Qualifying System $\mathsf{F}_{<:}$

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, Eberhard Karls Universität 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 $\mathsf{F}_{<:\mathsf{Q}}$, which combines the higher-rank bounded polymorphism of System $\mathsf{F}_{<:}$ with the theory of type qualifiers. We explore how the ideas used to construct System $\mathsf{F}_{<:\mathsf{Q}}$ 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 $\mathsf{F}_{<:\mathsf{Q}}$.

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

Additional Key Words and Phrases: System $\mathsf{F}_{<:}$, Type Qualifiers, Type Systems

ACM Reference Format:
Edward Lee, Yaoyu Zhao, Ondřej Lhoták, James You, Kavin Satheeskumar, and Jonathan Immanuel Brachthäuser. 2024. Qualifying System $\mathsf{F}_{<:}$: Some Terms and Conditions May Apply. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 115 (April 2024), 30 pages. https://doi.org/10.1145/3649832

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, Eberhard Karls Universität Tübingen, Tübingen, BaWü, Germany, jonathan.brachthaeuser@uni-tuebingen.de.

ACM 2475-1421/2024/4-ART115
https://doi.org/10.1145/3649832

There are at least two ways to address this. We can view the modification of `toLowerCase`'s argument `in` 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.

```
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.

```
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 `String`s. 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 String`s, which should also accept `mutable String`s as well.

```
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 2023; 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 [Markstrum 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 const to arrive at the monomorphic but equally general version of toLowerCase from above.

```
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?

```
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.*

```
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.

```
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.

```
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.

```
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:

```
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 => 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.

```
def compose[A,B,C,Qf,Qg](f: Qf (A => B), g: Qg (B => C))): {Qf | Qg} (A => C)
```

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 $\mathsf{F}_{<:\mathsf{Q}}$ of System $\mathsf{F}_{<:}$ [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 $\mathsf{F}_{<:\mathsf{Q}}$ 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 $\mathsf{F}_{<:\mathsf{Q}}$, 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 $\mathsf{F}_{<:\mathsf{Q}}$ (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 `const`, which qualifies an existing type to be read-only. It comes equipped with a dual qualifier `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.
`noexcept <: 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 `noexcept` 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 `noexcept` are *negative*, or *contravariant*; `String => String noexcept` 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 $\mathcal{L}$, and they give a design recipe for enriching an existing type system with support for type qualifiers.

