Qualifying System $F_{<:}$

Some terms and conditions may apply

EDWARD LEE, University of Waterloo, Canada
YAOYU ZHAO, École Polytechnique Fédérale de Lausanne, Switzerland
ONDŘEJ LHOTÁK, University of Waterloo, Canada
JAMES YOU, University of Waterloo, Canada
KAVIN SATHEESKUMAR, University of Waterloo, Canada
JONATHAN IMMANUEL BRACHTHÄUSER, University of Tübingen, Germany

Type qualifiers offer a lightweight mechanism for enriching existing type systems to enforce additional, desirable, program invariants. They do so by offering a restricted but effective form of subtyping. While the theory of type qualifiers is well understood and present in many programming languages today, polymorphism over type qualifiers remains an area less well examined. We explore how such a polymorphic system could arise by constructing a calculus, System $F_{<:Q}$, which combines the higher-rank bounded polymorphism of System $F_{<:}$ with the theory of type qualifiers. We explore how the ideas used to construct System $F_{<:Q}$ can be reused in situations where type qualifiers naturally arise—in reference immutability, function colouring, and capture checking. Finally, we re-examine other qualifier systems in the literature in light of the observations presented while developing System $F_{<:Q}$.

CCS Concepts: Software and its engineering → General programming languages; Semantics; Polymorphism.

Additional Key Words and Phrases: System $F_{<:}$, Type Qualifiers, Type Systems

ACM Reference Format:

1 INTRODUCTION

Static type systems classify the values a program reduces to. For example, the signature of the function

```
    def toLowerCase(in: String): String = { ... }
```

enforces that it takes in a String as an argument and returns a String as a result. If strings are implemented as mutable heap objects, how would we express the additional property that toLowerCase does not mutate its input?

---

Authors’ addresses: Edward Lee, University of Waterloo, Waterloo, ON, Canada, e45lee@uwaterloo.ca; Yaoyu Zhao, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, yaoyu.zhao@uwaterloo.ca; Ondřej Lhoták, University of Waterloo, Waterloo, ON, Canada, olhotak@uwaterloo.ca; James You, University of Waterloo, Waterloo, ON, Canada, j35you@uwaterloo.ca; Kavin Satheeskumar, University of Waterloo, Waterloo, ON, Canada, ksatheeskumar@uwaterloo.ca; Jonathan Immanuel Brachthäuser, University of Tübingen, Tübingen, BaWü, Germany, jonathan.brachthaeuser@uni-tuebingen.de.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2024 Copyright held by the owner/author(s).
ACM 2473-1421/2024/4-ART115
https://doi.org/10.1145/3649832

Proc. ACM Program. Lang., Vol. 8, No. OOPSLA1, Article 115. Publication date: April 2024.
There are at least two ways to address this. We can view the modification of `toLowerCase`'s argument as a property of `toLowerCase` or we can view mutability as a property of the argument string in itself. The former viewpoint leads to solutions like (co-)effect systems [Petricek et al. 2014] that describe the relation of a function to the context it is called in. The latter viewpoint, of viewing it as a property of the argument, leads to systems that enrich the types of values with additional information. In this paper, we adopt the latter view.

One such system is Type qualifiers [Foster et al. 1999], in which we could qualify the type of `toLowerCase`'s argument with the type qualifier `const` to express that `toLowerCase` cannot modify its argument. We may choose to annotate its result with the type qualifier `const` to indicate that its result is a `const` String which cannot be changed by `toLowerCase`'s caller.

```python
def toLowerCase(in: const String): const String = {...}
```

The function `toLowerCase` now accepts a read-only `String` as an argument and presumably returns a new `String` that is a copy of its argument except in lowercase. More importantly, since the input string is qualified as `const`, we know that this version of `toLowerCase` cannot mutate the input string, for example, by calling a method like `in.setCharAt(0, 'A')`, which would replace the character of index 0 of the string with the character A.

Perhaps this is too restrictive. After all, `toLowerCase` will allocate a new `String` and does not impose invariants on it; its caller should be permitted to mutate the value returned. We should instead annotate `toLowerCase` as follows, with a `mutable` qualifier on its return value.

```python
def toLowerCase(in: const String): mutable String = {...}
```

Subtyping naturally arises in this context—a `mutable` `String` can be a subtype of a `const` `String`. Returning a `mutable` string should not cause existing calls to `toLowerCase` to break.

Similarly, it would be impractical if `toLowerCase` only accepted read-only `Strings`. After all, any operation one could perform on a read-only `String` should be semantically valid on a `mutable` `String` as well. Therefore a `mutable` `String` not only can but should be a subtype of `const` `String`. For example, we may have other `String` functions that accept `const` `Strings`, which should also accept `mutable` `Strings` as well.

```python
def reversed(in: const String): mutable String = {...}
reversed(toLowerCase("HELLO WORLD")) == "dlrow olleh"
```

Foster et al. [1999] were the first to recognize this natural subtyping relation induced by type qualifiers, which permitted type qualifiers to be integrated easily into existing type systems with subtyping. Perhaps the most well known qualifier is `const`. It is used to mark particular values as read-only or immutable and it is found in many languages and language extensions [Bright et al. 2020; Stroustrup 2007; Tschantz and Ernst 2005]. Other languages, such as OCaml and Rust, are exploring more exotic qualifiers to encode properties like locality, linearity, exclusivity, and synchronicity [Slater 2023a,b; Wuyts et al. 2022]. Qualifiers are so easy to use that many type system extensions start as type qualifier annotations on existing types; for Java there are multiple frameworks [Markstrun et al. 2010; Papi et al. 2008] for doing so, which have been used to model extensions of Java for checking nullability, energy consumption, and determinism amongst others.

While type qualifiers themselves are well-explored, qualifier polymorphism is still understudied. Sometimes parametric polymorphism is not necessary when subtyping is present. For example, the type signature that we gave to `toLowerCase`, `const` `String` => `mutable` `String`, is indeed the most permissive type that may be assigned. In languages with subtyping, type variables are only necessary to relate types and qualifiers in both return (covariant) and parameter (contravariant) positions; otherwise we can use their respective type bounds [Dolan 2016, Chapter 4.2.1]. For example, while we could have made `toLowerCase` polymorphic using a qualifier variable `Q` over
the immutability of its input, such a change is unnecessary as we can simply replace \( Q \) with its upper bound \( \text{const} \) to arrive at the monomorphic but equally general version of toLowerCase from above.

```java
def toLowerCase[Q <: const](in: Q String): mutable String = {...}
```

However, variables are indeed necessary when relating types and qualifiers in covariant positions to types and qualifiers in contravariant positions. For example, consider a substring function. Which qualifiers should we assign its arguments and return value?

```java
def substring(in: ??? String, from: Int, to: Int): ??? String = {...}
```

It would be reasonable to expect that a substring of an immutable string should itself be immutable, but also a substring of a mutable string should be mutable as well. To express this set of new constraints, we need parametric qualifier polymorphism.

```java
def substring[Q <: const](in: Q String, from: Int, to: Int): Q String
```

We also need to consider how qualifier polymorphism interacts with type polymorphism. For example, what should be the type of a function like slice, which returns a subarray of an array? It needs to be parametric over the the type of the elements stored in the array, where the element type itself could be qualified. This raises the question: should type variables range over unqualified types or both unqualified and qualified types? Foster’s original system does not address this issue, and existing qualifier systems disagree on what type variables range over and whether or not type variables can be qualified at all. For reasons we will demonstrate later in Section 5, type variables should range over unqualified types; to achieve polymorphism over both types and qualifiers, we need both type variables and qualifier variables for orthogonality.

```java
def slice[Qa <: const, Qv <: const, T <: Any](in: Qa Array[Qv T]): Qa Array[Qv T]
```

Polymorphism over qualified types, though, can be recovered through a lightweight form of syntactic sugar, as we show in Section 5 as well.

```java
def slice[Q <: const, QT <: const Any](in: Q Array[QT]): Q Array[QT]
```

Another under-explored area is that of merging type qualifiers, especially in light of parametric qualifier polymorphism. For example, consider the type qualifiers throws and noexcept, expressing that a function may throw an exception or that it does not throw any exception at all. Without polymorphism, it is easy to combine qualifiers. For example, a function like combined, that calls both pure and exception-throwing functions, should be qualified with the union of the two qualifiers, throws, expressing that an exception could be thrown from the calling function.

```java
def pure() = 0 // noexcept () => Int

def impure() = throw new Exception("Hello") // throws () => Unit

def combined() = { pure(); impure() } // throws () => Unit
```

Things are more complicated in the presence of qualifier parametric higher-order functions, such as:

```java
def compose[A,B,C,Qf,Qg](f: Qf (A => B), g: Qg (B => C)): ??? (A => C) = (x) => g(f(x))
```

What should be the qualifier on the return type (\( A \to C \)) of compose? Intuitively, if either \( f \) or \( g \) throws an exception, then the result of compose should be qualified with throws, but if neither throws any exception, then the composition should be qualified with noexcept. Ideally we would like some mechanism for specifying the union of the qualifiers annotated on both \( f \) and \( g \).
Existing qualifier systems have limited support for these use cases. Foster et al. [1999]’s original system is limited to simple ML-style qualifier polymorphism with no mechanism for specifying qualifier-polymorphic function types, and has limited support for combining qualifiers. Dietl et al. [2011]’s Checker Framework presents a restricted, implicit view of qualifier polymorphism via Java annotations. Systems that do support explicit qualifier polymorphism like that of Gordon et al. [2012] partially ignore the interaction between combinations of qualifier variables and their bounds, or present application-specific subqualification semantics seen in Boruch-Gruszecki et al. [2023] or Wei et al. [2024]. Must this always be the case? Is there something in common that we can generalize and apply to give a design recipe for designing qualifier systems with subqualification and polymorphism?

We believe this does not need to be the case; we show that it is possible to add qualifier polymorphism without losing the natural lattice structure of type qualifiers, and that there is a natural way to reconcile type polymorphism with qualifier polymorphism as well.

To illustrate these ideas, we start by first giving a design recipe for constructing a qualifier-polymorphic enrichment System $F_{<:Q}$ of System $F_{<:}$ [Cardelli et al. 1991], much in the same way that Foster et al. [1999] gives a design recipe for adding qualifiers to a base simply-typed lambda calculus. Our recipe constructs a calculus with the following desirable properties:

- **Higher-rank qualifier and type polymorphism:** We show how to add higher-rank qualifier polymorphism to a system with higher-rank [Leivant 1983] type polymorphism in Section 2.3.

- **Natural subtyping with qualifier variables:** We show that the subtyping that type qualifiers induce extends naturally even when working with qualifier variables. We achieve this by using the free lattice [Whitman 1941] generated over the original qualifier lattice. We illustrate these ideas, first in a simplified context over a fixed two-point qualifier lattice in Section 2.3 and generalize to an arbitrary bounded qualifier lattice in Section 2.7.

- **Easy meets and joins:** As we generalize the notion of a qualifier to that of an element from the free (qualifier) lattice, we recover the ability to combine qualifiers using meets and joins.

