Logically Qualified Types for Scala 3

Matt Bovel
matthieu@bovel.net
EPFL
Lausanne, Switzerland

Abstract

We present a new design and prototype implementation of refinement types for Scala, enabling types to be refined with logical predicates and statically checked by the compiler. Unlike previous work, our system integrates qualified types directly into the Scala type checker and performs subtyping via a lightweight, compiler-internal solver based on E-Graphs and expression normalization. The implementation builds on earlier student work, which explored syntax design and runtime checking mechanisms

CCS Concepts: • **Software and its engineering → Compilers**; **Constraints**; **Formal software verification**.

Keywords: Refinement Types, Scala

1 Introduction

This talk presents a proposal and prototype implementation of *qualified types* for Scala 3. A qualified type refines a base type `T` with a logical predicate `p(x)`, written as `{x: T with p(x)}`. It denotes the subset of values `x` of type `T` for which `p(x)` holds. For example, `{x: Int with x > 0}` represents the type of positive integers, and `{x: List[Int] with x.nonEmpty}` the type of non-empty lists of integers.

Similar constructs exist in other languages under different names: *refinement types* in Liquid Haskell [11], *boolean refinement types* in F*, *subset types* in Dafny or *subtypes* in Lean. We use the term *logically qualified types* or *qualified types* for short in Scala to avoid confusion with the existing *refined types*.

In essence, qualified types embed assertions in the type system: to accept `e: { x: T with p(x) }`, the compiler must prove that for any `e: T`, `p(e)` holds. In other words, `assert(p(e))` can never fail at runtime.

Embedding these checks in the type system brings three key benefits. First, predicates are discharged at compile time, eliminating runtime overhead and guaranteeing correctness for *all* values, not just those tested at runtime. Second, qualified types compose naturally, as they can be passed as type arguments to construct more complex types without additional boilerplate: `{ x: Int with x > 0 } => String` denotes a function that accepts only positive inputs, and `List[{ x: Int with x > 0 }]` a list of positive integers. Third, they make APIs more concise: predicates live in the signature, removing the need for separate assertions and improving IDE support and documentation.

2 Syntax and Predicate Language

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

`{x: T with p(x)}` is the *long-form* syntax for qualified types. It introduces an explicit binder `x` for the value being qualified, and is useful when a name is not 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 qualified value already has a name, for example 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 > 0
// desugars to:
val x: {x: Int with x > 0}
```

In addition to surface syntax, predicates must satisfy semantic restrictions to ensure they can be encoded into a logical form suitable for static reasoning. In particular, predicates must be *pure*: they must return the same result whenever evaluated with the same argument, during and across program executions.

To reflect this, the predicate language is restricted to a fragment of Scala consisting of constants, stable identifiers, field selections over `val` fields, term applications, type applications, and constructors of case classes without initializers. Functions called inside predicates are expected to be pure, but the system does not currently enforce this property mechanically. It is considered a responsibility of the programmer to avoid impure functions in this context.

3 Selfification

Scala already supports precise types such as literal types: 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: `val x: 42 = 42`. Qualified types follow the same principle: they are not inferred from terms by default. This avoids changing the types that Scala would normally assign, preserving source compatibility and avoiding a proliferation of fine-grained types that could impact performance.

Instead, qualified types must be written explicitly. When such a type is expected, the compiler attempts to *selfify* the expression: that is, to give `e: T` the qualified type `{x: T with x == e}`. This allows expressions to be lifted into types, as long as they are valid predicates under the restrictions described in the previous section.

```
val x: (Int with x == 42) = 42
val y: (Int with y == n + 2) = n + 2
```

4 Runtime checks

When a value's properties cannot be verified statically, runtime checks can be performed using pattern matching (as implemented in [1]):