(1) First, embed qualifiers into a lattice $\mathcal{L}$. 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:
(3) 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 $\{l_1\}\ T_1 <: \{l_2\}\ T_2$.
(4) Add rules for *introducing qualifiers*, typically in the introduction forms for typing values.
(5) 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 ($\sqcup$) or meet ($\sqcap$) 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` $\sqcup$ `Q` to just `Q`; the result of `twice` throws if `f` throws or if `f` throws, which is namely just if `f` throws.

```
def twice[A, Q](f: (A => A) Q): (A => A) Q = 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 $\vee, \wedge, \leq$ in place of $\sqcup, \sqcap, \sqsubseteq$ 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 $\mathcal{E}(X)$, the set of *lattice formulas generated by* X, is recursively defined by:

(1) If $x \in X$ then $x \in \mathcal{E}(X)$.
(2) If $f_1 \in \mathcal{E}(X)$ and $f_2 \in \mathcal{E}(X)$ then $f_1 \wedge f_2 \in \mathcal{E}(X)$.
(3) If $f_1 \in \mathcal{E}(X)$ and $f_2 \in \mathcal{E}(X)$ then $f_1 \vee f_2 \in \mathcal{E}(X)$.

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

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

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]

[1] Galatos [2023] and Negri and von Plato [2002] give algebraic and structural proofs of this result. Jipsen [2001] notes this can also be viewed as a refomulation of Whitman [1941, Theorem 1]. Skolem [1920] however is probably responsible for the original formulation of Theorem 2.3 though with *transitivity* as an additional rule (8).

Theorem 2.3 (Folklore). *$\mathcal{F}(X)$ is the lattice over $(\mathcal{E}(X)/\leq, \leq)$, where $\leq$ is a binary relation over $\mathcal{E}(X)$ defined by:*

(1) *For every $x \in X$, $x \leq x$.*
(2) *For every $f_1, f_2, f_3 \in \mathcal{E}(X)$, if $f_3 \leq f_1$ then $f_3 \leq f_1 \vee f_2$.*
(3) *For every $f_1, f_2, f_3 \in \mathcal{E}(X)$, if $f_3 \leq f_2$ then $f_3 \leq f_1 \vee f_2$.*
(4) *For every $f_1, f_2, f_3 \in \mathcal{E}(X)$, if $f_3 \leq f_1$ and $f_3 \leq f_2$ then $f_1 \vee f_2 \leq f_3$.*
(5) *For every $f_1, f_2, f_3 \in \mathcal{E}(X)$, if $f_1 \leq f_3$ then $f_1 \wedge f_2 \leq f_3$.*
(6) *For every $f_1, f_2, f_3 \in \mathcal{E}(X)$, if $f_2 \leq f_3$ then $f_1 \wedge f_2 \leq f_3$.*
(7) *For every $f_1, f_2, f_3 \in \mathcal{E}(X)$, if $f_3 \leq f_1$ and $f_3 \leq f_2$ then $f_3 \leq f_1 \wedge f_2$.*

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 *free lattice* generated over that base qualifier lattice, and we support qualifier polymorphism explicitly with bounds, following System $\mathsf{F}_{<:}$, instead of implicitly at prenex position with constraints, as Foster et al. [1999] do.

2.5 System $\mathsf{F}_{<:\mathsf{Q}}$

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

Assigning Qualifiers. In System $\mathsf{F}_{<:\mathsf{Q}}$, 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 $\mathsf{F}_{<:\mathsf{Q}}$ is only a base calculus.

Syntax. Figure 2 presents the syntax of System $\mathsf{F}_{<:\mathsf{Q}}$, with additions over System $\mathsf{F}_{<:}$ 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 *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 *tag* values with a qualifier expression P denoting the qualifier that value should be typed at and we add support for *asserting* as well as *upcasting* qualifier tags, following Foster et al. [1999, Section 2.2]. For example, the value $\lambda(x : \mathtt{Int})_\top x$ would represent the integer identity function qualified at $\top$.

While System $\mathsf{F}_{<:\mathsf{Q}}$ does not provide a default tag for values, negative (or contravariant) qualifiers like `noexcept` would inform a default qualifier tag choice of $\top$ – by default, functions are assumed to throw – and positive (or covariant) qualifiers like `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 $\mathsf{F}_{<:\mathsf{Q}}$ (defined in Figure 3) are largely unchanged from System $\mathsf{F}_{<:}$. To support *qualifier polymorphism*, we add the rule (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 (upqual) and (assert) for asserting

s, t	::=		**Terms**
	$\mid$	$\lambda(x)_P.t$	term abstraction
	$\mid$	x	term variable
	$\mid$	$s(t)$	application
	$\mid$	$\Lambda(X <: S)_P.t$	type abstraction
	$\mid$	$\Lambda(Y <: Q)_P.t$	qualifier abstraction
	$\mid$	$s[S]$	type application
	$\mid$	$s\{\!\{Q\}\!\}$	qualifier application
	$\mid$	`upqual` $Q\ s$	qualifier upcast
	$\mid$	`assert` $Q\ s$	qualifier assertion

S	::=		**Simple Types**
	$\mid$	$\top$	top type
	$\mid$	$T_1 \to T_2$	function type
	$\mid$	X	type variable
	$\mid$	$\forall(X <: S).T$	for-all type
	$\mid$	$\forall(Y <: Q).T$	qualifier for-all type
T	::=		**Qualified Types**
	$\mid$	$\{Q\}\ S$	qualified type
P, Q, R	::=		**Qualifiers**
	$\mid$	$\top, \bot$	Top and bottom
	$\mid$	Y	Qualifier variables
	$\mid$	$Q \wedge R \mid Q \vee R$	Meets and joins

Γ	::=		**Environment**
	$\mid$	$\cdot$	empty
	$\mid$	$\Gamma,\ x : T$	term binding
	$\mid$	$\Gamma,\ X <: S$	type binding
	$\mid$	$\Gamma,\ Y <: Q$	qualifier binding

v	::=		**Values**
	$\mid$	$\lambda(x)_P.t$	
	$\mid$	$\Lambda(X <: S)_P.t$	
	$\mid$	$\Lambda(Y <: Q)_P.t$	

C	::=		**Concrete Qualifiers**
	$\mid$	$\top$ or $\bot$	two-point lattice elements

Lattice facts reminder: $\bot \sqsubseteq \bot$, $\bot \sqsubseteq \top$, and $\top \sqsubseteq \top$. $\top \sqcap C = C$, $\top \sqcup C = \top$, $\bot \sqcap C = \bot$, and $\bot \sqcup C = C$.

Fig. 2. The syntax of System $\mathsf{F}_{<:\mathsf{Q}}$. Qualified differences to System $\mathsf{F}_{<:}$ highlighted in *grey*.

and upcasting qualifier tags: they coerce qualifier expressions to concrete qualifiers when possible and ensure that the concrete qualifiers are compatible before successfully reducing.

Subqualification. Next we show how simple subqualification extends from a extends from a lattice inequality in a base lattice (like how `noexcept` <: `throws`) to a lattice inequality in a free lattice. Figure 4 captures this free lattice structure of the qualifiers of System $\mathsf{F}_{<:\mathsf{Q}}$ with a *subqualification* judgment $\Gamma \vdash Q <: R$ to make precise the partial order between two lattice formulas in a free lattice, though slightly modified to support upper bounds on variables. This basic structure should appear familiar—it is a simplified subtyping lattice. It should not be surprising that this construction gives rise to the free lattice, though we make this property explicit in supplementary material. One can use this structure to deduce desirable subqualification judgments; for example, in an environment $\Gamma = [X <: A, Y <: B, A <: \top, B <: \top]$, we can show that $X \vee Y <: A \vee B$, using the following rule applications.

$$X <: A \vee B \text{ by (sq-join-intro-1)}$$
$$Y <: A \vee B \text{ by (sq-join-intro-2)}$$
$$X \vee Y <: A \vee B \text{ by (sq-join-elim)}$$

Subtyping. System $\mathsf{F}_{<:\mathsf{Q}}$ inherits most of its rules for subtyping from System $\mathsf{F}_{<:}$, with two changes made (Figure 5). The additional rule (sub-qall) handles subtyping for qualifier abstractions, and rule (sub-qtype) handles subtyping for qualified types. A qualified type $\{Q_1\}\ S_1$ is a subtype of