Next, to demonstrate the applicability of our qualifier polymorphism design recipe, we show how one can model three natural problems – reference immutability, function colouring, and capture tracking, using the ideas used to develop System $F_{<:Q}$ in Section 3. We then discuss how type polymorphism can interact with qualifier polymorphism in Section 5 to justify our design choices. We then re-examine a selection of other qualifier systems in light of our observations developed in our free lattice-based subqualification recipe in Section 6 to see how their subqualification rules fit in our free lattice based design recipe. Finally, we close with a discussion of related and potential future work in Section 7.

Our soundness proofs are mechanized in the Coq proof assistant; details are discussed in Section 4.

## 2 QUALIFIED TYPE SYSTEMS

In this section, we introduce System $F_{<:Q}$, a simple calculus with support for qualified types as well as type- and qualifier polymorphism. We start off with a brief explanation of what type qualifiers are (Subsection 2.1), introduce System $F_{<:Q}$ (Subsection 2.3), and show that it satisfies the standard soundness theorems (Subsection 2.6).

### 2.1 A Simply-Qualified Type System

As Foster et al. [1999] observes, type qualifiers induce a simple, yet highly useful form of subtyping on qualified types. Consider a qualifier like $\text{const}$, which qualifies an existing type to be read-only. It comes equipped with a dual qualifier $\text{mutable}$ which qualifies an existing type to be mutable. The
Qualifiers | Description
--- | ---
`mutable <: const` | Mutability; a mutable value can be used where an read-only value is expected, regardless of whether or not it should [Boyland 2006]. A covariant qualifier, as `mutable` is often omitted.
`nexcept <: throws` | Exception safety; a function which throws no exceptions can be called anywhere a function which throws could. A contravariant qualifier, as `throws` is often omitted. [Maurer 2015]
`sync <: async` | Synchronicity; a function which is synchronous and does not suspend can be called in contexts where a function which is asynchronous and suspends could. Covariant, as `sync` is assumed by default.
`nonnull <: nullable` | Nullability; a value which is guaranteed not to be null can be used in a context which can deal with nullable values. Covariant, in systems with this qualifier – most values ought not to be null.

Fig. 1. Examples of type qualifiers

Type `const T` is a supertype of `mutable T`, for all types T; a mutable value can be used wherever an immutable value is expected. Other qualifier pairs induce a subtype, like `nexcept` and `throws`—it is sound to use a function which throws no exception in a context which would handle exceptions. Figure 1 provides an overview of some qualifiers and describes which invariants they model.

Often one of the two qualifiers is assumed by omission – for example `mutable` and `throws` are often omitted; references are assumed to be mutable unless otherwise specified, and similarly functions are assumed to possibly throw exceptions as well. Qualifiers like `const`, where the smaller qualifier is omitted, are positive, or covariant; by example, `const String` is a supertype of a unqualified `String`. Conversely, qualifiers like `nexcept` are negative, or contravariant; `String => String` is a subtype of `String => String`.

### 2.2 Qualifying a Language

The observation that qualifiers induce subtyping relationships allows language designers to seamlessly integrate support for type qualifiers into existing languages with subtyping. As Foster et al. [1999] point out, these qualifiers embed into a qualifier lattice structure \( L \), and they give a design recipe for enriching an existing type system with support for type qualifiers.

1. First, embed qualifiers into a lattice \( L \). For example, `const` and `mutable` embed into a two-point lattice, where `const` is \( \top \) and `mutable` is \( \bot \). Other example qualifiers (and their embeddings) are described in Figure 1.
2. Second, extend the type system so that it operates on qualified types – a pair \( \{ l \} T \) where \( l \) is a qualifier lattice element and \( T \) a base type from the original system. This is done as follows:
   - Embed qualifiers into the subtyping system. Typically, for two qualified types \( \{ l_1 \} T_1 \) and \( \{ l_2 \} T_2 \) such that \( l_1 \sqsubseteq l_2 \) and \( T_1 <: T_2 \), one will add the subtyping rule \( \{ l_1 \} T_1 <: \{ l_2 \} T_2 \).
   - Add rules for introducing qualifiers, typically in the introduction forms for typing values.
   - Finally, augment the other typing rules, typically elimination forms, so that qualifiers are properly accounted for. One may also additionally add an assertion rule for statically checking qualifiers as well.
2.3 (Higher-rank) Qualifier Polymorphism

Foster’s original work allows one to add qualifiers to an existing type system. As we discussed earlier, we want more, though:

1. Qualifier Polymorphism: Certain functions ought to be polymorphic in the qualifiers they expect. For example, from our introduction, we should be able to express a substring function which is polymorphic in the mutability of the string passed to it. While this is easy enough, as Foster et al. [1999] shows, the interaction of lattice operations with qualifier variables is not so easy, as we discuss below.

2. Merging Qualifiers: We often need to merge qualifiers when constructing more complicated values. Merging is easy when working with a lattice; we can just take the lattice’s underlying join ($\lor$) or meet ($\land$) operation. But how do we reason about meets or joins of qualifier variables? For example, in a noexcept qualifier system, we should be able to collapse the qualifier on the result of a function like twice, which composes a function with itself, from $Q \lor Q$ to just $Q$, the result of twice throws if $f$ throws or if $f$ throws, which is namely just if $f$ throws.

$$\text{def} \quad \text{twice}[A, Q](f: (A \rightarrow A) Q): (A \rightarrow A) Q = \text{compose}(f, f)$$

To achieve this, we need to extend qualifiers from just elements of a two-point lattice, as in Foster et al. [1999], to formulas over lattices which can involve qualifier variables in addition to elements of the original lattice. Moreover, we would like to relate these formulas as well. But how?

2.4 Free Lattices

As Whitman [1941] observed, there is a lattice which encodes these relations over these lattice formulas, namely, the free lattice constructed over the original qualifier lattice. Free lattices capture exactly the lattice formula inequalities that are true in every lattice. This is formally specified by the following two definitions. Here, we use $\lor$, $\land$, $\leq$ in place of $\lor$, $\land$, $\subseteq$ to distinguish lattice formulas in general from a lattice formula in a fixed, concrete lattice.

Definition 2.1. Let $X$ be a set of variables. Then $E(X)$, the set of lattice formulas generated by $X$, is recursively defined by:

1. If $x \in X$ then $x \in E(X)$.
2. If $f_1 \in E(X)$ and $f_2 \in E(X)$ then $f_1 \land f_2 \in E(X)$.
3. If $f_1 \in E(X)$ and $f_2 \in E(X)$ then $f_1 \lor f_2 \in E(X)$.

Definition 2.2. Let $X$ be a set of variables. Then $F(X)$, the free lattice generated by $X$, is the lattice over $E(X)$ with ordering relation $\leq$ where $f_1[X] \leq f_2[X]$ for two formulas $f_1[X], f_2[X] \in F(X)$ if and only if $f_1[X \rightarrow L] \subseteq f_2[X \rightarrow L]$ in every lattice $L$ and instantiation $L$ of the variables in $X$ to elements of $L$, up to equivalence modulo $\leq$.

For example, the inequality $x \land y \leq x$ would hold in the free lattice $F(x, y)$ as it is true in every lattice but the inequality $x \leq y$ would not hold, as there is a concrete lattice $L$ and instantiation of $x$ and $y$ to elements of $L$ where $x \subseteq y$ is not true – take $L$ to be the two element lattice $\{0, 1\}, \leq \subseteq$, $x$ to be $1$, and $y$ to be $0$; clearly $1 \geq 0$.

Now while Definition 2.2 defines the free lattice extrinsically by its universal property, it unfortunately does not give a construction for the free lattice. However, it is folklore that the following explicit construction gives rise to the free lattice as well.1

---

1This can be viewed as a recasting of Whitman [1941, Theorem 1] using Gentzen-style natural deduction rules, as Galatos [2023]; Jipsen [2001] note. Galatos [2023] also gives an algebraic proof of Theorem 2.3, for those who prefer that over the syntactic, inductive proof presented in our mechanization as part of the soundness proof of System $F_\epsilon; Q$. 

Proc. ACM Program. Lang., Vol. 8, No. OOPSLA1, Article 115. Publication date: April 2024.
Theorem 2.3 (Folklore). \( F(X) \) is the lattice over \( (E(X)/\leq, \leq) \), where \( \leq \) is a binary relation over \( E(X) \) defined by:

1. For every \( x \in X \), \( x \leq x \).
2. For every \( f_1, f_2, f_3 \in E(X) \), if \( f_3 \leq f_1 \) then \( f_3 \leq f_1 \lor f_2 \).
3. For every \( f_1, f_2, f_3 \in E(X) \), if \( f_3 \leq f_2 \) then \( f_3 \leq f_1 \lor f_2 \).
4. For every \( f_1, f_2, f_3 \in E(X) \), if \( f_3 \leq f_1 \) and \( f_3 \leq f_2 \) then \( f_1 \lor f_2 \leq f_3 \).
5. For every \( f_1, f_2, f_3 \in E(X) \), if \( f_1 \leq f_3 \) then \( f_1 \land f_2 \leq f_3 \).
6. For every \( f_1, f_2, f_3 \in E(X) \), if \( f_2 \leq f_3 \) then \( f_1 \land f_2 \leq f_3 \).
7. For every \( f_1, f_2, f_3 \in E(X) \), if \( f_3 \leq f_1 \) and \( f_3 \leq f_2 \) then \( f_3 \leq f_1 \land f_2 \).

It should not be surprising to see free lattices here; as Dolan [2016, Chapter 3] observed, free lattices can be used to model subtyping lattices with unions, intersections, and variables as well. This allows us to generalize Foster et al. [1999]’s recipe for qualifying types. Instead of qualifying types by elements of the qualifier lattice, we qualify types by elements of the \emph{free lattice} generated over that base qualifier lattice, and we support qualifier polymorphism explicitly with bounds, following System \( F_{<:} \), instead of implicitly at prenex position with constraints, as Foster et al. [1999] do.

2.5 System \( F_{<:Q} \)

We are now ready to present our recipe by constructing System \( F_{<:Q} \), a qualified extension of System \( F_{<} \) with support for type qualifiers, polymorphism over type qualifiers, as well as meets (\( Q \land R \)) and joins (\( Q \lor R \)) over qualifiers. We start by constructing a simplified version of System \( F_{<:Q} \) which models a free lattice over a two-point qualifier lattice to illustrate our recipe.

Assigning Qualifiers. In System \( F_{<:Q} \), we qualify types with the free lattice generated over a base two-point lattice with \( \top \) and \( \bot \), but provide no interpretation of \( \top \) and \( \bot \) as System \( F_{<:Q} \) is only a base calculus.

Syntax. Figure 2 presents the syntax of System \( F_{<:Q} \), with additions over System \( F_{<} \) highlighted in grey. Type qualifiers \( Q \) include not only \( \top \) and \( \bot \) as in Foster et al. [1999]’s original system. Here, in addition, we support qualifier variables \( Y \), as well as meets and joins over qualifiers. Type variables support polymorphism over \emph{unqualified} types. To support qualifier polymorphism, we add a new qualifier for-all form \( \forall(Y <: Q).t \). Similarly, on the term-level, we add qualifier abstraction \( \Lambda(Y <: Q).P.t \) and qualifier application \( s\{Q\} \).

