Polymorphic Type Qualifiers

by

Edward Lee

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Jeff Foster Professor, Department of Computer Science, Tufts University
Supervisor:	Ondřej Lhoták Professor, School of Computer Science, University of Waterloo
Internal Member:	Yizhou Zhang Assistant Professor, School of Computer Science, University of Waterloo
Internal Member:	Prabhakar Ragde Professor, School of Computer Science, University of Waterloo
Internal-External Member:	Werner Dietl Associate Professor, Department of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is the result of my collaborations with my wonderful colleagues and coauthors: Ondřej Lhoták, Yaoyu Zhao, Jonathan Brachthäuser, James You, Kavin Satheeskumar, Jonathan Lindegaard Starup, and Magnus Madsen. The work in this thesis draws upon the following publications and associated proof artifacts, for which references are found below. The complete collection of proof artifacts can also be found at https://github.com/e45lee/phd-thesis-proofs.

1. Edward Lee, Ondřej Lhoták, Jonathan Lindegaard Starup, and Magnus Madsen. *Qualified Types for Boolean Algebras.* Proceedings of the ACM on Programming Languages, Volume 9, OOPSLA2, Article 318. October 2025. In: [53].

2. Edward Lee, Yaoyu Zhao, Ondřej Lhoták, James You, Kavin Satheeskumar, and Jonathan Brachthäuser. *Qualifying System $F_{<:}$.* Proceedings of the ACM on Programming Languages, Volume 8, OOPSLA1, Article 115. April 2024. In: [54].

3. Edward Lee, Ondřej Lhoták. *Simple Reference Immutability for System $F_{<:}$.* Proceedings of the ACM on Programming Languages, Volume 7, OOPSLA2, Article 252. October 2023. In: [51].

4. Edward Lee, Kavin Satheeskumar, and Ondřej Lhoták. *Dependency-free Capture Tracking.* In: Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs (FTfJP '23), July 18, 2023, Seattle, WA, USA. ACM, New York, NY, USA, 5 pages. In [52].

I understand that my thesis may be made electronically available to the public.

Abstract

Type qualifiers offer a simple yet effective mechanism for extending existing type systems to deal with additional constraints or safety requirements. For example, the `const` qualifier is a popular mechanism for annotating existing types to signify that the value in question is read-only in addition. A variable of type `const int` is both an integer and also cannot be written to.

While type qualifiers themselves are well-studied, polymorphism over type qualifiers remains an area less well examined. This has led to a number of ill-desired outcomes. For one, many practical systems implementing type qualifiers in their type systems simply ignore their interaction with generic types. Other systems implement polymorphism with seemingly unique and ad-hoc rules for dealing with qualifiers.

In this thesis, we show that this does not need to be the case. We start by examining three well-known qualifier systems: systems for tracking immutability, function colour, and captured variables, and show that despite their differences that they share surprising common structure. We then give a design recipe, inspired by that structure, using the mathematical structure of free lattices for modelling polymorphism over type qualifiers, to give a framework for polymorphic type qualifiers. We then show that our design recipe precisely captures this structure by recasting those three existing systems in our framework for qualifier polymorphism by free lattices. Finally, we extend type qualifiers from ranging over lattices to type qualifiers ranging over Boolean algebras, which we then use to extend an existing effect system with *effect exclusion* to support subeffecting as well via subqualification and subtyping.

Acknowledgements

I am indebted to a number of individuals for whom this thesis would not have been possible.

For my readers, Werner, Jeff, and Yizhou: thank you for reading this thesis, and for your thoughtful comments as well. For Prahbakar: thank you for reading this thesis, as well as for your thoughtful comments, but especially for your dedication to teaching and outreach, without which I do not think I would have started on this endeavour at all.

I would not started this thesis without inspiration from my professors in my undergraduate and my master's to inspire me, so to Gord, Ian, Costis, and Jim: thank you for believing in me.

