FIRST-CLASS REFINEMENT TYPES FOR SCALA 3

Matt Bovel, EPFL
co-supervised by Viktor Kunčak (Stainless) and Martin Odersky (Scala)
work done in collaboration with Quentin Bernet and Valentin Schneeberger

January 17, 2026

REFINEMENT TYPES

Refinement types are types qualified with logical predicates.

For example,

$$\{x : \mathrm{Int} \mid x > 0\}$$

denotes the type of all integers `x` such that `x > 0`.

Implemented in many languages: Liquid Haskell, Boolean refinement types in F*, Subset types in Dafny, etc.

Prior art in Scala: SMT-based checking of predicate-qualified types for Scala (Schmid and Kunčak, Scala Symposium 2016), Refined, Iron.

OUTLINE

We present a work-in-progress implementation of refinement types in Scala 3, with focus on:

- **First-class integration:** implemented in the Scala compiler directly, not as a plugin or a separate tool.
- **Typing:** imprecise types by default, recover refinements when needed.
- **Runtime checks:** pattern matching and sugar.
- **Solver:** lightweight custom solver for subtyping.
- **Mechanization:** are we sound yet?

Un type qualifié, by Marina Granados Castro

SYNTAX

Consider the type of non-empty lists:

$$\{l : \text{List}[\text{A}] \mid l.\text{nonEmpty}\}$$

In Scala, we use `with` instead of `|` because the latter is already used for union types:

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

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

SYNTAX: SHORTHAND

When a binder already exists, such as in:

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

We can omit it:

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

The second version is desugared to the first.

SYNTAX: EXAMPLE SIZED LIST API

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

```
def concat[T](xs: List[T], ys: List[T]):
  {res: List[T] with res.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)) // error
```

FIRST-CLASS

Liquid Haskell is a plugin that runs after type checking.

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

In contrast, our implementation is directly integrated into the Scala 3 compiler:

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

Refinement type subtyping is checked during type checking, not as a separate phase. Early prototypes did this as a separate phase, it was more complex and less reliable.

FIRST-CLASS: ERROR MESSAGES

Predicates are type-checked like other Scala expressions:

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

```
-- [E008] Not Found Error: tests/neg-custom-args/qualified-
types/predicate_error.scala:1:27 ----------------------------
1 |def f[A](l: List[A] with l.notEmpty) = () // error
  |                         ^^^^^^^^^^
  |                         value notEmpty is not a member of List[A]
  |                         - did you mean l.nonEmpty?
```

FIRST-CLASS: ERROR MESSAGES AND INFERENCE

Same inference and error reporting as for other Scala types:

```
def g[T](f: T => Unit, x: T) = f(x)
g((x: PosInt) => x * 2, -2) // error
```

```
-- [E007] Type Mismatch Error: tests/neg-custom-args/qualified-
types/infer.scala:2:29 ------------------------
2 |  g((x: PosInt) => x * 2, -2) // error
  |                          ^^
  |                          Found:    (-2 : Int)
  |                          Required: {v: Int with v > 0}
```

FIRST-CLASS: OVERLOAD RESOLUTION

Consider the following two overloads of `min`:

```
/** Minimum of a list. O(n) */
def min(l: List[Int]): Int = l.min

/** Minimum of a sorted list. O(1) */
def min(l: List[Int] with l.isSorted): Int = l.head
```

The second, more efficient overload is called if the list is known to be sorted:

```
val l2: List[Int] with l2.isSorted = l.sorted
min(l2) // calls second overload
```

TYPING

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

```scala
val x: /* Int */ = 42
```

Why not type `x` as `{v: Int with v == 42}` directly?

Because it would:

1. **Not be backward compatible:** overload resolution and implicit search return different results for a type vs. a more precise subtype.
2. **Hurt UX:** users would be flooded with complex types.
3. **Hurt performance:** big types slow down type checking.