Values and Qualifiers. To ensure that qualifiers have some runtime semantics in our base calculus, we add values with a qualifier expression \( P \) denoting the qualifier that value should be typed at and we add support for \emph{asserting} as well as \emph{upcasting} qualifier tags, following Foster et al. [1999, Section 2.2]. For example, the value \( \lambda(x : \text{Int})\top x \) would represent the integer identity function qualified at \( \top \).

While System \( F_{<:Q} \) does not provide a default tag for values, negative (or contravariant) qualifiers like \texttt{noexcept} would inform a default qualifier tag choice of \( \top \) – by default, functions are assumed to throw – and positive (or covariant) qualifiers like \texttt{const} would inform a default qualifier tag choice of \( \bot \) – by default, in mutable languages, values should be mutable. Put simply, the default value tag should correspond to the default, omitted, qualifier.

Semantics. The evaluation rules of System \( F_{<:Q} \) (defined in Figure 3) are largely unchanged from System \( F_{<} \). To support \emph{qualifier polymorphism}, we add the rule (\texttt{BETA-Q}) for reducing applications of a qualifier abstraction to a type qualifier expression. Finally, to ensure that qualifiers have some runtime semantics even in our base calculus, we add the rules (\texttt{UPQUAL}) and (\texttt{ASSERT}) for asserting
\(s, t ::= \)  
\[\begin{align*}
\lambda(x.p.t) & \quad \text{term abstraction} \\
x & \quad \text{term variable} \\
s(t) & \quad \text{application} \\
\Lambda(X <: S).p.t & \quad \text{type abstraction} \\
\Lambda(Y <: Q).p.t & \quad \text{qualifier abstraction} \\
s[S] & \quad \text{type application} \\
\Lambda(Y <: Q).p.s & \quad \text{qualifier application} \\
\text{upqual Q s} & \quad \text{qualifier upcast} \\
\text{assert Q s} & \quad \text{qualifier assertion}
\end{align*}\]

\(\Gamma ::= \)  
\[\begin{align*}
\cdot & \quad \text{empty} \\
\Gamma, x : T & \quad \text{term binding} \\
\Gamma, X <: S & \quad \text{type binding} \\
\Gamma, Y <: Q & \quad \text{qualifier binding}
\end{align*}\]

\(\sigma ::= \)  
\[\begin{align*}
\lambda(x.p.t) & \quad \text{term abstraction} \\
\Lambda(X <: S).p.t & \quad \text{type abstraction} \\
\Lambda(Y <: Q).p.t & \quad \text{qualifier abstraction} \\
\end{align*}\]

\(\mathcal{S} ::= \)  
\[\begin{align*}
\top & \quad \text{top type} \\
T \rightarrow T & \quad \text{function type} \\
X & \quad \text{type variable} \\
\forall(X <: S).T & \quad \text{for-all type} \\
\forall(Y <: Q).T & \quad \text{qualifier for-all type}
\end{align*}\]

\(\mathcal{P}, \mathcal{Q}, \mathcal{R} ::= \)  
\[\begin{align*}
\top, \bot & \quad \text{Top and bottom} \\
Y & \quad \text{Qualifier variables} \\
\bigwedge \mathcal{Q} & \quad \text{Meets and joins}
\end{align*}\]

\(\mathcal{V} ::= \)  
\[\begin{align*}
\lambda(x.p.t) & \quad \text{term abstraction} \\
\Lambda(X <: S).p.t & \quad \text{type abstraction} \\
\Lambda(Y <: Q).p.t & \quad \text{qualifier abstraction} \\
\end{align*}\]

\(\mathcal{C} ::= \)  
\[\begin{align*}
\top & \quad \text{or} \\
\bot & \quad \text{two-point lattice elements}
\end{align*}\]

\(\mathcal{L} ::= \)  
\[\begin{align*}
\top \sqsubseteq \bot & \quad \text{Lattice facts reminder: } \bot \sqsubseteq \bot, \bot \sqsubseteq \top, \top \equiv \top, \\
\bot \sqsubseteq \top & \quad \top \sqsubseteq \bot, \bot \sqsubseteq \top, \bot \sqsubseteq \bot, \top \sqsubseteq \top
\end{align*}\]

\(\mathcal{G} ::= \)  
\[\begin{align*}
\lambda(x.p.t) & \quad \text{term abstraction} \\
\Lambda(X <: S).p.t & \quad \text{type abstraction} \\
\Lambda(Y <: Q).p.t & \quad \text{qualifier abstraction}
\end{align*}\]

Fig. 2. The syntax of System \(F_{\mathcal{Q}}\). Qualified differences to System \(F_{\mathcal{Q}}\) highlighted in grey.
Qualifying System $F_{<:Q}$

**Evaluation for System $F_{<:Q}$**

$$ (\lambda (x)p.t)(s) \rightarrow t[x \mapsto s] \quad \text{(BETA-V)} $$

$$ (\Lambda (X <: S)p.t)(S') \rightarrow t[X \mapsto S'] \quad \text{(BETA-T)} $$

$$ (\Lambda (Y <: Q)p.t)(Q') \rightarrow t[Y \mapsto Q'] \quad \text{(BETA-Q)} $$

<table>
<thead>
<tr>
<th>$v$ tagged with $P$</th>
<th>eval($P$) $\sqsubseteq$ eval($Q$)</th>
<th>(UPQUAL)</th>
</tr>
</thead>
</table>

| assert $P \upsilon$ | eval($P$) | (ASSERT) |

$$ s \rightarrow t \quad \text{and} \quad \text{eval } Q $$

**Partial Qualifier Evaluation**

$$ \text{eval}(P) ::= \\
| C \quad => \quad C \\
| P \land R \quad => \quad \text{eval}(P) \sqcap \text{eval}(R) \\
| P \lor R \quad => \quad \text{eval}(P) \sqcup \text{eval}(R) \\
| \_ \quad => \quad \text{nothing, otherwise.} $$

Fig. 3. Reduction rules for System $F_{<:Q}$

**Subqualification for System $F_{<:Q}$**

$$ \Gamma \vdash Q <: \top \quad \text{(SQ-TOP)} $$

$$ \Gamma \vdash \bot <: Q \quad \text{(SQ-BOT)} $$

$$ \Gamma \vdash Q <: R_1 \quad \text{(SQ-JOIN-INTRO-1)} $$

$$ \Gamma \vdash Q <: R_1 \lor R_2 \quad \text{(SQ-JOIN-INTRO-2)} $$

$$ \Gamma \vdash R_1 <: Q \quad \Gamma \vdash R_2 <: Q \quad \text{(SQ-JOIN-ELIM)} $$

$$ \Gamma \vdash R_1 \land R_2 <: Q \quad \text{(SQ-MEET-ELIM-1)} $$

$$ \Gamma \vdash R_1 \lor R_2 <: Q \quad \text{(SQ-MEET-ELIM-2)} $$

$$ \Gamma \vdash Q <: R_1 \quad \Gamma \vdash Q <: R_2 \quad \text{(SQ-MEET-INTRO)} $$

$$ Y <: Q \in \Gamma \quad \Gamma \vdash Q <: R \quad \text{(SQ-VAR)} $$

$$ \Gamma \vdash Y <: R \quad \text{(SQ-REFL-VAR)} $$

Fig. 4. Subqualification rules of System $F_{<:Q}$.

another qualified type $\{Q_2\}$ $S_2$ only if the qualifiers are in a subqualification relationship $Q_1 <: Q_2$, and the simple types are as well: $S_1 <: S_2$. 

Proc. ACM Program. Lang., Vol. 8, No. OOPSLA1, Article 115. Publication date: April 2024.
Subtyping for System $F_{<:Q}$

\[
\begin{align*}
\Gamma \vdash S <: T & \quad \text{(SUB-TOP)} \\
X \in \Gamma & \quad \text{(SUB-REFL-SVAR)} \\
\Gamma \vdash X <: X & \\
X <: S_1 \in \Gamma & \quad \Gamma \vdash S_1 <: S_2 \\
\Gamma \vdash X <: S_2 & \quad \text{(SUB-SVAR)} \\
\Gamma \vdash Q_1 <: Q_2 & \quad \Gamma \vdash S_1 <: S_2 \\
\Gamma \vdash \{Q_1\} S_1 <: \{Q_2\} S_2 & \quad \text{(SUB-QTYPE)} \\
\end{align*}
\]

Fig. 5. Subtyping rules of System $F_{<:Q}$.

Typing for System $F_{<:Q}$

\[
\begin{align*}
\Gamma & \vdash t : T & \quad \text{(VAR)} \\
\Gamma, x : T \vdash t : T & \quad \text{(ABS)} \\
\Gamma, X <: S \vdash t : T & \quad \text{(T-ABS)} \\
\Gamma & \vdash \{Q\} S & \quad \Gamma \vdash Q <: P \\
\Gamma & \vdash \text{assert } P t : \{Q\} S & \quad \text{(TYP-ASSERT)} \\
\end{align*}
\]

Fig. 6. Typing rules for System $F_{<:Q}$

All other rules remain unchanged, except that rules (SUB-ARROW), (SUB-ALL), and (SUB-QALL) are updated to operate on qualified types $T$, instead of simple types $S$, wherever they are changed in the source syntax (see Figure 2) to use qualified types $T$ instead of simple types $S$.

Typing. Finally, Figure 6 defines the typing rules of System $F_{<:Q}$. The typing judgment assigns qualified types $T$ to expressions, and can be viewed as $\Gamma \vdash t : \{Q\} S$. As System $F_{<:Q}$ does not assign an interpretation to qualifiers, the introduction rules for typing values, (ABS), (T-ABS), and (Q-ABS), simply introduce qualifiers by typing values with their tagged qualifier, and the elimination rules remain unmodified. The only (new) elimination rules which deal with qualifiers are the new
Qualifying System $F_{<:Q}$

\[
P, Q, R ::= \begin{array}{c}
\text{Qualifiers in extended System } F_{<:Q} \\
\mid \overline{l} \quad \text{Base lattice elements } l \in L \\
\mid Y \quad \text{Qualifier variables} \\
\mid Q \land R \mid Q \lor R \quad \text{Meets and joins}
\end{array}
\]

\[
C ::= \begin{array}{c}
\text{Concrete Qualifiers} \\
\mid \overline{l} \quad \text{Base lattice elements } l \in L
\end{array}
\]

Fig. 7. The syntax of System $F_{<:Q}$ extended over a bounded lattice $L$. Differences to System $F_{<:Q}$ highlighted in grey.

rules (TYP-ASSERT) and (TYP-UPQUAL), which check that their argument is properly qualified. We additionally add (Q-ABS) and (Q-APP) to support qualifier polymorphism. Besides these changes, the typing rules immediately carry over from System $F_{<:}$.