Research does not happen in a vacuum, and I owe a lot to my collaborators and labmates for both the interesting research conversations we have shared as well as the support they have provided. To Raffi: thank you for your mentorship through the years. For Jonathan: thank you for your help, advice, and interesting discussions on effect systems. For James: thank you for being a friendly face in unfamiliar places, and for all your help navigating research and the PhD. To Magnus: thank you for your help and advice and your expertise on Boolean algebras as well. To Jack: thank you for always being there to bounce ideas off, both in the lab and outside. Finally, to Kavin, Cong, August, Yaoyu, Amir, Jianlin, and Avish: thank you for putting up with my ideas as well :)

To my friends: thank you for being there for me, and for supporting me in trying times. To Magnus and Daniela, Olga and Kirill, Christa, Jack, Avish, Eric, Alena, Manya, Brian, Ross, Zahra and Sam, Kris and Rebecca, Saveeta, Ifaz, Abel and Marianna, Rafael, and Brad: thank you especially so for being there during especially trying times![1]

This thesis and I owe a lot to my mother and my sister, who have always been behind me every step of the way, even on my more outlandish ventures. This thesis would not be possible without your help, and for that I am eternally grateful.

Finally, to Ondřej: I am indebted to you for seeing me through this thesis from start to finish. You have been a wonderful source of advice, and have helped me grow and flourish as an independent researcher. You have been generous with your support, kindness, advice and time through this process. For this, I am in your debt, and I offer you my respect, trust, and friendship.[2]

[1] I owe you lunch for all the help you all have been, and if you read this part of the thesis you can claim it! Or some root beer and butter tarts for you, Magnus.

[2] And also lunch, at Prague by the Scarborough Golf Course.

Table of Contents

List of Figures

List of Tables

Chapter 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?

There are at least two ways to address this. We can view the modification of the 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 [75] 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 thesis, we adopt the latter view.

One such system is that of *Type Qualifiers* [31], 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, we know that this version of `toLowerCase` cannot change its input; 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. [31] 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. Because of this natural subtyping, type qualifiers are both *optional* and *opt-in*, so existing code will still work without the additional annotation of qualifiers. Here, concretely, unqualified values and variables remain `mutable`, so existing code which relies on this behaviour still compiles and runs. Further, new code can *opt-in* to safer behaviour by qualifying new values and variables with `const`. Finally, existing code with unqualified mutable values can call into new code which use `const` to enforce safety as `mutable` is subqualified by `const`. As Bracha [13] noted, type systems which are opt-in ameliorate the costs that come with forcibly re-typing (here, re-qualifying) existing code to work with a new (here, qualifier-extended) type system.

Type qualifiers are themselves also naturally composable, as the direct product of two lattices is itself a third lattice that embeds the order of the two original lattices. This composability also allows for multiple qualifier systems to exist as *pluggable* [13] extensions in

harmony to a base type system in a single language, making them an attractive framework for adding additional checks to an existing type system.

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 [87, 15, 89]. Other languages, such as OCaml and Rust, are exploring more exotic qualifiers to encode properties like locality, linearity, exclusivity, and synchronicity [85, 92]. 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 [72, 64] for doing so, which have been used to model extensions of Java for checking *nullability* [72], *energy consumption* [81], and *determinism* [72] amongst others.

While type qualifiers themselves are well-explored, *qualifier polymorphism* is still understudied. In many cases it is not necessary. 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 = {...}
```

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 [27, Chapter 4.2.1]. 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.

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)
```

1.1 Existing Work

Existing qualifier systems have limited support for these use cases. Foster et al. [31]'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.

Many generic implementations of qualifier systems, such as Papi et al. [72]'s Checker Framework and derivatives thereof, such as team [88], present a restricted, implicit view of qualifier polymorphism via Java type annotations. Qualifier polymorphism in this setting is enabled through type polymorphism, but re-qualifying a polymorphic type variable simply overrides any existing qualifier when that type variable is instantiated. For example, the following fragment would not type under Papi et al. [72]'s system.

```
def unsound[T](x: T): @mutable T = x
```

Here, the qualifier annotation `@mutable` simply strips away any existing qualifier annotated on `T`. So if we instantiated `unsound` on an `@immutable` value:

```
val str: @readonly String = "Hello World"
val strm /*: @mutable String*/ = unsound[@readonly String](str)
```

we would get back a `@mutable` value, which is unsound and undesirable.