TYPING: SELFIFICATION

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

```
val x: {v: Int with v == 42} = 42
```

As a typing rule:

$$\frac{\Gamma \vDash a : A \qquad \text{firstorder}(A)}{\Gamma \vDash a : \{x : A \mid x == a\}}\,(\text{T-Self})$$

Selfification is standard in other refinement type systems. Typing based on the expected type is standard in Scala. We also do so for singleton types or union types.

TYPING: LOCAL UNFOLDING

The system can also recover precise selfified types from local definitions:

```
val v1: Int = readInt()
val v2: Int = v1
val v3: Int with (v3 == v1) = v2
```

Conceptually done by remembering definitions in a “fact context”:

$$\frac{\Gamma \vDash a : A \qquad \Gamma, x : A, \{x == a\} \vDash b : B \qquad \mathrm{firstorder}(A)}{\Gamma \vDash \mathtt{let}\ x : A = a\ \mathtt{in}\ b : \mathrm{avoid}(B, x)}\ (\text{T-LetEq})$$

Similar rule in System FR as Foundations for Stainless (Hamza, Voirol and Kunčak, OOPSLA 2019).

RUNTIME CHECKS

When static checking fails, a refinement 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 =>
        id: ID // matched, id has type ID
    case id      =>
        // default case, did not match
```

RUNTIME 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]` directly to skip the check altogether.

EXAMPLE: BOUND-CHECKED MERGE SORT

Specify a type for non-negative integers (`Pos`) and a safe division function:

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

def safeDiv(x: Pos, y: Pos with y > 1): {res: Pos with res < x} =
  (x / y).runtimeChecked
```

Define an opaque type for bound-checked sequences:

```
opaque type SafeSeq[T] = Seq[T]

object SafeSeq:
  def fromSeq[T](seq: Seq[T]): SafeSeq[T] = seq
  def apply[T](elems: T*): SafeSeq[T] = fromSeq(elems)
```

EXAMPLE: BOUND-CHECKED MERGE SORT (2)

Add some methods to `SafeSeq`:

```
extension [T](a: SafeSeq[T])
  def len: Pos = a.length.runtimeChecked
  def apply(i: Pos with i < a.len): T =  a(i)
  def ++(that: SafeSeq[T]): SafeSeq[T] = a ++ that
  def splitAt(i: Pos with i < a.len): (SafeSeq[T], SafeSeq[T]) =
    a.splitAt(i)
```

These methods are only defined for non-empty sequences:

```
extension [T](a: SafeSeq[T] with a.len > 0)
  def head: T = a.head
  def tail: SafeSeq[T] = a.tail
```

EXAMPLE: BOUND-CHECKED MERGE SORT (3)

We can match on non-empty sequences, ensuring `head` and `tail` are safe to use:

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

This would be simplified with flow-sensitive typing.

EXAMPLE: BOUND-CHECKED MERGE SORT (4)

`middle` is known to be less than `len`, so `splitAt` is safe to use:

```
def mergeSort[T: Ordering](list: SafeSeq[T]): SafeSeq[T] =
  val len = list.len
  val middle = safeDiv(len, 2)
  if middle == 0 then
    list
  else
    val (left, right) = list.splitAt(middle)
    merge(mergeSort(left), mergeSort(right))
```

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,
- equality reasoning.

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: EXTERNAL CHECKS

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 explicit checks only:

```
x: {v: Int with v > 0} // checked by the type checker
x.runtimeChecked: {v: Int with v > 0} // checked at runtime
x.externallyChecked: {v: Int with v > 0} // checked by an external tool
x.asInstanceOf[{v: Int with v > 0}] // unchecked
```

MECHANIZATION

Syntax of the language formalized so far:

$$
\begin{aligned}
A, B &::= X \mid \texttt{Unit} \mid \texttt{Bool} \mid \Pi x : A.\ B \mid \forall X.\ A \mid \{x : A \mid b\} \mid A \vee B \mid A \wedge B \\
a, b, f &::= \texttt{unit} \mid \texttt{true} \mid \texttt{false} \mid x \mid \lambda x : A.\ b \mid \Lambda X.\ b \mid f\ a \mid f[A] \\
&\quad \mid \texttt{let}\ x : A = b\ \texttt{in}\ a \mid a == b \mid \texttt{if}\ a\ \texttt{then}\ b_1\ \texttt{else}\ b_2
\end{aligned}
$$

Mechanization done in Rocq, using a definitional interpreter, semantic types, and Autosubst (for de Bruijn indices). Does not yet include the implication solver.