2.6 Metatheory

System $F_{<:Q}$ satisfies the standard progress and preservation theorems.

**Theorem 2.4 (Preservation).** Suppose $\Gamma \vdash s : T$, and $s \rightarrow t$. Then $\Gamma \vdash t : T$ as well.

**Theorem 2.5 (Progress).** Suppose $\emptyset \vdash s : T$. Then either $s$ is a value, or $s \rightarrow t$ for some term $t$.

While System $F_{<:Q}$ does not place any interpretation on qualifiers outside of upqual and assert, such a system can already be useful. For one, the static type of a value will always be greater than the tag annotated on it and this correspondence is preserved through reduction by preservation. This property can already be used to enforce safety constraints. For example, as Foster et al. [1999] point out, one can use a negative type qualifier sorted to distinguish between sorted and unsorted lists. By default, most lists would be tagged at $\top$, marking them as unsorted lists. A function like merge, though, which merges two sorted lists into a third sorted list, would expect two $\bot$-tagged lists, assert that they are actually $\bot$-tagged, and produce a $\bot$-tagged list as well. While this scheme does not ensure that all $\bot$-tagged lists are sorted, so long as programmers are careful to ensure that they never construct explicitly $\bot$-tagged unsorted lists, they can ensure that functions which expect sorted lists are actually passed sorted lists.

2.7 Generalizing Qualifiers to General Lattices

Qualifiers often come in more complicated lattices: for example, protection rings [Karger and Herbert 1984] induce a countable lattice, and combinations of binary qualifiers induce a product lattice. Now, we show how we can tweak the recipe used to construct System $F_{<:Q}$ for two-point lattices to support general (countable, bounded) qualifier lattices $L$ as well.

**Syntax.** The syntax changes needed to support this construction are listed in Figure 7. Lattice elements are now generalized from $\top$ and $\bot$ to elements $l$ from our base lattice $L$, but as $L$ is bounded, note that we still have distinguished elements $\top$ and $\bot$ in $L$.

**Subqualification.** The subqualification changes needed to support this construction are listed in Figure 8. These are exactly the rules needed to support the free lattice construction over any arbitrary countable bounded lattice. Rule (SQ-LIFT) simply lifts the lattice order $\sqsubseteq$ that $L$ is equipped with up to the free lattice order defined by the subqualification lattice. Rules (SQ-EVAL-ELIM) and
Subqualification for System $F_{<:Q}$ over a lattice $L$

\[
\frac{l_1, l_2 \in L \quad l_1 \sqsubseteq l_2}{\Gamma \vdash l_1 <: l_2} \quad (\text{SQ-LIFT})
\]

\[
\frac{\Gamma \vdash l <: R}{\Gamma \vdash l : T} \quad \Gamma \vdash l = \text{eval} Q' \quad \Gamma \vdash l : R \quad \Gamma \vdash Q <: R \quad \Gamma \vdash Q <: Q' \quad \Gamma \vdash Q' <: R \quad \Gamma \vdash Q <: R \quad (\text{SQ-EVAL-INTRO})
\]

Fig. 8. Extended sub-qualification rules for System $F_{<:Q}$.

(SQ-EVAL-INTRO) are a little more complicated, though, but are necessary in order to relate textual meets and joins of elements of the base lattice $L$, like $l_1 \lor l_2$, to their actual meets and joins in the qualifier lattice, $l_1 \sqcup l_2$. We would expect that these two terms would be equivalent in the subqualification lattice; namely, that $\Gamma \vdash l_1 \lor l_2 <: l_1 \sqcup l_2$ and that $\Gamma \vdash l_1 \sqcup l_2 <: l_1 \lor l_2$. However, without the two evaluation rules (SQ-EVAL-ELIM) and (SQ-EVAL-INTRO), we would only be able to conclude that $\Gamma \vdash l_1 \lor l_2 <: l_1 \sqcup l_2$, but not the other desired inequality $\Gamma \vdash l_1 \sqcup l_2 <: l_1 \lor l_2$.

To discharge this equivalence, (SQ-EVAL-ELIM) and (SQ-EVAL-INTRO) use eval to simplify qualifier expressions. Again, it should not be surprising that this gives rise to the free lattice of extensions of $L$, though we make this precise in the supplementary material.

Soundness. Like simple System $F_{<:Q}$, System $F_{<:Q}$ extended over a bounded lattice $L$ also satisfies the standard soundness theorems:

**Theorem 2.6 (Preservation for Extended System $F_{<:Q}$).** Suppose $\Gamma \vdash s : T$, and $s \longrightarrow t$. Then $\Gamma \vdash t : T$ as well.

**Theorem 2.7 (Progress for Extended System $F_{<:Q}$).** Suppose $\emptyset \vdash s : T$. Then either $s$ is a value, or $s \longrightarrow t$ for some term $t$.

3 APPLICATIONS

Having introduced our design recipe by constructing System $F_{<:Q}$ as a qualified extension of System $F_{<}$, we now study how our subqualification and polymorphism recipe can be reused in three practical qualifier systems. For brevity, we will base our qualifier systems on System $F_{<:Q}$, as it already provides rules and semantics for typing, subqualification and qualifier polymorphism, which we modify below.

While each system has application-specific semantics tied to the interpretations of the qualifiers we are now assigning, all three systems share the same common higher-rank polymorphism and expressiveness at the qualifier level using free lattices as base System $F_{<:Q}$; in essence, expressiveness and polymorphism come for free from base System $F_{<:Q}$.

3.1 Reference Immutability

We start by examining one well-studied qualifier system, that of reference immutability [Huang et al. 2012; Potanin et al. 2013; Tschantz and Ernst 2005]. In this setting, each (heap) reference can be either mutable or immutable. An immutable reference cannot be used to mutate the value or any other values transitively reached from it, so a value read through a readonly-qualified compound object or reference is itself readonly as well.
case class Box[X](var v: X)

def good(x : Box[Int]) = { x.v = 5 }

def bad1(y : readonly Box[Int]) = { y.v = 7 }

def bad2(y : readonly Box[Cell[Int]]) = { y.v.v = 5 }

def access(z: readonly Box[Box[Int]]): readonly Box[Int] = { z.v }

For example, a reference immutability system would deem the function good to be well-typed because it mutates the value of a Box through a mutable reference x. However, it would disallow bad1 because it mutates the box through a read-only reference y. Moreover, it would also disallow bad2 because it mutates the box referenced indirectly through the read-only reference y. This can also be seen by looking at the access function, which returns a read-only reference of type @readonly Box[Int] to the value of the box referenced by z.

Mutable and read-only references can coexist for the same value, so a read-only reference does not itself guarantee that the value will not change through some other, mutable reference. This is in contrast to the stronger guarantee of object immutability, which applies to values, and ensures that a particular value does not change through any of the references to it [Potanin et al. 2013; Zibin et al. 2007]. So, for example, we could create a cell with both a mutable and a readonly reference to it, mutate the cell through the mutable reference, and read the updated value through the readonly reference.

val mutable_ref = Box(10)
val readonly_ref: readonly Box[Int] = mutable_ref

good(mutable_ref)
println(readonly_ref.v) // prints 5

Reference immutability systems have long been studied in various contexts [Dort and Lhoták 2020; Gordon et al. 2012; Huang et al. 2012; Lee and Lhoták 2023; Tschantz and Ernst 2005; Zibin et al. 2007]. Here, we show that we can reuse our recipe to model reference immutability in a setting with higher rank polymorphism and subtyping over both qualifiers and ground types, in a calculus System \( F_{\leq QM} \).

Assigning Qualifiers. We need to define how qualifiers mutable and readonly are assigned to \( \top \) and \( \bot \) in System \( F_{\leq QM} \). Since a mutable reference can always be used where a readonly reference is expected, we assign mutable to \( \bot \) and readonly to \( \top \). This is reflected in Figure 9.

Syntax and Evaluation. Now we need to design syntax and reduction rules for references and immutable references. We add support for references via box forms and we add rules for introducing and eliminating boxes. A box reduces at runtime to some location \( l \) in a store \( \sigma \) that maps locations to values. Reduction now takes place over pairs of terms and stores \( \langle t, \sigma \rangle \):

\[
\langle \text{set-box!} (\text{box} \bot 10), [] \rangle \rightarrow \langle \text{set-box!} (\text{box} \bot 0x0001) 5, [0x0001 : 10] \rangle \\
\rightarrow \langle 10, [0x0001 : 5] \rangle
\]

To distinguish between mutable and immutable references (boxes), we reuse the qualifiers tagged on values. Values with tags \( P \) that eval to \( \bot \) are mutable, whereas values with tags \( P \) that otherwise evaluate to \( \top \) are read-only. Writing to a box requires that it be mutable, or tagged at \( \bot \). So the following the term get stuck.

\[
\langle \text{set-box!} (\text{box} \top 10), [] \rangle \rightarrow \langle \text{set-box!} (\text{box} \top 0x0001) 5, [0x0001 : 10] \rangle \\
\rightarrow \text{gets stuck.}
\]
Additional Evaluation Rules for System $\text{F_{<:QM}}$

\[
\begin{align*}
\langle s, \sigma \rangle & \rightarrow \langle t, \sigma' \rangle \\
\end{align*}
\]

Fig. 9. The syntax of System $\text{F_{<:QM}}$.

Fig. 10. Reduction rules for System $\text{F_{<:QM}}$.

One can explicitly mark a value immutable by upqual-ing to $\top$. The elimination form for reading from a reference, $\text{(deref)}$, ensures that a value read from a reference tagged $\text{readonly}$, or at $\top$, remains $\text{readonly}$. This is reflected in the updated operational semantics (Figure 10).

Typing. We now need to define new typing rules for reference forms and to possibly adjust existing typing rules to account for our new runtime interpretation of qualifiers. For this system, we only need to add typing rules, as shown in Figure 11. To ensure immutability safety, the standard reference update elimination form $\text{(ref-update)}$ is augmented to check that a reference can only be written to if and only if it can be typed as $\text{mutable box}$. Finally, the standard reference read elimination form $\text{(ref-elim)}$ is augmented to enforce that the mutability of the value read from a reference is joined with the mutability of the reference itself to ensure transitive immutability.
Additional Typing and Runtime Typing for System F_{c:<:QM}

\[
\frac{\Gamma \vdash t : T}{\Gamma \vdash \text{box}_P t : \{P\} \text{box } T} \quad \text{(REF-INTRO)}
\]

\[
\frac{\Gamma \vdash \text{unbox } t : \{Q_1 \lor Q_2\} S}{\Gamma \vdash t : T} \quad \text{(REF-ELIM)}
\]

\[
\frac{l : T \in \Sigma}{\Gamma \vdash \text{box}_P l : \{P\} \text{box } T} \quad \text{(RUNTIME-REF-INTRO)}
\]

\[
\frac{\Gamma \vdash \text{set-box! } s t : T}{\Gamma \vdash t : T} \quad \text{(REF-UPDATE)}
\]

\[
\frac{\text{dom}(\sigma) = \text{dom}(\Sigma) \quad \forall l \in \text{dom}(\Sigma), \Gamma \vdash \sigma(l) : \Sigma(l)}{\Gamma \vdash \sigma} \quad \text{(STORE)}
\]

Fig. 11. Typing rules for System F_{c:<:QM}; notable changes highlighted in grey.

safety. Other than qualifiers, our construction is completely standard; we merely add a store \(\sigma\) and a runtime store environment \(\Sigma\) mapping store locations to types.