Some systems, such as Gordon et al. [38] do support explicit qualifier polymorphism, but they partially ignore the interaction between combinations of qualifier variables and their bounds. While they support explicit qualifier polymorphism in their calculus, *combinations of qualifier variables* are left as a black box. Here, in this example, drawn from Gordon et al. [38]'s reference immutability system, reading a reference from a record returns a new reference which is only mutable if it was qualified to be mutable and the enclosing record was also qualified to be mutable as well. Hence, the mutability of the result `r.f` is the combination of the initial mutability qualifiers `QR` and `QX`, denoted `QR ~> QX`. This will be `@mutable` if both `QR` and `QX` are instantiated at runtime with `@mutable`, and `@readonly` otherwise.

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

However, there is no mechanism in Gordon et al. [38]'s system for reasoning about the relationship between `QR`, `QX`, and `QR ~> QX`.

Other systems present application-specific *subqualification* semantics seen in Boruch-Gruszecki et al. [9] or Wei et al. [90]. And as we will examine in Chapter 2, even seemingly unrelated systems share similar demands on their qualifiers; one often needs polymorphism

and the ability to reason about subtypes in the presence of polymorphic qualifiers. Here, for example, in a system for capturing or reachability as seen in Boruch-Gruszecki et al. [9] and Wei et al. [90], one ought to be able to reason that the set of values reachable from the result of `triple` is strictly contained by the unions of the set of values reachable from its arguments `x`, `y`, and `z`. Here, {*} denotes that the parameters `x`, `y`, and `z` can be instantiated with arguments that can reach any other possible value.

```
def triple[X, Y, Z](x: {*} X, y: {*} Y, z: {*} Z) = (x, y, z)
```

1.2 This thesis

This thesis is concerned with providing a principled account of how polymorphism interacts with type qualifiers. We demonstrate that lattice-based type qualifier systems can be extended with *free lattices* to add polymorphism free in a principled manner using ideas grounded in universal algebra. We recast existing qualifier systems that take an ad-hoc approach to qualifier polymorphism using our work to demonstrate that free lattices provide both an universal and correct algebraic structure for handling the polymorphism seen in existing systems. We then build on our work with free lattices to add support for distributivity and negation, building a polymorphic qualifier system grounded on Boolean algebras, and give a use case in the form *effect exclusion* to demonstrate its usefulness. In short, we show that polymorphism and type qualifier systems can be combined in an elegant and mathematically universal fashion without sacrificing the ergonomic extensibility and usability of existing type qualifier systems.

1.2.1 Common Traits

To this end, we start by examining how type qualifiers (and qualifier-like) systems are used to model existing problems in programming language design. We examine existing work in a well-studied traditional type qualifier system, that of *reference immutability*, to see how many systems handle the interaction between type qualifiers and polymorphism.

While all of these systems are built on Foster et al. [31]'s formalization of type qualifiers via lattices, they all take unique approaches for how they deal with polymorphism in the presence of the subtyping that qualifiers induce, including:

- Not supporting polymorphism over qualifiers at all, such as Tschantz and Ernst [89], or

- Supporting a restricted form of polymorphism which is not as general as full parametric polymorphism via qualifier variables, such as Dietl et al. [24], or
- Supporting polymorphism via qualifier variables but only supporting a restricted form of subtyping that type qualifiers give via the lattice order, such as Gordon et al. [38].

This situation is clearly not ideal. Moreover, there are other systems that use qualifier-like approaches to model extensions to their type system as well which themselves deal with the problem that polymorphism poses. Of particular interest to us are recent systems for tracking capture and tracking effects, which tag existing value and function types with additional information for recording the set of variables they capture and the set of effects they use by the set of functions they call.

While these systems do not at first glance appear to be qualifier systems (strictly speaking), we show that they share some common algebraic structures with traditional qualifier systems. However, all examined systems give wildly different and unique rules for accounting for variables in the presence of polymorphism, which is undesirable.

1.2.2 Shared Structure

However, not all is lost. There is a natural algebraic structure for accounting for variables within the lattice-based qualifier framework of Foster et al. [31], which naturally handles the interplay between polymorphic qualifier variables and the lattice-based ordering of type qualifiers. Based on Whitman [91]'s *free lattices*, we give a natural extension of System $\mathtt{F}_{<:}$ using the ideas of Foster et al. [31], for a base polymorphic calculus System $\mathtt{F}_{<:\mathtt{Q}}$ with a principled treatment of type qualifiers with *qualifier variables* which we can remix in other applications. This we do using a well-known folklore representation of free lattices using deduction rules [67] that neatly slots into the deduction rules for both subtyping and typing for base System $\mathtt{F}_{<:}$.