Evaluation for System $\mathsf{F}_{<:\mathsf{Q}}$

$$\boxed{s \longrightarrow t} \text{ and } \boxed{\mathtt{eval}\, Q}$$

$$(\lambda(x)_P.t)(s) \longrightarrow t[x \mapsto s] \qquad (\textsc{beta-v})$$

$$(\Lambda(X <: S)_P.t)[S'] \longrightarrow t[X \mapsto S'] \qquad (\textsc{beta-T})$$

$$(\Lambda(Y <: Q)_P.t)\{\!\!\{Q'\}\!\!\} \longrightarrow t[Y \mapsto Q'] \qquad (\textsc{beta-Q})$$

$$\frac{v \text{ tagged with } P \qquad \mathtt{eval}(P) \sqsubseteq \mathtt{eval}(Q)}{\mathtt{upqual}\; Q\; v \longrightarrow v \text{ retagged with } Q} \ (\textsc{upqual})$$

$$\frac{v \text{ tagged with } P \qquad \mathtt{eval}(P) \sqsubseteq \mathtt{eval}(Q)}{\mathtt{assert}\; P\; v \longrightarrow v} \ (\textsc{assert})$$

$$\frac{s \longrightarrow t}{E[s] \longrightarrow E[t]} \ (\textsc{context})$$

$$\begin{array}{lll} E & ::= & \textbf{Evaluation Context} \\ & | & [] \\ & | & E(t) \mid v(E) \\ & | & E[S] \mid E[Q] \\ & | & \mathtt{upqual}\; P\; E \\ & | & \mathtt{assert}\; P\; E \end{array}$$

$$\begin{array}{llll} \mathtt{eval}(P) & ::= & & \textbf{Partial Qualifier Evaluation} \\ & \mid C & \texttt{=>} & C \\ & \mid P \wedge R & \texttt{=>} & \mathtt{eval}(P) \sqcap \mathtt{eval}(R) \\ & \mid P \vee R & \texttt{=>} & \mathtt{eval}(P) \sqcup \mathtt{eval}(R) \\ & \mid \_ & \texttt{=>} & \text{nothing, otherwise.} \end{array}$$

Fig. 3. Reduction rules for System $\mathsf{F}_{<:\mathsf{Q}}$

Subqualification for System $\mathsf{F}_{<:\mathsf{Q}}$

$$\boxed{\Gamma \vdash Q <: R}$$

$$\Gamma \vdash Q <: \top \qquad (\textsc{sq-top})$$

$$\Gamma \vdash \bot <: Q \qquad (\textsc{sq-bot})$$

$$\frac{\Gamma \vdash Q <: R_1}{\Gamma \vdash Q <: R_1 \vee R_2} \ (\textsc{sq-join-intro-1})$$

$$\frac{\Gamma \vdash Q <: R_2}{\Gamma \vdash Q <: R_1 \vee R_2} \ (\textsc{sq-join-intro-2})$$

$$\frac{\Gamma \vdash R_1 <: Q \qquad \Gamma \vdash R_2 <: Q}{\Gamma \vdash R_1 \vee R_2 <: Q} \ (\textsc{sq-join-elim})$$

$$\frac{\Gamma \vdash R_1 <: Q}{\Gamma \vdash R_1 \wedge R_2 <: Q} \ (\textsc{sq-meet-elim-1})$$

$$\frac{\Gamma \vdash R_2 <: Q}{\Gamma \vdash R_1 \wedge R_2 <: Q} \ (\textsc{sq-meet-elim-2})$$

$$\frac{\Gamma \vdash Q <: R_1 \qquad \Gamma \vdash Q <: R_2}{\Gamma \vdash Q <: R_1 \wedge R_2} \ (\textsc{sq-meet-intro})$$

$$\frac{Y <: Q \in \Gamma \qquad \Gamma \vdash Q <: R}{\Gamma \vdash Y <: R} \ (\textsc{sq-var})$$

$$\frac{Y <: Q \in \Gamma}{\Gamma \vdash Y <: Y} \ (\textsc{sq-refl-var})$$

Fig. 4. Subqualification rules of System $\mathsf{F}_{<:\mathsf{Q}}$.

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: $S_1 <: S_2$.

Subtyping for System $\mathsf{F}_{<:\mathsf{Q}}$ $\boxed{\Gamma \vdash S_1 <: S_1 \text{ and } \Gamma \vdash T_1 <: T_2}$

$$\Gamma \vdash S <: \top \qquad \text{(SUB-TOP)}$$

$$\frac{X \in \Gamma}{\Gamma \vdash X <: X} \ \text{(SUB-REFL-SVAR)}$$

$$\frac{X <: S_1 \in \Gamma \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash X <: S_2} \ \text{(SUB-SVAR)}$$

$$\frac{\Gamma \vdash Q_1 <: Q_2 \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash \{Q_1\}\ S_1 <: \{Q_2\}\ S_2} \ \text{(SUB-QTYPE)}$$

$$\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma \vdash T_3 <: T_4}{\Gamma \vdash T_1 \to T_3 <: T_2 \to T_4} \ \text{(SUB-ARROW)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, X <: S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(X <: S_1).T_1 <: \forall(X <: S_2).T_2} \ \text{(SUB-ALL)}$$

$$\frac{\Gamma \vdash Q_2 <: Q_1 \qquad \Gamma, Y <: Q_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(Y <: Q_1).T_1 <: \forall(Y <: Q_2).T_2} \ \text{(SUB-QALL)}$$

Fig. 5. Subtyping rules of System $\mathsf{F}_{<:\mathsf{Q}}$.

Typing for System $\mathsf{F}_{<:\mathsf{Q}}$ $\boxed{\Gamma \vdash t : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \ \text{(VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda(x)_P.t : \{\ P\ \}\ T_1 \to T_2} \ \text{(ABS)}$$

$$\frac{\Gamma, X <: S \vdash t : T}{\Gamma \vdash \Lambda(X <: S)_P.t : \{\ P\ \}\ \forall(X <: S).T} \ \text{(T-ABS)}$$

$$\frac{\Gamma, X <: S \vdash t : T}{\Gamma \vdash \Lambda(Y <: Q)_P.t : \{\ P\ \}\ \forall(Y <: Q).T} \ \text{(Q-ABS)}$$

$$\frac{\Gamma \vdash t : \{Q\}\ S \qquad \Gamma \vdash Q <: P}{\Gamma \vdash \mathtt{assert}\, P\ t : \{Q\}\ S} \ \text{(TYP-ASSERT)}$$

$$\frac{\Gamma \vdash t : \{Q\}\ T_1 \to T_2 \qquad \Gamma \vdash s : T_1}{\Gamma \vdash t(s) : T_2} \ \text{(APP)}$$

$$\frac{\Gamma \vdash t : \{Q\}\ \forall(X <: S).T \qquad \Gamma \vdash S' <: S}{\Gamma \vdash t[S'] : T[X \mapsto S']} \ \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : \{R\}\ \forall(Y <: Q).T \qquad \Gamma \vdash Q' <: Q}{\Gamma \vdash t\{\!\{Q'\}\!\} : T[Y \mapsto Q']} \ \text{(Q-APP)}$$

$$\frac{\Gamma \vdash s : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash s : T_2} \ \text{(SUB)}$$

$$\frac{\Gamma \vdash t : \{Q\}\ S \qquad \Gamma \vdash Q <: P}{\Gamma \vdash \mathsf{upqual}\, P\ t : \{P\}\ S} \ \text{(TYP-UPQUAL)}$$

Fig. 6. Typing rules for System $\mathsf{F}_{<:\mathsf{Q}}$

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 $\mathsf{F}_{<:\mathsf{Q}}$. The typing judgment assigns qualified types T to expressions, and can be viewed as $\Gamma \vdash t : \{Q\}\ S$. As System $\mathsf{F}_{<:\mathsf{Q}}$ 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

P, Q, R	$::=$		**Qualifiers in extended System $\mathsf{F}_{<:\mathsf{Q}}$**
	$\vert$	l	Base lattice elements $l \in L$
	$\vert$	Y	Qualifier variables
	$\vert$	$Q \wedge R \mid Q \vee R$	Meets and joins
C	$::=$		**Concrete Qualifiers**
	$\vert$	l	Base lattice elements $l \in L$

Fig. 7. The syntax of System $\mathsf{F}_{<:\mathsf{Q}}$ extended over a bounded lattice $\mathcal{L}$. Differences to System $\mathsf{F}_{<:\mathsf{Q}}$ 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 $\mathsf{F}_{<:}$.

2.6 Metatheory

System $\mathsf{F}_{<:\mathsf{Q}}$ satisfies the standard progress and preservation theorems.

THEOREM 2.4 (PRESERVATION). *Suppose* $\Gamma \vdash s : T$, *and* $s \longrightarrow t$. *Then* $\Gamma \vdash t : T$ *as well.*

THEOREM 2.5 (PROGRESS). *Suppose* $\varnothing \vdash s : T$. *Then either* s *is a value, or* $s \longrightarrow t$ *for some term* t.

While System $\mathsf{F}_{<:\mathsf{Q}}$ 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 $\mathsf{F}_{<:\mathsf{Q}}$ for two-point lattices to support general (countable, bounded) qualifier lattices $\mathcal{L}$ 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 $\mathcal{L}$, but as $\mathcal{L}$ is bounded, note that we still have distinguished elements $\top$ and $\bot$ in $\mathcal{L}$.

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 $\mathcal{L}$ is equipped with up to the free lattice order defined by the subqualification lattice. Rules (SQ-EVAL-ELIM) and

Subqualification for System $\mathsf{F}_{<:\mathsf{Q}}$ over a lattice $\mathcal{L}$ $\boxed{\Gamma \vdash Q <: R}$

$$\frac{l_1, l_2 \in \mathcal{L} \qquad l_1 \sqsubseteq l_2}{\Gamma \vdash l_1 <: l_2}\ (\textsc{sq-lift})$$

$$\frac{\Gamma \vdash Q <: Q' \qquad \Gamma \vdash l = \mathsf{eval}\, Q' \qquad \Gamma \vdash l <: R}{\Gamma \vdash Q <: R}\ (\textsc{sq-eval-elim})$$

$$\frac{\Gamma \vdash Q <: l \qquad \Gamma \vdash l = \mathsf{eval}\, Q' \qquad \Gamma \vdash Q' <: R}{\Gamma \vdash Q <: R}\ (\textsc{sq-eval-intro})$$