**Metatheory.** We can prove the standard soundness theorems without any special difficulty:

**Theorem 3.1 (Preservation of System F_{c:<:QM}).** Suppose \(\langle s, \sigma \rangle \rightarrow \langle t, \sigma' \rangle\). If \(\Gamma \vdash \sigma\) and \(\Gamma \vdash s : T\) for some type \(T\), then there is some environment extension \(\Sigma'\) of \(\Sigma\) such that \(\Gamma \vdash \sigma'\) and \(\Gamma \vdash t : T\).

**Theorem 3.2 (Progress for System F_{c:<:QM}).** Suppose \(\emptyset \vdash \sigma\) and \(\emptyset \vdash s : T\). Then either \(s\) is a value or there is some \(t\) and \(\sigma'\) such that \(\langle s, \sigma \rangle \rightarrow \langle t, \sigma' \rangle\).

With only progress and preservation, we can already state something meaningful about the immutability safety of System F_{c:<:QM}; we know that well-typed programs will not get stuck trying to write to a \(\bot\)-tagged reference.

Moreover, the typing rules, in particular (REF-ELIM), give us our desired transitive immutability safety as well; values read from a \(\top\)-tagged value will remain \(\top\)-tagged and therefore read-only as well. Finally, as qualifier tags only affect reduction by blocking reduction (that is, getting stuck), we almost directly recover full immutability safety as well for free, by noting that references typed (by subtyping) at **readonly** can be re-tagged at **readonly** as well without affecting reduction, assuming the original program was well-typed.

### 3.2 Function Colouring

Function colouring [Nystrom 2015] is another qualifier system. In this setting, functions are qualified with a kind that indicates a **colour** for each function, and there are restrictions on which other functions a function can call depending on the colours of the callee and caller. For example, **noexcept** and **throws** form a function colouring system—functions qualified **noexcept** can only call functions qualified **noexcept**. Another instantiation of this problem is the use of the qualifiers **sync** and **async** in asynchronous programming. **async**-qualified functions may call all functions but **sync**-qualified functions may only call other **sync**-qualified functions.

Asynchronous functions are often used in languages like JavaScript to interact with external resources that may take to respond. The program should not **block** waiting on a response. For
example, we may have a function `fetch` which fetches the contents of a web page from a server as a string.

```scala
def fetch(url: String): String = ??? // has type: (String => String) async
```

Function `fetch` is asynchronous as fetching a webpage takes time. So when we call `fetch` in some function, we suspend and give up control flow to other parts of our program until the response is ready, at which point control flow transfers back to our function.

```scala
val review = fetch("https://oopsla24.hotcrp.com/paper/73/")
// sends request to get review (F5!)
// transfers control flow to rest of program.
println(review)
// when review is ready, control flow transfers
// back here and we print it.
```

Polymorphism with function colours is known to be painful [Nystrom 2015]. Consider a higher-order function `map`:

```scala
def map[X, Y](l: List[X], f: (X => Y)) = ???
```

What should its colour be? Well, if we only called `map` with synchronous functions, like `increment`, then it follows that `map` itself can be synchronous, as it performs no operations which can block our program.

```scala
def increment(i: Int) = i + 1
map([1, 2, 3], increment) // returns [2, 3, 4], doesn't block.
```

However, what if we called `map` on `fetch`, for example, to fetch multiple websites?

```scala
val pages = map(follow, fetch) // returns contents of web pages; can block.
```

Here, `map` calls an asynchronous function, namely `fetch`, to fetch a list of web sites. This operation is blocking, so it follows that `map` in this context has to be marked `async` as it performs operations which can block our program. So what is the colour of `map`?

The answer is that the colour of a function like `map` depends on the function `f` it is applying. Without a mechanism to express this dependency, such as colour polymorphism, functions like `map` need to be implemented twice—once for an `async`-qualified `f`, and once for a `sync`-qualified `f`.

```scala
def map[X, Y, Q](l: List[X], f: Q (X => Y)) : Y = ???
```

Moreover, function colouring requires a mechanism for mixing colours! Consider function composition:

```scala
def compose[A, B, C](f: A => B, g: B => C) = (x) => g(f(x))
```

The colour of the result of `compose` needs to be the join of the colours of `f` and `g`. If either `f` or `g` are asynchronous, then the result of `compose` is as well, but if both `f` and `g` are synchronous, then so should be the result of composing them.

```scala
def compose[A, B, C, Q, R](f: Q (A => B), g: R (B => C)): {Q | R} (A => C) = (x) => g(f(x))
```

We now show how our recipe can be used to construct System $F_{<:\text{QA}}$, a calculus that enforces these restrictions.

*Assigning Qualifiers.* Since a synchronous function can be called anywhere that an asynchronous function could be, we assign the $\top$ qualifier to `async` and the $\bot$ qualifier to `sync`.  

Proc. ACM Program. Lang., Vol. 8, No. OOPSLA1, Article 115. Publication date: April 2024.
Fig. 12. The syntax of System $F_{<:QA}$.

Evaluation for System $F_{<:QA}$

\[
\begin{align*}
\langle s(t), \kappa \rangle & \rightarrow \langle s, \text{arg } t :: \kappa \rangle \quad \text{(CONG-APP)} \\
\langle v, \text{arg } t :: \kappa \rangle & \rightarrow \langle t, \text{app } v :: \kappa \rangle \quad \text{(CONG-ARG)} \\
\langle s[S], \kappa \rangle & \rightarrow \langle s, \text{targ } S :: \kappa \rangle \quad \text{(CONG-TAPP)} \\
\langle s[Q], \kappa \rangle & \rightarrow \langle s, \text{qarg } Q :: \kappa \rangle \quad \text{(CONG-QAPP)} \\
\langle v, \text{barrier } C :: \kappa \rangle & \rightarrow \langle v, \kappa \rangle \quad \text{(BREAK-BARRIER)}
\end{align*}
\]

\[
\begin{align*}
C \leq C_i \text{ for all barrier } C_i \text{ frames on } \kappa & \quad \text{eval } P = C \quad \text{(REDUCE-APP)} \\
\langle v, \text{app } \lambda (x).p.t :: \kappa \rangle & \rightarrow \langle t[x \mapsto v], \text{barrier } C :: \kappa \rangle \\
\langle \lambda (X :: S).p.t, \text{targ } S' :: \kappa \rangle & \rightarrow \langle t[X \mapsto S'], \text{barrier } C :: \kappa \rangle \quad \text{(REDUCE-TAPP)} \\
\langle \lambda (Y :: Q).p.t, \text{qarg } Q' :: \kappa \rangle & \rightarrow \langle t[Y \mapsto Q'], \text{barrier } C :: \kappa \rangle \quad \text{(REDUCE-QAPP)}
\end{align*}
\]

Fig. 13. Operational Semantics (CK-style) for System $F_{<:QA}$

**Syntax.** Figure 12 presents the modified syntax of System $F_{<:QA}$. To keep track of the synchronicity that a function term should run in, we reuse the tags already present in values. An example of an asynchronous function term is $\lambda (x)_{\text{async}} \cdot x$, and an example of a function that is polymorphic in its qualifier is $\lambda (Y :: \text{sync})_{\text{async}} \cdot \lambda (f)_Y$. $f(1)$, describing a function that should run in the same synchronicity context as its argument $f$.

**Evaluation.** To model synchronicity safety, Figure 13 describes the operational semantics of System $F_{<:QA}$ using Felleisen and Friedman [1987]-style CK semantics, extended with special barrier frames installed on the stack denoting the colour of the function that was called. When a function is called, we place a barrier with the evaluated colour of the function itself; so a term like

\[
\langle 1, \text{app } \lambda (x)_{\bot}.x :: \kappa \rangle \rightarrow \langle 1, \text{barrier } \bot :: \kappa \rangle
\]

placing a barrier marking a synchronous function on the stack.

Barriers are used to ensure that functions that are called are compatible with the rest of a stack; namely, an asynchronous function can be called only if there are no barriers on the stack marked synchronous. So a call that would place an asynchronous function above a synchronous function on the stack:

\[
\langle (\lambda (x)_T).t, \text{barrier } \bot \rangle \rightarrow \langle \lambda (x)_T.t, \text{arg } v :: \text{barrier } \bot \rangle \\
\rightarrow \langle v, \text{app } \lambda (x)_T.t :: \text{barrier } \bot \rangle \\
\rightarrow \text{gets stuck.}
\]

The other evaluation contexts are standard.
Typing for System $F_{<:QA}$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Pre</th>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A-VAR)</td>
<td>$x : T \in \Gamma$</td>
<td>$\Gamma \vdash x : T$</td>
</tr>
<tr>
<td>(A-ABS)</td>
<td>$\Gamma, x : T_1 \vdash P \vdash t : T_2$</td>
<td>$\Gamma \vdash \text{sync} + \lambda(x)p.t : {P} T_1 \rightarrow T_2$</td>
</tr>
<tr>
<td>(A-T-ABS)</td>
<td>$\Gamma, \text{sync} + \Lambda(X &lt; : S)p.t : {P} \forall(X &lt; : S).T$</td>
<td>$\Gamma, Y &lt; : Q \vdash \text{sync} + \Lambda(Y &lt; : Q)p.t : {P} \forall(Y &lt; : Q).T$</td>
</tr>
<tr>
<td>(A-APP)</td>
<td>$\Gamma \vdash s : T_1 \quad \Gamma \vdash t : {R} T_1 \rightarrow T_2$</td>
<td>$\Gamma \vdash R \vdash t(s) : T_2$</td>
</tr>
<tr>
<td>(A-T-APP)</td>
<td>$\Gamma \vdash R \vdash t : {R} \forall(X &lt; : S).T$</td>
<td>$\Gamma \vdash R \vdash t[S'] : T[X \mapsto S']$</td>
</tr>
<tr>
<td>(A-Q-APP)</td>
<td>$\Gamma \vdash R \vdash t : {R} \forall(Y &lt; : Q).T$</td>
<td>$\Gamma \vdash R \vdash t{Q'} : T[Y \mapsto Q']$</td>
</tr>
<tr>
<td>(A-SUB)</td>
<td>$\Gamma \vdash R \vdash s : T_1 \quad \Gamma \vdash t_1 &lt; : T_2$</td>
<td>$\Gamma \vdash R \vdash t_1 &lt; : T_2$</td>
</tr>
<tr>
<td>(A-SUB-EFF)</td>
<td>$\Gamma \vdash R \vdash s : T_1 \quad \Gamma \vdash t \vdash s : T_2$</td>
<td>$\Gamma \vdash Q \vdash s : T_2$</td>
</tr>
</tbody>
</table>

Fig. 14. Typing rules for System $F_{<:QA}$

Typing. To guarantee soundness, Figure 14 endows the typing rules of System $F_{<:QA}$ with modified rules for keeping track of the synchronicity context that a function needs. We extend the typing rules with a colour context $R$ to keep track of the synchronicity of the functions being called. This colour context $R$ is simply a qualifier expression, and is introduced by the introduction rules for typing abstractions by lifting the qualifier tagged on those abstractions – see rules (A-ABS), (A-T-ABS), and (A-Q-ABS). As creating an abstraction is effect free, the introduction forms (A-ABS), (A-T-ABS), and (A-Q-ABS) can run in any colour context, in particular, at sync or ⊥.

