doi:10.1145/3729327](https://doi.org/10.1145/3729327)

BACKUP: `List.collect`

Scala type parameters are *erased* at run time, 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]
```