Fig. 8. Extended sub-qualification rules for System $\mathsf{F}_{<:\mathsf{Q}}$.

(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 $\mathcal{L}$, like $l_1 \vee l_2$, 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 $\Gamma \vdash l_1 \vee l_2 <: l_1 \sqcup l_2$ and that $\Gamma \vdash l_1 \sqcup l_2 <: l_1 \vee l_2$. However, without the two evaluation rules (sq-eval-elim) and (sq-eval-intro), we would only be able to conclude that $\Gamma \vdash l_1 \vee l_2 <: l_1 \sqcup l_2$, but not the other desired inequality $\Gamma \vdash l_1 \sqcup l_2 <: l_1 \vee l_2$.

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 $\mathcal{L}$, though we make this precise in the supplementary material.

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

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

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

3 APPLICATIONS

Having introduced our design recipe by constructing System $\mathsf{F}_{<:\mathsf{Q}}$ as a qualified extension of System $\mathsf{F}_{<:}$, 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 $\mathsf{F}_{<:\mathsf{Q}}$, 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 $\mathsf{F}_{<:\mathsf{Q}}$; in essence, expressiveness and polymorphism come for *free* from base System $\mathsf{F}_{<:\mathsf{Q}}$.

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[Box[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 $\mathsf{F}_{<:\mathsf{QM}}$.

Assigning Qualifiers. We need to define how qualifiers `mutable` and `readonly` are assigned to $\top$ and $\bot$ in System $\mathsf{F}_{<:\mathsf{QM}}$. 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 σ that maps locations to values. Reduction now takes place over pairs of terms and stores $\langle t, \sigma\rangle$:

$$\begin{array}{rcl} \langle \texttt{set-box! (box}_{\bot}\ 10)\ 5, [\,]\rangle & \longrightarrow & \langle \texttt{set-box! (box}_{\bot}\ \texttt{0x0001})\ 5, [\texttt{0x0001} : 10]\rangle \\ & \longrightarrow & \langle 10, [\texttt{0x0001} : 5]\rangle \end{array}$$

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 term gets stuck.

$$\begin{array}{rcl} \langle \texttt{set-box! (box}_{\top}\ 10)\ 5, [\,]\rangle & \longrightarrow & \langle \texttt{set-box! (box}_{\top}\ \texttt{0x0001})\ 5, [\texttt{0x0001} : 10]\rangle \\ & \longrightarrow & \text{gets stuck.} \end{array}$$

$$
\begin{array}{rcll}
s, t & ::= & & \textbf{Terms} \\
 & \dots & & \\
 & | & \mathtt{box}_P\ t & \text{reference cell} \\
 & | & \mathtt{unbox}\ s & \text{deferencing} \\
 & | & \mathtt{set\text{-}box!}\ s\ t & \text{reference update}
\end{array}
\qquad
\begin{array}{rcll}
S & ::= & & \textbf{Types} \\
 & \dots & & \\
 & | & \mathtt{box}\ S & \text{reference type} \\
P, Q, R & ::= & & \textbf{Qualifiers} \\
 & \dots & & \text{as before, except:} \\
 & | & \mathtt{readonly} & \text{as } \top \\
 & | & \mathtt{mutable} & \text{as } \bot
\end{array}
$$

$$
\begin{array}{rcll}
 & & l & \textbf{Location} \\
s, t & ::= & & \textbf{Runtime Terms} \\
 & | & \mathtt{box}_P\ l & \text{runtime reference} \\
v & ::= & & \textbf{Runtime Values} \\
 & \dots & & \\
 & | & \mathtt{box}_P\ l &
\end{array}
\qquad
\begin{array}{rcll}
\sigma & ::= & & \textbf{Store} \\
 & | & \cdot & \text{empty} \\
 & | & \sigma,\ l : v & \text{cell } l \text{ with value } v \\
\Sigma & ::= & & \textbf{Store Environment} \\
 & | & \cdot & \text{empty} \\
 & | & \sigma,\ l : T & \text{cell binding}
\end{array}
$$

Fig. 9. The syntax of System $\mathsf{F}_{<:\mathtt{QM}}$.

Additional Evaluation Rules for System $\mathsf{F}_{<:\mathtt{QM}}$ $\boxed{\langle s, \sigma \rangle \longrightarrow \langle t, \sigma' \rangle}$

$$
\frac{l \notin \sigma}{\langle \mathtt{box}_P\ v, \sigma \rangle \longrightarrow \langle \mathtt{box}_P\ l, (\sigma, l : v) \rangle}\ (\textsc{ref-store})
\qquad
\frac{l : v \in \sigma \qquad \mathtt{eval}(P) \sqsubseteq \bot}{\langle \mathtt{set\text{-}box!}\ (\ \mathtt{box}_P\ l\ )\ v', \sigma \rangle \mapsto \langle v, \sigma[l \mapsto v'] \rangle}\ (\textsc{write-ref})
$$

$$
\frac{l : v \in \sigma \qquad v \text{ tagged with } Q}{\langle \mathtt{unbox}\ \ \mathtt{box}_P\ l, \sigma \rangle \longrightarrow \langle v \text{ retagged at } P \vee Q, \sigma \rangle}\ (\textsc{deref})
\qquad
\frac{\langle s, \sigma \rangle \longrightarrow \langle t, \sigma' \rangle}{\langle E[s], \sigma \rangle \longrightarrow \langle E[t], \sigma' \rangle}\ (\textsc{context})
$$

$$
\begin{array}{rcl}
E & ::= & \dots \\
 & | & \mathtt{box}_P\ E \\
 & | & \mathtt{unbox}\ E \\
 & | & \mathtt{set\text{-}box!}\ E\ \ t \mid \mathtt{set\text{-}box!}\ v\ \ E
\end{array}
\qquad \textbf{Evaluation Context}
$$

Fig. 10. Reduction rules for System $\mathsf{F}_{<:\mathtt{QM}}$

One can explicitly mark a value immutable by `upqual`-ing to ⊤. The elimination form for reading from a reference, (DEREF), ensures that a value read from a reference tagged `readonly`, or at ⊤, remains `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 (REF-UPDATE) is augmented to check that a reference can only be written to if and only if it can be typed as `mutable` box . Finally, the standard reference read elimination form (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 $\mathsf{F}_{<:\mathtt{QM}}$ $\boxed{\Gamma \mid \Sigma \vdash t : T \text{ and } \Gamma \mid \Sigma \vdash \sigma}$

$$\frac{\Gamma \mid \Sigma \vdash t : T}{\Gamma \mid \Sigma \vdash \mathsf{box}_P\ t : \{P\}\ \mathsf{box}\ T}\ (\textsc{ref-intro}) \qquad \frac{\Gamma \mid \Sigma \vdash t : \{Q_1\}\ \mathsf{box}\{Q_2\}\ S}{\Gamma \mid \Sigma \vdash \mathsf{unbox}\ t : \{\ Q_1 \vee Q_2\ \}\ S}\ (\textsc{ref-elim})$$

$$\frac{l : T \in \Sigma}{\Gamma \mid \Sigma \vdash \mathsf{box}_P\ l : \{P\}\ \mathsf{box}\ T}\ (\textsc{runtime-ref-intro}) \qquad \frac{\Gamma \vdash s : \{\ \mathtt{mutable}\ \}\ \mathsf{box}\ T \qquad \Gamma \vdash t : T}{\Gamma \vdash \mathtt{set\text{-}box!}\ s\ \ t : T}\ (\textsc{ref-update})$$

$$\frac{dom(\sigma) = dom(\Sigma) \qquad \forall l \in dom(\Sigma),\ \Gamma \mid \Sigma \vdash \sigma(l) : \Sigma(l)}{\Gamma \mid \Sigma \vdash \sigma}\ (\textsc{store})$$

Fig. 11. Typing rules for System $\mathsf{F}_{<:\mathtt{QM}}$; notable changes highlighted in *grey*.

safety. Other than qualifiers, our construction is completely standard; we merely add a store σ and a runtime store environment Σ mapping store locations to types.

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

Theorem 3.1 (Preservation of System $\mathsf{F}_{<:\mathtt{QM}}$). *Suppose* $\langle s, \sigma\rangle \longrightarrow \langle t, \sigma'\rangle$. *If* $\Gamma \mid \Sigma \vdash \sigma$ *and* $\Gamma \mid \Sigma \vdash s : T$ *for some type* T, *then there is some environment extension* Σ' *of* Σ *such that* $\Gamma \mid \Sigma' \vdash \sigma'$ *and* $\Gamma \mid \Sigma' \vdash t : T$.

Theorem 3.2 (Progress for System $\mathsf{F}_{<:\mathtt{QM}}$). *Suppose* $\varnothing \mid \Sigma \vdash \sigma$ *and* $\varnothing \mid \Sigma \vdash s : T$. *Then either* s *is a value or there is some* t *and* σ' *such that* $\langle s, \sigma\rangle \longrightarrow \langle t, \sigma'\rangle$.

With only progress and preservation, we can already state something meaningful about the immutability safety of System $\mathsf{F}_{<:\mathtt{QM}}$: 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.