To ensure safety when applying functions in the elimination form (A-APP), we check that the colour context is compatible with the type of the function being called; subsumption in (A-SUB-EFF) allows functions to run if the qualifiers do not exactly match but when the qualifier on the function is subqualified by the colour context. The typing rules outside of manipulating the context $R$ remain otherwise unchanged.

Metatheory. With all this, we can state and prove progress and preservation for System $F_{<:QA}$.

**Theorem 3.3 (Progress of System $F_{<:QA}$).** Suppose $\langle c, \kappa \rangle$ is a well-typed machine configuration. Then either $c$ is a value and $\kappa$ is the empty continuation, or there is a machine state $\langle c', \kappa' \rangle$ that it steps to.

**Theorem 3.4 (Preservation of System $F_{<:QA}$).** Suppose $\langle c, \kappa \rangle$ is a well-typed machine configuration. Then if it steps to another configuration $\langle c', \kappa' \rangle$, that configuration is also well-typed.

Note that progress and preservation guarantee meaningful safety properties about System $F_{<:QA}$: namely that an asynchronous function is never called above a synchronous function during evaluation, as such a call would get stuck, by (REDUCE-APP).

**Observations.** System $F_{<:QA}$ can be used to model function colouring with other qualifiers as well; for example, we could model colours noexcept and throws by assigning noexcept to $\bot$ and
throws to $T_1$; (REDUCE-APP) would ensure that a function which could throw cannot be called if any function on the call stack is qualified at noexcept. More interestingly, System $F_{<:0a}$ could be viewed as a simple effect system; the synchronicity context $R$ can be seen as the effect of a term! We discuss this curious connection between qualifiers and effects in Section 7.3.

### 3.3 Tracking Capture

Finally, our design recipe can be remixed to construct a qualifier system to qualify values based on what they capture. Some base values are meaningful and should be tracked, and other values are forgettable.

**Motivation.** One application of such a system is the effects-as-capabilities discipline [Dennis and Van Horn 1966], which enables reasoning about which code can perform side effects by simply tracking capabilities, special values that grant the holder the ability to perform side effects, such as the ability to perform I/O or the ability to throw an exception.

**What to track?** Suppose, for example, that we have a base capability named pandora, which allows its holder to produce arbitrary values. Such a precious value really ought to be tracked and not forgotten, as in the hands of the wrong user, it can perform dangerous side effects!

```java
val pandora : {tracked} [A] (Unit => A) = ???
```

However, it is not only pandora itself that is dangerous. Actors that capture pandora can themselves cause dangerous side effects. For example, some values should never be generated [Aaronson 2002]:

```java
def takeOverTheWorld(): Unit = {
    val powerful_proof = pandora[P.equals_NP_proof]()
    powerful_proof.use()
} // pandora is captured by takeOverTheWorld.
```

In general, values that capture meaningful values—capabilities—become meaningful themselves, since they can perform side effects, so they should also be tracked. Now, while it is clear that pandora and takeOverTheWorld are both dangerous, they are dangerous for different reasons: pandora because it intrinsically is and takeOverTheWorld because it captures pandora.

**Distinguishing Capabilities.** In practical applications, we may wish to distinguish between different effects, modelled by different capabilities. For example, we may wish to reason about a more pedestrian side effect – printing – separately from the great evil that pandora can perform. It is reasonable to expect that we can print in more contexts than we can use the pandora.

```java
val print : {tracked} String => Unit = ???
```

```java
def helloWorld() = print("Hello␣World!") // tracked as it captures print
```

```java
def runCodeThatCanPrint(f: ??? () => Unit) = f()
runCodeThatCanPrint(helloWorld) // OK
runCodeThatCanPrint(takeOverTheWorld) // Should be forbidden
```

In this example, function runCodeThatCanPrint only accepts thunks that print as a side effect. What type annotation should we give to its argument $f$? In particular, what qualifier should we use to fill in the blank? It should not be tracked, as otherwise we could pass takeOverTheWorld to runCodeThatCanPrint – an operation which should be disallowed. Instead we would like to fill that blank with print; to denote that runCodeThatCanPrint can accept any thunk which is no more dangerous than print itself. Figure 15 summarizes the different variables in the above examples and the qualifiers we would like to assign to their types.

As Boruch-Gruszecki et al. [2023]; Odersky et al. [2021] show, such a capture tracking system could be used to guarantee desirable and important safety invariants. They model capture tracking
<table>
<thead>
<tr>
<th>Term</th>
<th>Qualifier</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandora</td>
<td>tracked</td>
<td>As pandora is a base capability.</td>
</tr>
<tr>
<td>print</td>
<td>tracked</td>
<td>As print is a base capability.</td>
</tr>
<tr>
<td>takeOverTheWorld</td>
<td>pandora</td>
<td>As takeOverTheWorld is no more dangerous than pandora.</td>
</tr>
<tr>
<td>helloWorld</td>
<td>print</td>
<td>As helloWorld is no more dangerous than print.</td>
</tr>
</tbody>
</table>

Fig. 15. Qualifier assignments in Capture Tracking

\[
s, t ::= \\
\ldots \\
| s\{Q\}(t) \quad \text{term application} \\
S ::= \\
\ldots \\
| (x : T_1) \rightarrow T_2 \quad \text{function type} \\
P, Q, R ::= \\
\ldots \\
| x \quad \text{term variables} \\
| \text{tracked (as } \top) \quad \text{tracked values}
\]

\[
\text{Evaluation: } s \rightarrow t
\]

\[
(\lambda(x)p.t)\{Q\}(s) \rightarrow (\text{C-BETA-V}) \\
f[x \mapsto \text{type } Q][x \mapsto \text{term } s] \rightarrow (\text{C-BETA-V})
\]

\[
\text{Subqualification: } \Gamma \vdash Q <: R
\]

\[
x : \{Q\} S \in \Gamma \quad \Gamma \vdash Q <: R \\
\Gamma \vdash x <: R \quad (\text{SQ-TVAR})
\]

\[
x : \{Q\} S \in \Gamma \quad \Gamma \vdash x <: x \quad (\text{SQ-REFL-TVAR})
\]

\[
\text{Subtyping: } \\
\Gamma \vdash S_1 <: S_2 \\
\Gamma \vdash T_2 <: T_1 \\
\Gamma \vdash x : T_2 \vdash T_3 <: T_4 \\
\Gamma \vdash (x : T_2) \rightarrow T_3 <: (x : T_1) \rightarrow T_4 \quad (\text{C-SUB-ARROW})
\]

\[
\text{Fig. 16. Evaluation, Syntax, Subtyping for System } F_{<;QC}
\]

using sets of variables, but a set is just a lattice join of the singletons in that set! For example, Boruch-Gruszecki et al. [2023] would give the following evil_monologue\(^2\) function the capture set annotation \{takeOverTheWorld, print\}, while we would give it the qualifier annotation \{takeOverTheWorld | print\}.

\[
def evil_monologue(): Unit = \\
\quad \text{print("I expect you to die in polynomial time, Mr. Bond.")}
\quad \text{takeOverTheWorld()}
\]

Using this insight, we can model capture tracking as an extension System \(F_{<;QC}\) of System \(F_{<;Q}\).

**Assigning Qualifiers.** We attach a qualifier tracked to types, denoting which values we should keep track of. The qualifier tracked induces a two-point lattice, where tracked is at \(\top\), and values that should not be tracked, or should be forgotten, are qualified at \(\bot\). Base capabilities will be given the tracked qualifier.

\(^2\)Scene from James Bond: The Travelling Salesman.
Qualifying System $F_{<\cdot QC}$

Typing for System $F_{<\cdot QC}$

\[
\Gamma \vdash t : T
\]

\[
\Gamma, x : T_1 \vdash t : T_2 \quad \Gamma \vdash \forall y \in \text{fv}(t) \cdot x \ y < : P
\]

\[
\Gamma \vdash \lambda(x)_{P} \cdot t : \{P\} (x : T_1) \rightarrow T_2 \quad (C\text{-ABS})
\]

\[
\Gamma, X <: S \vdash t : T \quad \Gamma \vdash \forall y \in \text{fv}(t) \ y < : P
\]

\[
\Gamma \vdash \lambda(X <: S)_{P} \cdot t : \{P\} \forall(X <: S)T \quad (C\text{-T-ABS})
\]

\[
\Gamma, X <: S \vdash t : T \quad \Gamma \vdash \forall y \in \text{fv}(t) \ y < : P
\]

\[
\Gamma \vdash \lambda(Y <: Q)_{P} \cdot t : \{P\} \forall(Y <: Q)T \quad (C\text{-Q-ABS})
\]

Fig. 17. Typing rules for System $F_{<\cdot QC}$

Syntax – Tracking Variables. Figure 16 defines the syntax of System $F_{<\cdot QC}$. To reflect the underlying term-variable-based nature of capture tracking, term bindings in System $F_{<\cdot QC}$ introduce both a term variable in term position as well as a qualifier variable in qualifier position with the same name as the term variable.

Term bindings now serve double duty introducing both term variables and qualifier variables, so a term like the identity function $\lambda(x) \cdot x$ would be given the type $\{\bot\} \cdot (x : \{Q\} S) \rightarrow \{x\} S$ to indicate that it is not tracked but the result might be tracked depending on whether or not its argument $x$ is tracked as well. This still induces a free lattice structure generated over the two-point lattice that tracked induces, except in this case, the free lattice includes both qualifier variables introduced by qualifier binders in addition to qualifier variables introduced by term binders as well. As term binders introduce both a term and qualifier variable, term application in System $F_{<\cdot QC}$ now requires a qualifier argument to be substituted for that variable in qualifier position. As such, term application in System $F_{<\cdot QC}$ now has three arguments $s\{Q\}(t)$ – a function $s$, a qualifier $Q$, and an argument $t$; see Figure 16. In this sense, term abstractions in System $F_{<\cdot QC}$ can be viewed as a combination of a qualifier abstraction $\Lambda[x <: Q]$ followed by a term abstraction $\lambda(x : \{x\} T)$.