```
type ID = {s: String with s.matches(idRegex)}
"12e7-e89b-12d3" match
  case _: ID => // s matches idRegex
  case _     => // s does not match idRegex
```

When the program should fail if the predicate is not satisfied, the `runtimeChecked` method can be used. It performs a dynamic check and throws an exception if the predicate does not hold. This mechanism was implemented by Valentin Schneeberger as part of his Bachelor's thesis [8].

As with pattern matching in Scala more generally, matching against type parameters is not supported, since type arguments are erased at runtime. For example, one cannot match against a `List[ID]` or a `ID => String` type. For collections and in others cases, this limitation can be lifted by defining a custom `TypeTest` instance.

5 Subtyping

To determine whether a qualified type `{x: T with p(x)}` is a subtype of another `{y: S with q(y)}`, the system checks if `T <: S` and if the predicates are related by logical implication—that is, if `p(x)` implies `q(x)` for all `x`.

To achieve this, our implementation uses a lightweight custom solver designed to handle common cases efficiently. It combines equality reasoning, normalization, and predicate inlining from both types and definitions.

Equality reasoning. The solver is based on the E-Graph data structure, originally introduced by Nelson [5] and more recently popularized through EGG [12]. E-Graphs efficiently compute the congruence closure of a set of equalities, allowing the solver to track when different expressions are known to be equivalent. For example, from `v == a` and `a > 3`, it can deduce `v > 3`.

Normalization. The solver also performs syntactic normalization to bring predicates into canonical form. This includes reordering and regrouping terms in sums and products over `Int` and `Long`. For example, `x + 3 * y` and `2 * y + x + y` are treated as equivalent. This is similar to the `ring` tactic in Coq and the `ring_nf` tactic in Lean.

Qualifiers flattening and unfolding. When a predicate refers to a value whose qualified type is known, the solver can inline the associated predicate. For example, given `y: Int with y == x + 1`, the predicate `v == y + 1` is flattened into `y == x + 1 && v == y + 1`, allowing the solver to conclude `v == x + 2`. If a referenced term does not have a qualified type, the solver can instead inspect its local definition directly and apply selfification on demand. For example, if `y` is defined as `val y: Int = x + 1`, the solver assumes `y == x + 1`, leading to the same conclusion. This form of local unfolding complements type-based flattening and helps eliminate trivial indirections.

Compatibility with other Scala types. Qualified types integrate with other parts of the Scala type system. For example, a literal type `1` is a subtype of `{x: Int with x == 1}`, and therefore also of `{x: Int with x > 0}`.

6 Related Work

In the Scala ecosystem, libraries such as Refined[10] and Iron[4], offer user-level encodings of refinement types using opaque type aliases and implicit evidences. Compared to these approaches, our system aims to increase expressiveness, reduce boilerplate, and improve performance by integrating refinement checks directly into the compiler.

A previous prototype for refinement typing in Scala was presented at the Scala Symposium 2016 [7]. That design implemented a second type-checking phase and delegated predicate checking to the Stainless verifier [3]. It required a separate inference algorithm which proved difficult to implement. Although the architecture was not pursued further, it has been the a source of inspiration for the present work.

7 Future Work

Our immediate next step is to submit a SIP proposal to integrate qualified types into the Scala 3 compiler.

Longer-term goals include formalizing the system, proving its soundness, and potentially building on foundations like those developed for Liquid Haskell [2] and in work on refinement-typed Featherweight Java [9].

While the current solver is designed to be lightweight and predictable, we envision extending it with SMT solvers such as Princess [6] to handle predicates beyond its current scope. This would enable more complete reasoning about theories like linear arithmetic and algebraic data types—beyond the limited normalization rules currently implemented.

Additional directions include support for existentials inside predicates, enabling abstraction over unknown values, and flow-sensitive typing, which would allow refining types based on control-flow information.

Acknowledgments

We would like to thank Viktor Kunčak, Martin Odersky, Sébastien Doeraene, Guillaumes Martes, Dimi Racordon, Eugène Flesselle, and Hamza Remmal for their guidance and 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] 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

[4] Iltotore. 2021. Iron: A compile-time and runtime refinement library for Scala 3. https://github.com/Iltotore/iron. Accessed: 2025-07-17.

[5] Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (April 1980), 356–364. doi:10.1145/322186.322198

[6] Philipp Rümmer. 2008. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LNCS, Vol. 5330)*. Springer, 274–289.

[7] Georg Stefan Schmid and Viktor Kuncak. 2016. SMT-based checking of predicate-qualified types for Scala. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala* (Amsterdam, Netherlands) *(SCALA 2016)*. Association for Computing Machinery, New York, NY, USA, 31–40. doi:10.1145/2998392.2998398

[8] Valentin Schneeberger. 2024. *Runtime Checks for Qualified Types in Scala 3.* Bachelor's thesis. École Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland.

[9] Ke Sun, Di Wang, Sheng Chen, Meng Wang, and Dan Hao. 2024. Formalizing, Mechanizing, and Verifying Class-Based Refinement Types. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 39:1–39:30. doi:10.4230/LIPIcs.ECOOP.2024.39

[10] Frank S. Thomas. 2015. refined: Refined is a Scala library for refinement types. https://github.com/fthomas/refined. Accessed: 2025-07-17.

[11] 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

[12] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304