```
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.

```
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`:

```
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.

```
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?

```
val follow = ["https://plg.uwaterloo.ca/~e45lee", "https://plg.uwaterloo.ca/~
    olhotak", "https://b-studios.de"]
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`.

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

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

```
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.

```
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 $\mathsf{F}_{<:\mathsf{QA}}$, 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.

$$
\begin{array}{rcll}
P, Q, R & ::= & & \textbf{Qualifiers} \\
 & \ldots & & \text{as before, except:} \\
 & | & \mathsf{async}\ (\text{as } \top) & \text{async qualifier} \\
 & | & \mathsf{sync}\ (\text{as } \bot) & \text{sync qualifier} \\
\kappa & ::= & & \textbf{Evaluation Context} \\
 & | & [] & \\
 & | & f :: \kappa &
\end{array}
\qquad
\begin{array}{rcll}
f & ::= & & \textbf{Evaluation Frames} \\
 & | & \mathsf{barrier}\, C & \text{barrier} \\
 & | & \mathsf{arg}\, t & \text{argument} \\
 & | & \mathsf{app}\, v & \text{application} \\
 & | & \mathsf{targ}\, T & \text{type application} \\
 & | & \mathsf{qarg}\, Q & \text{qualifier application}
\end{array}
$$

Fig. 12. The syntax of System $\mathsf{F}_{<:\mathsf{QA}}$.

Evaluation for System $\mathsf{F}_{<:\mathsf{QA}}$ $\boxed{\langle c, \kappa \rangle \longrightarrow \langle c', \kappa' \rangle}$

$$\langle s(t), \kappa \rangle \longrightarrow \langle s, \mathsf{arg}\, t :: \kappa \rangle \quad (\text{CONG-APP})$$

$$\langle v, \mathsf{arg}\, t :: \kappa \rangle \longrightarrow \langle t, \mathsf{app}\, v :: \kappa \rangle \quad (\text{CONG-ARG})$$

$$\langle s[S], \kappa \rangle \longrightarrow \langle s, \mathsf{targ}\, S :: \kappa \rangle \quad (\text{CONG-TAPP})$$

$$\langle s\{\!\{Q\}\!\}, \kappa \rangle \longrightarrow \langle s, \mathsf{qarg}\, Q :: \kappa \rangle \quad (\text{CONG-QAPP})$$

$$\langle v, \mathsf{barrier}\, C :: \kappa \rangle \longrightarrow \langle v, \kappa \rangle \quad (\text{BREAK-BARRIER})$$

$$\frac{C \le C_i \text{ for all } \mathsf{barrier}\ C_i \text{ frames on } \kappa \qquad \mathsf{eval}\, P = C}{\langle v, \mathsf{app}\, \lambda(x)_P.t :: \kappa \rangle \longrightarrow \langle t[x \mapsto v], \mathsf{barrier}\, C :: \kappa \rangle} \quad (\text{REDUCE-APP})$$

$$\frac{C \le C_i \text{ for all } \mathsf{barrier}\ C_i \text{ frames on } \kappa \qquad \mathsf{eval}\, P = C}{\langle \Lambda(X <: S)_P.t, \mathsf{targ}\, S' :: \kappa \rangle \longrightarrow \langle t[X \mapsto S'], \mathsf{barrier}\, C :: \kappa \rangle} \quad (\text{REDUCE-TAPP})$$

$$\frac{C \le C_i \text{ for all } \mathsf{barrier}\ C_i \text{ frames on } \kappa \qquad \mathsf{eval}\, P = C}{\langle \Lambda(Y <: Q)_P.t, \mathsf{qarg}\, Q' :: \kappa \rangle \longrightarrow \langle t[Y \mapsto Q'], \mathsf{barrier}\, C :: \kappa \rangle} \quad (\text{REDUCE-QAPP})$$

Fig. 13. Operational Semantics (CK-style) for System $\mathsf{F}_{<:\mathsf{QA}}$

Syntax. Figure 12 presents the modified syntax of System $\mathsf{F}_{<:\mathsf{QA}}$. 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 $\lambda(x)_{\mathsf{async}}.\ x$, and an example of a function that is polymorphic in its qualifier is $\Lambda(Y <: \mathsf{sync})_{\mathsf{async}}.\lambda(f)_Y.\ f(1)$, 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 $\mathsf{F}_{<:\mathsf{QA}}$ 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

$$
\begin{array}{lll}
\langle 1, \mathsf{app}\, \lambda(x)_\bot.x :: \kappa \rangle & \longrightarrow & \langle 1, \mathsf{barrier}\, \bot :: \kappa \rangle \\
 & & \text{placing a barrier marking a synchronus function on the stack.}
\end{array}
$$

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:

$$
\begin{array}{lll}
\langle (\lambda(x)_\top.t)\ v, \mathsf{barrier}\, \bot \rangle & \longrightarrow & \langle \lambda(x)_\top.t), \mathsf{arg}\, v :: \mathsf{barrier}\, \bot \rangle \\
 & \longrightarrow & \langle v, \mathsf{app}\, \lambda(x)_\top.t :: \mathsf{barrier}\, \bot \rangle \\
 & \longrightarrow & \text{gets stuck.}
\end{array}
$$

The other evaluation contexts are standard.

Typing for System $\mathsf{F}_{<:\mathsf{QA}}$ $\boxed{\Gamma \mid R \vdash s : T}$

$$\frac{x : T \in \Gamma}{\Gamma \mid R \vdash x : T} \quad \text{(A-VAR)}$$

$$\frac{\Gamma, x : T_1 \mid P \vdash t : T_2}{\Gamma \mid \mathsf{sync} \vdash \lambda(x)_P.t : \{P\}\ T_1 \to T_2} \quad \text{(A-ABS)}$$

$$\frac{\Gamma, X <: S \mid P \vdash t : T}{\Gamma \mid \mathsf{sync} \vdash \Lambda(X <: S)_P.t : \{P\}\ \forall(X <: S).T} \quad \text{(A-T-ABS)}$$

$$\frac{\Gamma, Y <: Q \mid P \vdash t : T}{\Gamma \mid \mathsf{sync} \vdash \Lambda(Y <: Q)_P.t : \{P\}\ \forall(Y <: Q).T} \quad \text{(A-Q-ABS)}$$

$$\frac{\Gamma \mid R \vdash t : \{\ R\ \}\ T_1 \to T_2 \qquad \Gamma \vdash s : T_1}{\Gamma \mid R \vdash t(s) : T_2} \quad \text{(A-APP)}$$

$$\frac{\Gamma \mid R \vdash t : \{\ R\ \}\ \forall(X <: S).T \qquad \Gamma \vdash S' <: S}{\Gamma \mid R \vdash t[S'] : T[X \mapsto S']} \quad \text{(A-T-APP)}$$

$$\frac{\Gamma \mid R \vdash t : \{\ R\ \}\ \forall(Y <: Q).T \qquad \Gamma \vdash Q' <: Q}{\Gamma \mid R \vdash t\{\!\{Q'\}\!\} : T[Y \mapsto Q']} \quad \text{(A-Q-APP)}$$

$$\frac{\Gamma \mid R \vdash s : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \mid R \vdash s : T_2} \quad \text{(A-SUB)}$$

$$\frac{\Gamma \mid R \vdash s : T_1 \qquad \Gamma \vdash R <: Q}{\Gamma \mid Q \vdash s : T_2} \quad \text{(A-SUB-EFF)}$$

Fig. 14. Typing rules for System $\mathsf{F}_{<:\mathsf{QA}}$

Typing. To guarantee soundness, Figure 14 endows the typing rules of System $\mathsf{F}_{<:\mathsf{QA}}$ 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 $\bot$.