Subqualification. One essential change is that we need to adjust subqualification to account for qualifier variables bound by term binders in addition to qualifier variables bound by qualifier binders. These changes are the addition of two new rules, $(SQ\text{-REFL-TVAR})$ and $(SQ\text{-TVAR})$. Rule $(SQ\text{-REFL-TVAR})$ accounts for reflexivity in System $F_{<\cdot QC}$’s adjusted subqualification judgment. $(SQ\text{-TVAR})$ accounts for subqualification for qualifier variables bound by term binders, and formalizes this notion of less dangerous we discussed earlier—that takeOverTheWorld can be used in a context that allows the use of pandora, and that helloWorld can be used in a context that allows the use of print. Interestingly, though, if we squint at $(SQ\text{-TVAR})$ carefully, glossing over the text in faint gray, we observe that it is just a close duplicate of the existing subqualification rule for qualifier variables, $(SQ\text{-VAR})$!

\[
\text{takeOverTheWorld : pandora Unit} \Rightarrow \text{Unit} \in \Gamma \quad \Gamma \vdash \text{pandora} <: \text{pandora}
\]

\[
\Gamma \vdash \text{takeOverTheWorld} <: \text{pandora}
\]
Subtyping. As function binders introduce a qualifier variable, so do function types as well; for example, \( x : \{Q\} S \rightarrow \{x\} S \). Subtyping needs to account for this bound qualifier variable; see (C-SUB-ARROW).

Typing. Values are now qualified with the free variables that they close over (i.e., that they capture). To ensure this is faithfully reflected in the value itself, we check that the tag on the value super-qualifies the free variables that value captures. This is reflected in the modified typing rules for typing abstractions: (C-ABS), (C-T-ABS), and (C-Q-ABS). The only other apparent changes are in the rules for term application typing and variable typing. While those rules look different, they reflect how term abstractions are a combination of qualifier and term abstractions, and in that setting are no different than the standard rules for typing term variables, term application, and qualifier application! These changes to the typing rules are reflected in Figure 17.

Soundness. Again, we can prove the standard soundness theorems for System \( \text{F}_{<:Q} \), using similar techniques as Lee et al. [2023].

Theorem 3.5 (Preservation for System \( \text{F}_{<:Q} \)). Suppose \( \Gamma \vdash s : T \), and \( s \rightarrow t \). Then \( \Gamma \vdash t : T \) as well.

Theorem 3.6 (Progress for System \( \text{F}_{<:Q} \)). Suppose \( \emptyset \vdash s : T \). Either \( s \) is a value, or \( s \rightarrow t \) for some term \( t \).

In addition, we recover a prediction lemma [Boruch-Gruszecki et al. 2023; Odersky et al. 2021] relating how the free variables of values relate to the qualifier annotated on their types; in essence, that the qualifier given on the type contains the free variables present in the value \( v \).

Lemma 3.7 (Capture Prediction for System \( \text{F}_{<:Q} \)). Let \( \Gamma \) be an environment and \( v \) be a value such that \( \Gamma \vdash v : \{Q\} S \). Then \( \Gamma \vdash \left\{ \bigvee_{y \in \text{fv}(v)} y \right\} <: Q \).

4 MECHANIZATION

The mechanization of System \( \text{F}_{<:Q} \) (from Section 2.3), its derived calculi, System \( \text{F}_{<:QM} \), System \( \text{F}_{<:QA} \), and System \( \text{F}_{<:QC} \) (from Section 3), and extended System \( \text{F}_{<:Q} \) (from Section 2.7), is derived from the mechanization of System \( \text{F}_< \) by Aydemir et al. [2008], with some inspiration taken from the mechanization of Lee et al. [2023] and Lee and Lhoták [2023]. All lemmas and theorems stated in this paper regarding these calculi have been formally mechanized, though our proofs relating the subqualification structure to free lattices are only proven in text, as we have found Coq’s tooling for universal algebra lacking. Additionally, we give a mechanized proof of the crux of Theorem 2.3; namely, that \( \leq \) is reflexive and transitive, though we note that Galatos [2023] independently give an direct, algebraic proof of this result as well.

5 TYPE POLYMORPHISM AND QUALIFIER POLYMORPHISM

We chose to model polymorphism separately for qualifiers and simple types. We introduced a third binder, qualifier abstraction, for enabling polymorphism over type qualifiers, orthogonal to simple type polymorphism. An alternate approach one could take to design a language which needs to model polymorphism over type qualifiers is to have type variables range over qualified types, that is, types like \texttt{mutable Box[Int]} as well as \texttt{const Box[Int]}. This approach can been seen in systems like Lee and Lhoták [2023]; Tschantz and Ernst [2005]; Zibin et al. [2010]. However, it also comes with its difficulties: how do we formally interpret repeated applications of type qualifiers, for example, with a generic \texttt{inplace_map} which maps a function over a reference cell?

```scala
case class Box[X](var elem: X)
// Is this well formed?
```

Proc. ACM Program. Lang., Vol. 8, No. OOPSLA1, Article 115. Publication date: April 2024.
What should it mean if `inplace_map` is applied on a `Box[const Box[Int]]`? Then `inplace_map` would expect a function `f` with type `(const (const Box[Int])) => const Box[Int]`. While our intuition would tell us that `const (const Box[Int])` is really just a `const Box[Int]`, discharging this equivalence in a proof is not so clear. Many systems, like those of Zibin et al. [2007] and Tschantz and Ernst [2005], sidestep this issue by explicitly preventing type variables from being further qualified, but this approach prevents functions like `inplace_map` from being expressed at all. Another approach, taken by Lee and Lhoták [2023], is to show that these equivalences can be discharged through subtyping rules which normalize equivalent types. However, their approach led to complexities in their proof of soundness and it is unclear if their system admits algorithmic subtyping rules.

Our proposed approach, while verbose, avoids all these complexities by explicitly keeping simple type polymorphism separate from type qualifier polymorphism. We would write `inplace_map` as:

```scala
case class Box[Q, X](var elem: Q X)

def inplace_map[X](r: mutable Box[X], f: const X => X): Unit = {
  r.elem = f(r.elem);
}
```

Moreover, we can desugar qualified type polymorphism into a combination of simple type polymorphism and type qualifier polymorphism. We can treat a qualified type binder in surface syntax as a pair of simple type and type qualifier binders, and have qualified type variables play double duty as simple type variables and type qualifier variables, as seen in qualifier systems like Wei et al. [2024]. So our original version of `inplace_map` could desugar as follows:

```scala
// original
def inplace_map[X](r: mutable Box[X], f: const X => X): Unit = {
  r.elem = f(r.elem);
}
// desugared =
// X splits into Xq and Xs
```

One problem remains for the language designer, however: how do type qualifiers interact with qualified type variables? In our above example, we chose to have the new qualifier annotation `const` strip away any existing type qualifier on `X`; this is the approach that the Checker Framework takes [Papi et al. 2008]. Alternatively, we could instead merge the qualifiers together:

```scala
// original
def inplace_map[Xq, Xs](r: mutable Box[{Xq} Xs], f: const Xs => Xs): Unit = {
  r.elem = f(r.elem);
}
// desugared =
// X splits into Xq and Xs
```

### 6 REVISITING QUALIFIER SYSTEMS

Free lattices have been known by mathematicians since Whitman’s time as the proper algebraic structure for modelling lattice inequalities involving formulas with variables—word problems—over arbitrary lattices. In this light, it is somewhat surprising that existing qualifier systems have not taken advantage of that structure explicitly, especially so given that it is folklore knowledge in the literature that intersection and union types make the subtyping lattice a free lattice, as Dolan
observed. Here, we revisit some existing qualifier systems to examine how their qualifier structure compares to the structure we present with the free lattice of qualifiers.

A Theory of Type Qualifiers. The original work of Foster et al. [1999] introduced the notion of type qualifiers and gave a system for ML-style let polymorphism using a variant of HM(X) constraint-based type inference [Odersky et al. 1999]. Qualifier-polymorphic types in Foster’s polymorphic qualifier system are a type scheme \( \forall Y/C.T \) for some vector of qualifier variables \( Y \) used in qualified type \( T \) modulo qualifier ordering constraints in \( C \), such as \( Y_1 < Y_2 \). However, in their system, constraints cannot involve formulas with qualifier variables (e.g., \( X < Y_1 \land Y_2 \) is an invalid constraint), nor are constraints expressible in their source syntax for qualifier-polymorphic function terms.

Qualifiers for Tracking Capture and Reachability. Our subqualification system was inspired by the subcapturing system pioneered by Boruch-Gruszecki et al. [2023] for use in their capability tracking system for Scala. They model sets of free variables coupled with operations for merging sets together. Sets of variables are exactly joins of variables – the set \( \{a, b, c\} \) can be viewed as the lattice formula \( a \lor b \lor c \), and their set-merge substitution operator \( \{a, b, c\}[a \mapsto \{d, e\}] = \{d, e, b, c\} \), is just substitution for free lattice formulas – \( (a \lor b \lor c)[a \mapsto (d \lor e)] = (d \lor e) \lor b \lor c \). With this translation in mind, we can see that they model a free (join)-semilattice, and that their subcapturing rules involving variables in sets are just translating what the lattice join would be into a set framework.

Independently, Wei et al. [2024] building off of Bao et al. [2021] recently developed a qualifier system for tracking reachability using variable sets as well. Like Boruch-Gruszecki et al. [2023], their subqualification system models a free join-semilattice, with one additional wrinkle. They model a notion of set overlap respecting their subcapturing system as well as a notion of freshness in their framework to ensure that the set of values reachable from a function are disjoint, or fresh, from the set of values reachable from that function’s argument. While overlap exists only at the metatheoretic level and does not exist in the qualifier annotations, it can be seen that their notion of overlap is exactly what the lattice meet of their set-qualifiers would be when interpreted as lattice terms. Additionally, while freshness unfortunately does not fit in the framework of a free lattice, we conjecture that freshness can be modelled in a setting where lattices are extended with complementation as well, such as in free complemented distributive lattices. They are currently working on extending their system to work over free join-semilattice terms though.\(^3\)

Boolean Formulas as Qualifiers. Madsen and van de Pol [2021] recently investigated modelling nullability as a type qualifier. Types in their system comprise a scheme of type variables \( \alpha \) and Boolean variables \( \beta \) over a pair of simple type \( S \) and Boolean formula \( (S, \phi) \), where values of a qualified type \( (S, \phi) \) are nullable if and only if \( \phi \) evaluates to \( \text{true} \).\(^4\) Boolean formulas form a Boolean algebra, and Boolean algebras are just complemented distributive lattices, so Boolean formulas over a set of variables \( \beta \) are just free complemented distributive lattices generated over variables in \( \beta \). In this sense, we can view Madsen and van de Pol [2021] as an ML-polymorphism style extension of Foster et al. [1999] that solves the problem of encoding qualifier constraints: one can just encode them using Boolean formulas.