See Mechanizing Refinement Types (Borkowski, Vazou and Jhala, POPL 2024), Type Soundness Proofs with Definitional Interpreters (Amin and Tiark Rompf, POPL 2017), A Logical Approach to Type Soundness (Timany, Krebbers, Dreyer and Birkedal, Journal of the ACM 2024), Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions (Schäfer, Tebbi and Smolka, ITP 2015).

MECHANIZATION: INTERPRETATION

A semantic type is a predicate on values. The interpretation $[\![A]\!]^{\rho}_{\delta}$ maps a syntactic type A to a semantic type, given a type variable environment δ and a value environment ρ:

$$
\begin{aligned}
&\cdots \\
[\![\mathtt{Bool}]\!]^{\rho}_{\delta} &= \lambda v.\ v = \mathtt{true} \lor v = \mathtt{false} \\
&\cdots \\
[\![\{x : A \mid p\}]\!]^{\rho}_{\delta} &= \lambda v.\ [\![A]\!]^{\rho}_{\delta}(v) \land \rho, x \mapsto v \vdash p \Downarrow \mathtt{true} \\
&\cdots
\end{aligned}
$$

MECHANIZATION: TYPING

The rule for let-bindings that stores equalities in the fact context:

$$\frac{\Gamma \vDash a : A \qquad \mathrm{firstorder}(A) \qquad \Gamma, x : A, \{x == a\} \vDash b : B}{\Gamma \vDash \texttt{let}\ x : A = a\ \texttt{in}\ b : \mathrm{avoid}(B, x)}\ (\text{T-LetEq})$$

The rule for selfification:

$$\frac{\Gamma \vDash a : A \qquad \mathrm{firstorder}(A)}{\Gamma \vDash a : \{x : A \mid x == a\}}\ (\text{T-Self})$$

The rule for `if` expressions:

$$\frac{\Gamma \vDash a : \texttt{Bool} \qquad \Gamma, \{a == \texttt{true}\} \vDash b_1 : B_1 \qquad \Gamma, \{a == \texttt{false}\} \vDash b_2 : B_2}{\Gamma \vDash \texttt{if}\ a\ \texttt{then}\ b_1\ \texttt{else}\ b_2 : B_1 \vee B_2}\ (\text{T-If})$$

CONCLUSION

- **Syntax:** `{x: T with p(x)}`, can omit binder,
- **First-class:** integrates with Scala UX and features (overloading, extension methods, givens, etc.),
- **Typing:** imprecise types by default, can recover refinements using *selfification* and local unfolding,
- **Runtime checks:** pattern matching, `.runtimeChecked`,
- **Subtyping:** normalization, local unfolding, equality reasoning, compatibility with other types,
- **Future work:** flow-sensitive typing, external checks,
- **Mechanization:** System F with refinement types and more, using a definitional interpreter and semantic types.

Un type qualifié, by Marina Granados Castro

BACKUP: PREDICATE RESTRICTIONS

```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?

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

BACKUP: `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]
```

BACKUP: SPECIFY USING ASSERTIONS 😕

We can use assertions:

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

BACKUP: SPECIFY USING DEPENDENT TYPES 😕

Can we use path-dependent types?

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

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 (2)

This would be required for reasoning with refinement types inside cases:

```
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)
```

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: LOCAL UNFOLDING

As mentioned, refinement 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

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.