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 $\mathsf{F}_{<:\mathsf{QA}}$.

THEOREM 3.3 (PROGRESS OF SYSTEM $\mathsf{F}_{<:\mathsf{QA}}$). *Suppose $\langle c, \kappa \rangle$ is a well-typed machine configuration. Then either c is a value and k is the empty continuation, or there is a machine state $\langle c', \kappa' \rangle$ that it steps to.*

THEOREM 3.4 (PRESERVATION OF SYSTEM $\mathsf{F}_{<:\mathsf{QA}}$). *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 $\mathsf{F}_{<:\mathsf{QA}}$: 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 $\mathsf{F}_{<:\mathsf{QA}}$ 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 $\top$; (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 $\mathsf{F}_{<:\mathsf{QA}}$ 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!

```
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]:

```
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.

```
val print : {tracked} String => Unit = ???
def helloWorld() = print("Hello␣World!")  // tracked as it captures print
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

Term	**Qualifier**	**Reason**
pandora	tracked	As pandora is a base capability.
print	tracked	As print is a base capability.
takeOverTheWorld	**pandora**	As takeOverTheWorld is no more dangerous than pandora.
helloWorld	**print**	As helloWorld is no more dangerous than print.

Fig. 15. Qualifier assignments in Capture Tracking

$$\begin{array}{rcll}
s,t & ::= & & \textbf{Terms}\\
 & & \dots & \\
 & | & s\{\!\!\{Q\}\!\!\}(t) & \text{term application}\\
S & ::= & & \textbf{Types}\\
 & & \dots & \\
 & | & (x : T_1) \to T_2 & \text{function type}\\
P,Q,R & ::= & & \textbf{Qualifiers}\\
 & & \dots & \text{as before, except:}\\
 & | & x & \text{term variables}\\
 & | & \texttt{tracked}\ (\text{as}\ \top) & \text{tracked values}
\end{array}$$

Evaluation: $\boxed{s \longrightarrow t}$

$$\begin{array}{c} (\lambda(x)_P.t)\{\!\!\{\ Q\ \}\!\!\}(s) \longrightarrow \\ t[x \mapsto_{\texttt{type}} Q][x \mapsto_{\texttt{term}} s] \end{array} \quad (\textsc{C-beta-v})$$

Subqualification: $\boxed{\Gamma \vdash Q \mathrel{<:} R}$

$$\frac{x : \{Q\}\ S \in \Gamma \qquad \Gamma \vdash Q \mathrel{<:} R}{\Gamma \vdash x \mathrel{<:} R} \quad (\textsc{sq-tvar})$$

$$\frac{x : \{Q\}\ S \in \Gamma}{\Gamma \vdash x \mathrel{<:} x} \quad (\textsc{sq-refl-tvar})$$

Subtyping: $\boxed{\Gamma \vdash S_1 \mathrel{<:} S_2}$

$$\frac{\Gamma \vdash T_2 \mathrel{<:} T_1 \qquad \Gamma, x : T_2 \vdash T_3 \mathrel{<:} T_4}{\Gamma \vdash (x : T_2) \to T_3 \mathrel{<:} (x : T_1) \to T_4} \quad (\textsc{C-sub-arrow})$$

Fig. 16. Evaluation, Syntax, Subtyping for System $\mathsf{F}_{<:\mathsf{QC}}$

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 = {
    print("I␣expect␣you␣to␣die␣in␣polynomial␣time,␣Mr.␣Bond.")
    takeOverTheWorld()
}
```

Using this insight, we can model capture tracking as an extension System $\mathsf{F}_{<:\mathsf{QC}}$ of System $\mathsf{F}_{<:\mathsf{Q}}$.

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.

Typing for System $\mathsf{F}_{<:\mathsf{QC}}$ $\boxed{\Gamma \vdash t : T}$

$$\frac{x : \{Q\}\ S \in \Gamma}{\Gamma \vdash x : \{x\}\ S}\ \text{(C-VAR)}$$

$$\frac{\Gamma \vdash s : (x : \{Q\}\ S) \to T \qquad \Gamma \vdash Q' <: Q \qquad \Gamma \vdash t : \{Q'\}\ S}{\Gamma \vdash s\{\!\{Q'\}\!\}(t) : T[x \mapsto_{\mathsf{type}} Q']}\ \text{(C-APP)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2 \qquad \Gamma \vdash \bigvee_{y \in \mathsf{fv}(t) - x} y <: P}{\Gamma \vdash \lambda(x)_P.t : \{P\}\ (x : T_1) \to T_2}\ \text{(C-ABS)}$$

$$\frac{\Gamma, X <: S \vdash t : T \qquad \Gamma \vdash \bigvee_{y \in \mathsf{fv}(t)} y <: P}{\Gamma \vdash \Lambda(X <: S)_P.t : \{P\}\ \forall(X <: S).T}\ \text{(C-T-ABS)}$$

$$\frac{\Gamma, X <: S \vdash t : T \qquad \Gamma \vdash \bigvee_{y \in \mathsf{fv}(t)} y <: P}{\Gamma \vdash \Lambda(Y <: Q)_P.t : \{P\}\ \forall(Y <: Q).T}\ \text{(C-Q-ABS)}$$

Fig. 17. Typing rules for System $\mathsf{F}_{<:\mathsf{QC}}$

Syntax – Tracking Variables. Figure 16 defines the syntax of System $\mathsf{F}_{<:\mathsf{QC}}$. To reflect the underlying term-variable-based nature of capture tracking, term bindings in System $\mathsf{F}_{<:\mathsf{QC}}$ 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 $\lambda(x)_\bot.x$ would be given the type $\{\bot\}\ (x : \{Q\}\ S) \to \{x\}\ S$ 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 $\mathsf{F}_{<:\mathsf{QC}}$ now requires a qualifier argument to be substituted for that variable in qualifier position. As such, term application in System $\mathsf{F}_{<:\mathsf{QC}}$ 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 $\mathsf{F}_{<:\mathsf{QC}}$ 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-REFL-TVAR) and (SQ-TVAR). Rule (SQ-REFL-TVAR) accounts for reflexivity in System $\mathsf{F}_{<:\mathsf{QC}}$'s adjusted subqualification judgment. (SQ-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-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-VAR)!

$$\frac{\texttt{takeOverTheWorld} : \texttt{pandora}\ \texttt{Unit => Unit} \in \Gamma \qquad \Gamma \vdash \texttt{pandora} <: \texttt{pandora}}{\Gamma \vdash \texttt{takeOverTheWorld} <: \texttt{pandora}}$$

Subtyping. As function binders introduce a qualifier variable, so do function types as well; for example, x in $(x : \{Q\}\ S) \to \{x\}\ S$. 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 $\mathsf{F}_{<:\mathsf{QC}}$, using similar techniques as Lee et al. [2023].

