First-Class Refinement Types in Scala

Matt Bovel
matthieu@bovel.net
EPFL
Lausanne, Switzerland

1 Introduction

In this talk, we discuss usability and pratical aspects of our prototype implementation of *refinement types* for Scala 3. A refinement type, written `{x: T with p(x)}`, denotes the subset of values `x` of type `T` for which `p(x)` holds. For example, `{x: List[Int] with x.nonEmpty}` represents the type of non-empty lists of integers. Similar constructs exist in other languages under different names: *refinement types* in Liquid Haskell [5], *boolean refinement types* in F*, *subset types* in Dafny or *subtypes* in Lean.

In the main existing implementation of refinement types, *Liquid Haskell*, refinement types are written as comments, ignored by the type checker, and verified in a separate phase. For example, the following declaration states that the binding `x` of type `Int` has the refinement type of even integers:

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

Because Liquid Haskell refinements are written in comments and processed separately, they remain second-class: harder to debug and less integrated with the language. A recent usability study [3] observed that "comments are usually seen as just optional information in the code and not something that is directly used by the compiler," and one participant remarked, "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."

Our work takes the opposite approach: refinement types are made first-class in Scala, written, inferred, and checked like ordinary Scala types. This aims to preserve the language's look and feel and enabling integration with existing features.

2 Syntax

Several syntactic variants were explored in the context of Quentin Bernet's Master's thesis [1]. The syntax adopted in this proposal was chosen for its clarity and compatibility with existing Scala constructs.

`{x: T with p(x)}` is the *long-form* syntax for refinement types. It introduces an explicit binder `x` for the value being refined and is useful when no name is already available, for example in type aliases or function return types:

```
type Pos = {x: Int with x > 0}
def fill(n: Pos, v: Int):
  {res: List[Int] with res.size == n} = ???
```

When the refined value already has a name, such as in a `val` or parameter declaration, the binder can be omitted. This *short form* reuses the existing name in the predicate and desugars to the long form:

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

3 Mixed-Precision Type Inference

Scala already supports precise types such as literal types: for example, the literal 42 can be given the singleton type 42. However, this precision is typically lost due to *widening*: `val x = 42` is inferred to have type `Int`, unless an explicit type annotation is provided, as in `val x: 42 = 42`. Our approach generalizes this mechanism into a form of mixed-precision type inference, where refinement information is preserved only when relevant, avoiding unnecessary precision that could harm performance or compatibility.

Refinement types are not inferred by default; they are introduced only when the bidirectional typing algorithm needs to check an expression against a refinement type. In such cases, the compiler attempts to *selfify* the expression—that is, to give `e: T` the refinement type `{x: T with x == e}`. This allows valid expressions to be lifted into types.

Additionally, the typing context is enriched with equality facts, which allow the system to recover precision lost during widening. These facts are introduced by the following typing rule for `let` bindings:

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1,\ x = e_1 \vdash e_2 : T_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \text{avoid}(T_2, x)} \quad \text{(T-Let)}$$

Where `avoid(T2, x)` removes references to `x` in `T2` (that avoids the need for well-formedness conditions as in [2], or explicit existential introduction as in [4], and is closer to the algorithmic typing strategy used in Scala).

The T-Let rule enables later recovery of precision lost after widening. For example, in the following code, `x` is typed as `Int`, but `y` can recover a precise refinement:

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

These mechanisms are key to maintaining backward compatibility: they allows existing programs to type-check unchanged while still benefiting from additional precision when refinements are expected.

Acknowledgments

We would like to thank Viktor Kunčak, Martin Odersky, Sébastien Doeraene, Guillaumes Martes, Dimi Racordon, Eugène Flesselle, and Hamza Remmal for the helpful discussions.

References

[1] Quentin Bernet. 2024. *Syntax and Runtime Checks for Qualified Types in Scala 3*. Master's thesis. École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland.

[2] Michael H. Borkowski, Niki Vazou, and Ranjit Jhala. 2024. Mechanizing Refinement Types. *Proc. ACM Program. Lang.* 8, POPL, Article 70 (Jan. 2024), 30 pages. doi:10.1145/3632912

[3] 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

[4] Jad Hamza, Nicolas Voirol, and Viktor Kunčak. 2019. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 166 (Oct. 2019), 30 pages. doi:10.1145/3360592

[5] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 269–282. doi:10.1145/2628136.2628161