Reference Immutability for C# [Gordon et al. 2012]. Of existing qualifier systems, the polymorphism structure of Gordon et al. [2012] is closest to System \( \text{F}_{\leq, g} \). Polymorphism is possible over both mutability qualifiers and simple types in Gordon’s system, but must be done separately, as in System


\(^4\)Technically they model a triple \( (S, \phi, \gamma) \) where \( \gamma \) is another Boolean formula which evaluates to \( \text{true} \) if values of type \( (S, \phi, \gamma) \) are non-nullable.
Qualifying System $F_{<:Q}$: The `inplace_map` function that we discussed earlier would be expressed with both a simple type variable as well as with a qualifier variable:

```python
def inplace_map\[Q, X\](r: mutable Box[[Q] X], f: readonly X => {Q} X): Unit
```

Gordon’s system also allows for mutability qualifiers to be merged using an operator `~>`. For example, a polymorphic read function `read` could be written as the following in Gordon’s system:

```python
def read\[QR, QX, X\](r: {QR} Box[[QX] X]): {QR ~> QX} X = r.f
```

Now, `~>` acts as a restricted lattice join. Given two concrete mutability qualifiers $C$ and $D$, $C ~> D$ will reduce to the lattice join of $C$ and $D$. However, the only allowable judgment in Gordon’s system for `~>` when qualifier variables are present, say $C ~> Y$, is that it can be widened to `readonly`.

Reference Immutability for DOT [Dort and Lhoták 2020]. roDOT extends the calculus of Dependent Object Types [Amin et al. 2016] with support for reference immutability. In their system, immutability constraints are expressed through a type member field $x.M$ of each object, where $x$ is mutable if and only if $M \leq \bot$, and $x$ is read-only if and only if $M \geq \top$. As $M$ is just a Scala type member, $M$ can consist of anything a Scala type could consist of, but typically it consists of type meets and type joins of $\top$, $\bot$, type variables $Y$, and the mutability members $y.M$ of other Scala objects $y$.

While this may seem odd, we can view $M$ as a type qualifier member field of its containing object $x$; the meets and joins in roDOT’s subtyping lattice for $M$ correspond to meets and joins in System $F_{<:Q}$’s subqualification lattice. In this sense, we can view type polymorphism in roDOT as a combination of polymorphism over simple types and type qualifiers in System $F_{<:Q}$. A type $T$ in roDOT breaks down into a pair of a simple type $T \setminus M$ – $T$ without its mutability member $M$, and $M$ itself. In this sense, Dort and Lhoták [2020] provide a different method to encode subqualification; they encode it in type members $M$ and reuse the subtyping lattice to encode the free lattice structure needed to deal with qualifier polymorphism and qualifier variables.

Qualifiers as Types. A similar strategy for encoding the free lattice structure of qualifiers in the subtyping lattice can also be seen in Osvald et al. [2016]; Xhebraj et al. [2022]; Zhao [2023]. Instead of encoding the type as a object member, they instead encode it using a combination of generic type parameters and Scala annotations on types. Concretely, for an object $y$ with type $T$, instead of using $y.M$ to encode the locality/mutability of an object $y$, they instead annotate $y$’s type $T$ with a Scala annotation $T ~\text{@local/@mut}[M]$ parameterized by type $M$ to denote that $y$ has locality/mutability $M$.

7 RELATED AND FUTURE WORK

7.1 Languages with Type Qualifier Systems

Rust. The Rust community is currently investigating approaches [Wuyts et al. 2022] for adding qualifiers to Rust. Their current proposal is to generalize the notion of qualified types from being a pair of one qualifier and base type to be a tuple of qualifiers coupled to a base type. Qualifier abstractions are keyed with the kind of qualifier (const, async, etc, ...) they abstract over.

For example, the following is a function `read_to_string` that is polymorphic in the synchronicity of its reader argument; `async<A>` binds the synchronicity qualifier argument $A$ in addition to annotating the type of `read_to_string` with that synchronicity $A$.

```rust
async<A> fn read_to_string\(reader: \&mut impl \text{Read} \times A\) \rightarrow \text{std::io::Result\{String}}
```
This is easy to see sound using similar ideas to our proof of simplified System $F_{<:Q}$, and avoids the complications around subqualification that free lattices over arbitrary lattices pose. One would extend simple System $F_{<:Q}$ with a binder for each qualifier category (like `async` or `noexcept`) instead of using extended System $F_{<:Q}$ with the product lattice over the lattices for each qualifier category. However this proposal has proven controversial in the Rust community due the additional syntactic complexity it imposes.

**OCaml.** The OCaml community [Slater 2023a,b] is investigating adding *modes* to types for tracking properties like *uniqueness*, *locality*, and *linearity*, amongst others; these modes are essentially type qualifiers. They aim to leverage these modes to prevent safety issues from arising from data races in multithreaded OCaml code.

**Pony.** Pony’s *reference capabilities* [Clebsch et al. 2015] are essentially type qualifiers on base types that qualify how values may be shared or used. Pony has qualifiers for various forms of uniqueness, linearity, and ownership properties. While Pony has bounded polymorphism over *qualified types*, Pony does not allow type variables to be requalified, nor does it have polymorphism over qualifiers.

7.2 Implementing Type Qualifiers

The Checker Framework [Dietl et al. 2011; Papi et al. 2008] is an extensible framework for adding user-defined type qualifiers to Java’s type system. The Checker Framework generally allows for qualifying type variables with qualifiers, but in their system, there is no relationship between a type variable $X$ and a qualified type variable $Q \times X$. Re-qualifying a type variable strips any existing conflicting qualifier from that type variable and what it is instantiated with. The Checker Framework has also been used to model effect systems as well: [Gordon et al. 2013].

7.3 Effect Systems

Effect systems are closely related to type qualifiers. Traditionally, effect annotations are used to describe properties of *computation*, whereas type qualifiers are used to describe properties of *data*. In the presence of first-class functions, this distinction is often blurred; for example, modern C++ refers to `noexcept` as a type qualifier on function types [Maurer 2015], whereas traditionally it would be viewed as an effect annotation. In contrast to type qualifiers, both effect polymorphism [Lucassen and Gifford 1988] and the lattice structure of effects [Rytz et al. 2012] are well-studied. However, the interaction of effect polymorphism with subtyping and sub-effecting remains understudied.

Many effect systems use *row polymorphism* to handle polymorphic effect variables with a restricted form of sub-effecting by subsets [Leijen 2014]. As for Rytz et al. [2012], they present a lightweight framework with no effect variables. Formal systems studying sub-effecting respecting *effect bounds* on *effect variables* remain rare, despite Java’s exception system being just that [Gosling et al. 2014, Section 8.4.8.3]. Curiously, the two extant formal effect systems with these features share much in common with well-known qualifier systems. For example, the sub-effecting system Leijen and Tate [2010] can be viewed as a variant of the lattice-based subqualification system of Foster et al. [1999] with HM(X)-style polymorphism. More interestingly, the novel Indirect-Call rule of Gariano et al. [2019], the reachability rule of Wei et al. [2024], and the subcapturing rule of Boruch-Gruszecki et al. [2023] all model subqualification in a free join-semilattice (of effects). In light of all these similarities, and of recent work modelling effect systems with Boolean formulas [Lutze et al. 2023], we conjecture that a system modelling free distributive complemented lattices could be used to present a unifying treatment of both effects and qualifiers in the presence of subtyping, subeffecting, and subqualification.
7.4 Boolean Algebras and Subtyping
The work of Madsen and van de Pol [2021] on Boolean formula qualifier systems does not model subtyping over qualified types \((S, \phi)\); it would be sensible to say \((S, \phi) <: (S, \phi')\) if \(\phi \Rightarrow \phi'\). They conjecture that such a subtyping system would be sound. While we cannot answer this conjecture definitively, as we only model free lattices, not free complemented distributive lattice systems, it would be interesting future work to extend our framework and theirs to see if a system modelling free complemented distributive lattice systems with subqualification is sound.

7.5 Algorithmic Subtyping
System \(\text{F}_{<:Q}\)’s subtyping rules are syntax-directed and admit algorithmic rules, but it is not so easy to see if extended System \(\text{F}_{<:Q}\) admits algorithmic subtyping rules. The difficulty is that extended System \(\text{F}_{<:Q}\) needs two new rules \((\text{sq-eval-elim})\) and \((\text{sq-eval-intro})\) to handle transitivity through base lattice elements, and these rules are not syntax-directed. It remains an open question whether extended System \(\text{F}_{<:Q}\) admits algorithmic subtyping rules. We conjecture that algorithmic subtyping rules could exist for a particular instantiation of extended System \(\text{F}_{<:Q}\) to a fixed base qualifier lattice \(\mathcal{L}\). Moreover, we think that whether or not algorithmic subtyping rules would exist could depend on certain algebraic properties of \(\mathcal{L}\). For example, if \(\mathcal{L}\) is a product lattice for which algorithmic subtyping rules could be written for each lattice in the product, then algorithmic subtyping rules can be written for \(\mathcal{L}\) as well.

7.6 Flow Sensitivity
Foster et al. [2002] extended the original work of Foster et al. [1999] with support for flow sensitivity on type qualifiers. Even though flow sensitivity can be sometimes avoided, for example, with pattern matching as Madsen and van de Pol [2021] show with \texttt{nullable} as a qualifier, flow sensitivity is a natural addition to many qualifier systems like \texttt{nullable}. One often checks if a variable \(x\) is \texttt{NULL} with an if statement, with \(x\) qualified \texttt{nullable} in the branch that fails the test and \texttt{nonnull} in the branch that passes. We conjecture that the ideas that Foster et al. [2002] use to extend their system to support flow sensitivity can also be used to add flow sensitivity to System \(\text{F}_{<:Q}\). It would also be interesting to investigate the underlying algebraic structure of the resulting system, especially in light of recent work on flow sensitive effect systems by Gordon [2021].

8 CONCLUSION
In this paper, we presented a recipe for modelling higher-rank polymorphism, subtyping, and subqualification in systems with type qualifiers by using the free lattice generated from an underlying qualifier lattice. We show how a base calculus like System \(\text{F}_{<:}\) can be extended using this structure by constructing such an extension System \(\text{F}_{<:Q}\), and we show how the recipe can be applied to model three problems where type qualifiers are naturally suited—reference immutability, function colouring, and capture tracking. We then re-examine existing qualifier systems to look at how free lattices of qualifiers show up, even if only indirectly or in restricted form. We hope that this work advances our understanding of the structure of polymorphism over type qualifiers.

ACKNOWLEDGMENTS
We thank Brad Lushman, John Boyland, Guannan Wei, Brian Zimmerman, James Noble, and the anonymous OOPSLA reviewers for their useful feedback in reading over early drafts of this work. We also thank Ross Willard for his useful insights into free lattices. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada and by an Ontario Graduate Scholarship.
DATA-AVAILABILITY STATEMENT

The artifact that supports this paper is available on Software Heritage [Lee et al. 2024a] and on the ACM Digital Library [Lee et al. 2024b].

REFERENCES


Proc. ACM Program. Lang., Vol. 8, No. OOPSLA1, Article 115. Publication date: April 2024.
†19, ‡22


Bjarne Stroustrup. 2007. The C++ programming language - special edition (3. ed.). Addison-Wesley. ‡2


Received 20-10-2023; accepted 2024-02-24

Proc. ACM Program. Lang., Vol. 8, No. OOPSLA1, Article 115. Publication date: April 2024.