Theorem 3.5 (Preservation for System $\mathsf{F}_{<:\mathsf{QC}}$). *Suppose $\Gamma \vdash s : T$, and $s \longrightarrow t$. Then $\Gamma \vdash t : T$ as well.*

Theorem 3.6 (Progress for System $\mathsf{F}_{<:\mathsf{QC}}$). *Suppose $\varnothing \vdash s : T$. Either s is a value, or $s \longrightarrow t$ 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 $\mathsf{F}_{<:\mathsf{QC}}$). *Let Γ be an environment and v be a value such that $\Gamma \vdash v : \{Q\}\ S$. Then $\Gamma \vdash \left\{\bigvee_{y \in \mathsf{fv}(v)} y\right\}$ <: Q.*

4 MECHANIZATION

The mechanization of System $\mathsf{F}_{<:\mathsf{Q}}$ (from Section 2.3), its derived calculi, System $\mathsf{F}_{<:\mathsf{QM}}$, System $\mathsf{F}_{<:\mathsf{QA}}$, and System $\mathsf{F}_{<:\mathsf{QC}}$, (from Section 3), and extended System $\mathsf{F}_{<:\mathsf{Q}}$ (from Section 2.7), is derived from the mechanization of System $\mathsf{F}_{<:}$ 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, similar to Negri and von Plato [2002]'s paper proof, though we note that Galatos [2023] independently give a 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 `mutable Box[Int]` as well as `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 `inplace_map` which maps a function over a reference cell?

```
case class Box[X](var elem: X)
// Is this well formed?
def inplace_map[X](r: mutable Box[X], f: const X => X): Unit = {
    r.elem = f(r.elem);
}
```

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:

```
case class Box[Q, X](var elem: Q X)
def inplace_map[Q, X](r: mutable Box[{Q} X], f: const X => Q 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:

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

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 X` 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:

```
def inplace_map[Xq, Xs](r: mutable Box[{Xq} Xs], f: {const | Xq} 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 with free variables. 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* $\forall\overline{Y}/C.T$ for some vector of qualifier variables $\overline{Y}$ 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., $X <: Y_1 \wedge Y_2$ is an invalid constraint), nor are constraints expressible in their source syntax for qualifier-polymorphic function terms.

While type qualifiers were only formalized with Foster et al.'s work, type qualifiers themselves were already quite popular by then. For example, `const` and `volatile` were already in use in C at that time [Kernighan and Ritchie 1988]. Additionally, the *Clean* programming language modelled *uniqueness* as a type qualifier with support for polymorphism by *constrained uniqueness schemes* by 1993 [Barendsen and Smetsers 1996]. The work on Clean predates Foster et al. [1999] and uses different language (type *attributes*), but it is striking how Clean's *uniqueness schemes* are essentially Foster et al.'s type schemes but specialized to uniqueness as a type qualifier.

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 $a \vee b \vee c$, and their set-merge substitution operator $\{a, b, c\}[a \mapsto \{d, e\}] = \{d, e, b, c\}$, is just substitution for free lattice formulas – $(a \vee b \vee c)[a \mapsto (d \vee e)] = (d \vee e) \vee b \vee c$. 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 $\overline{\alpha}$ and Boolean variables $\overline{\beta}$ over a pair of simple type S and Boolean formula (S, ϕ), where values of a qualified type (S, ϕ) are nullable if and only if ϕ evaluates to true.[4] Boolean formulas form a Boolean algebra, and Boolean algebras are just complemented distributive lattices, so Boolean formulas over a set of variables $\overline{\beta}$ are just free complemented distributive lattices generated over variables in $\overline{\beta}$. In this sense, we can view Madsen and van de Pol [2021] as an ML-polymorphism

[3] https://github.com/TiarkRompf/reachability/tree/main/base/lambda_star_syntactic.

[4] Technically they model a triple (S, ϕ, γ) where γ is another Boolean formula which evaluates to true if values of type (S, ϕ, γ) are non-nullable.

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 F$_{<:Q}$. Polymorphism is possible over *both* mutability qualifiers and simple types in Gordon's system, but must be done separately, as in 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:

```
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:

```
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. This provides an alternate encoding of the free lattice of qualifiers using the free lattices of types under subtyping.

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 `@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`.

```
async<A> fn read_to_string(reader: &mut impl Read * A)
  -> std::io::Result<String> { ... }
```

This is easy to see sound using similar ideas to our proof of simplified System $\mathsf{F}_{<:\mathsf{Q}}$. One would extend simple System $\mathsf{F}_{<:\mathsf{Q}}$ with a binder for each qualifier category instead of using the product lattice in extended System $\mathsf{F}_{<:\mathsf{Q}}$. However this proposal has proven controversial due to its syntactic overhead.

OCaml. The OCaml community [Slater 2023] 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 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-Callε 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, ϕ); it would be sensible to say $(S, \phi) <: (S, \phi')$ if $\phi \implies \phi'$. They conjecture that such a subtyping system would be sound. While we cannot answer this conjecture definitively, as we only model free lattices, 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 $\mathsf{F}_{<:\mathsf{Q}}$'s subtyping rules are syntax-directed and admit algorithmic rules, but it is not so easy to see if extended System $\mathsf{F}_{<:\mathsf{Q}}$ admits algorithmic subtyping rules. The difficulty is that extended System $\mathsf{F}_{<:\mathsf{Q}}$ needs two new non-syntax directed rules (sq-eval-elim) and (sq-eval-intro) to handle transitivity through base lattice elements. It remains an open question whether extended System $\mathsf{F}_{<:\mathsf{Q}}$ admits algorithmic subtyping rules. We conjecture that algorithmic subtyping rules could exist for a particular instantiation of extended System $\mathsf{F}_{<:\mathsf{Q}}$ 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 each lattice in the product admits algorithmic subtyping, then we think that 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 `nullable` as a qualifier, flow sensitivity is a natural addition to many qualifier systems. One often checks if a variable `x` is `NULL` with an `if` statement, with `x` qualified `nullable` in the branch that fails the test and `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 $\mathsf{F}_{<:\mathsf{Q}}$. 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 $\mathsf{F}_{<:}$ can be extended using this structure by constructing such an extension System $\mathsf{F}_{<:\mathsf{Q}}$, 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 helpful feedback. 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

Scott Aaronson. 2002. Polynomial Hierarchy Collapses: Thousands Feared Tractable. https://scottaaronson.com/writings/phcollapse.pdf

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 249–272.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 3–15. https://doi.org/10.1145/1328438.1328443

Yuyan Bao, Guannan Wei, Oliver Bračevac, Yuxuan Jiang, Qiyang He, and Tiark Rompf. 2021. Reachability types: tracking aliasing and separation in higher-order functional programs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 139 (oct 2021), 32 pages. https://doi.org/10.1145/3485516

Erik Barendsen and Sjaak Smetsers. 1996. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Math. Struct. Comput. Sci.* 6, 6 (1996), 579–612. https://doi.org/10.1017/S0960129500070109

Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4, Article 21 (nov 2023), 52 pages. https://doi.org/10.1145/3618003

John Boyland. 2006. Why we should not add readonly to Java (yet). *J. Object Technol.* 5, 5 (2006), 5–29. https://doi.org/10.5381/JOT.2006.5.5.A1

Walter Bright, Andrei Alexandrescu, and Michael Parker. 2020. Origins of the D Programming Language. *Proc. ACM Program. Lang.* 4, HOPL, Article 73 (jun 2020), 38 pages. https://doi.org/10.1145/3386323

Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. 1991. An Extension of System F with Subtyping. In *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings (Lecture Notes in Computer Science, Vol. 526)*, Takayasu Ito and Albert R. Meyer (Eds.). Springer, 750–770. https://doi.org/10.1007/3-540-54415-1_73

Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Pittsburgh, PA, USA) *(AGERE! 2015)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (mar 1966), 143–155. https://doi.org/10.1145/365230.365252

Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 681–690. https://doi.org/10.1145/1985793.1985889

Stephen Dolan. 2016. *Algebraic subtyping*. Ph. D. Dissertation.

Vlastimil Dort and Ondřej Lhoták. 2020. Reference Mutability for DOT. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 18:1–18:28. https://doi.org/10.4230/LIPIcs.ECOOP.2020.18

Mattias Felleisen and D. P. Friedman. 1987. A Calculus for Assignments in Higher-Order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 314. https://doi.org/10.1145/41625.41654

Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 192–203. https://doi.org/10.1145/301618.301665

Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/512529.512531

Nikolaos Galatos. 2023. Decidability of Lattice Equations. https://doi.org/10.1007/s11225-023-10063-4

Isaac Oscar Gariano, James Noble, and Marco Servetto. 2019. Callε: an effect system for method calls. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, Hidehiko Masuhara and Tomas Petricek (Eds.). ACM, 32–45.

https://doi.org/10.1145/3359591.3359731

Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1, Article 4 (apr 2021), 79 pages. https://doi.org/10.1145/3450272

Colin S. Gordon, Werner Dietl, Michael D. Ernst, and Dan Grossman. 2013. Java UI : Effects for Controlling UI Object Access. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, 179–204. https://doi.org/10.1007/978-3-642-39038-8_8

Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and Reference Immutability for Safe Parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 21–40. https://doi.org/10.1145/2384616.2384619

James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition* (1st ed.). Addison-Wesley Professional.

Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. ReIm and ReImInfer: Checking and Inference of Reference Immutability and Method Purity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 879–896. https://doi.org/10.1145/2384616.2384680

Peter Jipsen. 2001. A Gentzen system and decidability for residuated lattices. *Preprint* (2001). https://www1.chapman.edu/~jipsen/reslat/gentzenrl.pdf

Paul A. Karger and Andrew J. Herbert. 1984. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *1984 IEEE Symposium on Security and Privacy*. 2–2. https://doi.org/10.1109/SP.1984.10001

Brian W. Kernighan and Dennis Ritchie. 1988. *The C Programming Language, Second Edition*. Prentice-Hall. https://en.wikipedia.org/wiki/The_C_Programming_Language

Edward Lee and Ondřej Lhoták. 2023. Simple Reference Immutability for System $F_{<:}$. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 252, 25 pages. https://doi.org/10.1145/3622828

Edward Lee, Kavin Satheeskumar, and Ondřej Lhoták. 2023. Dependency-Free Capture Tracking. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs*. Seattle, WA. https://doi.org/10.1145/3605156.3606454

Edward Lee, Yaoyu Zhao, Ondřej Lhoták, James You, Kavin Satheeskumar, and Jonathan Immanuel Brachthäuser. 2024a. *Artifact for the OOPSLA 2024 paper 'Qualifying System F-sub'*. https://archive.softwareheritage.org/swh:1:snp:25948423337bcc31981da471b67258ff572a5585

Edward Lee, Yaoyu Zhao, Ondřej Lhoták, James You, Kavin Satheeskumar, and Jonathan Immanuel Brachthäuser. 2024b. *Artifact for the OOPSLA 2024 paper 'Qualifying System F-sub'*. https://doi.org/10.1145/3580431

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (jun 2014), 100–126. https://doi.org/10.4204/eptcs.153.8

Daan Leijen and Ross Tate. 2010. *Convenient Explicit Effects using Type Inference with Subeffects*. Technical Report MSR-TR-2010-80. https://www.microsoft.com/en-us/research/publication/convenient-explicit-effects-using-type-inference-with-subeffects/

Daniel Leivant. 1983. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 88–98. https://doi.org/10.1145/567067.567077

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *Proc. ACM Program. Lang.* 7, ICFP, Article 204 (aug 2023), 28 pages. https://doi.org/10.1145/3607846

Magnus Madsen and Jaco van de Pol. 2021. Relational Nullable Types with Boolean Unification. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 110 (oct 2021), 28 pages. https://doi.org/10.1145/3485487

Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. 2010. JavaCOP: Declarative pluggable types for java. *ACM Trans. Program. Lang. Syst.* 32, 2, Article 4 (feb 2010), 37 pages. https://doi.org/10.1145/1667048.1667049

Jens Maurer. 2015. P0012R1: Make exception specifications be part of the type system, version 5. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0012r1.html

Sara Negri and Jan von Plato. 2002. Permutability of rules in lattice theory. *algebra universalis* 48, 4 (01 Dec 2002), 473–477. https://doi.org/10.1007/s000120200012

Bob Nystrom. 2015. What Color is Your Function? https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/

Martin Odersky, Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, and Ondřej Lhoták. 2021. Safer Exceptions for Scala. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Scala* (Chicago, IL, USA) *(SCALA 2021)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3486610.3486893

Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory Pract. Object Syst.* 5, 1 (1999), 35–55.

Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. 2016. Gentrification gone too far? affordable 2nd-class values for fun and (co-)effect. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 234–251. https://doi.org/10.1145/2983990.2984009

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 201–212. https://doi.org/10.1145/1390630.1390656

Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D. Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). Lecture Notes in Computer Science, Vol. 7850. Springer, 233–269. https://doi.org/10.1007/978-3-642-36946-9_9

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP 2012 – Object-Oriented Programming*, James Noble (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–282.

Thoralf Skolem. 1920. Logisch-Kombinatorische Untersuchungen Über Die Erfüllbarkeit Oder Bewiesbarkeit Mathematischer Sätze Nebst Einem Theorem Über Dichte Mengen. In *Selected Works in Logic*, Thoralf Skolem (Ed.). Universitetsforlaget.

Max Slater. 2023. Oxidizing OCaml: Rust-Style Ownership. https://blog.janestreet.com/oxidizing-ocaml-ownership/

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

Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 211–230. https://doi.org/10.1145/1094811.1094828

Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proc. ACM Program. Lang.* 8, POPL, Article 14 (jan 2024), 32 pages. https://doi.org/10.1145/3632856

Philip M. Whitman. 1941. Free Lattices. *Annals of Mathematics* 42, 1 (1941), 325–330. http://www.jstor.org/stable/1969001

Yoshua Wuyts, Oli Scherer, and Niko Matsakis. 2022. Announcing the keyword generics initiative: Inside rust blog. https://blog.rust-lang.org/inside-rust/2022/07/27/keyword-generics.html

Anxhelo Xhebraj, Oliver Bračevac, Guannan Wei, and Tiark Rompf. 2022. What If We Don't Pop the Stack? The Return of 2nd-Class Values. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:29. https://doi.org/10.4230/LIPIcs.ECOOP.2022.15

Yaoyu Zhao. 2023. *Adding Reference Immutability to Scala.* Master's thesis. http://hdl.handle.net/10012/19601

Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. 2007. Object and Reference Immutability Using Java Generics. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/1287624.1287637

Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. 2010. Ownership and immutability in generic Java. In *OOPSLA 2010, Object-Oriented Programming Systems, Languages, and Applications*. Reno, NV, USA, 598–617.

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