1.2.3 Revisiting Applications

Armed with our base calculus System $\mathtt{F}_{<:\mathtt{Q}}$, we show how existing qualifier systems can be remixed with principled rules from System $\mathtt{F}_{<:\mathtt{Q}}$ for handling the interaction between qualifiers, variables, and subtyping. We do this in the three application areas we examined earlier: reference immutability, capture tracking, and exception tracking. To do so, we

first show how capture sets and effect sets seen in capture tracking and exception tracking are better seen as *lattice expressions*. We then show how the lattice qualifiers of all three systems can be extended with the free lattice rules for supporting polymorphism from System $\mathtt{F}_{<:\mathtt{Q}}$ to give simple, principled, parametric polymorphic calculi based on System $\mathtt{F}_{<:\mathtt{Q}}$ that express immutability, capture tracking, and exception tracking soundly, that reuse much of their proofs despite having unique operational semantics for each application.

1.2.4 Expressive Extensions

Finally, we show how type qualifiers based on lattices can be neatly generalized to support more expressive algebraic structures. We show how lattices in System $\mathtt{F}_{<:\mathtt{Q}}$ can be generalized to handle distributive lattices and Boolean algebras with the same desirable ordering/subtyping properties while retaining a principled treatment of qualifier variables and polymorphism, and apply it to an existing effect exclusion system implemented in Flix to give it support for subeffecting as well. This we do via recasting Boolean algebras as a lightweight order-theoretic extension of lattices with distributivity and negation, and in turn extending the deduction rules for System $\mathtt{F}_{<:\mathtt{Q}}$ with rules for distributivity and negation in a calculus System $\mathtt{F}_{<:\mathtt{B}}$.

1.3 Contributions and Organization

This thesis is organized as follows. In Chapter 2 we present two qualifier systems: one for reference immutability [89], and one for capture tracking [10] to illustrate two ad-hoc qualifier systems that share some common algebraic structure in their qualifier systems. Next, in Chapter 3 we illustrate how this common algebraic structure can be formalized using the free lattice theory of lattices ([91]). We then recast the two systems we presented in Chapter 2 with our newfound formalism from Chapter 3 in Chapter 4. We then show how type qualifiers grounded on (free) lattices can be generalized to qualifiers grounded on more interesting algebraic structures (free Boolean algebras), and how they can be used to model effect systems with support for effect exclusion in Chapter 5. We survey related work in Chapter 6, and wrap up with future work and a conclusion in Chapters 7 and 8.

Chapter 2

Background – Motivation

As Foster et al. [31] pointed out, type qualifiers provide an elegant, lightweight approach for augmenting existing type systems for tracking additional information – qualifiers like `const`, for example. However, their implicit and predicative system for qualifier polymorphism has led to many ad-hoc approaches for dealing with qualifier-like variables in type system extensions which also need to support both predicative and higher-rank polymorphism.

Here, in this chapter, we examine three such systems which augment existing types for tracking immutability, capture, and effects. At face value these systems appear to be unrelated; some do not even claim to be qualifier systems at all! However, they share many common design similarities with each other and with Foster et al. [31]'s original system as well. Where they differ *subtly*, as we will show, is in how they handle interaction between polymorphism and qualifier-like terms. These differences, though, are minor enough that one would suspect there is some common structure underlying all – motivating our presentation of polymorphic type qualifiers in Chapter 3.

2.1 Reference Immutability

We start by examining *reference immutability* [89, 43], a technique for controlling mutation in an otherwise impure language. In an impure language, mutation arises naturally, but can cause bugs which can be hard to untangle; for example, two modules which at first glance are completely unrelated may interact through some shared mutable variable. Taming – or controlling – where and how mutation can occur can reduce these issues.

In this setting, the type of each reference to a value can be either mutable or immutable. An immutable reference cannot be used to mutate the value or any other values transitively reached from it.

Reference immutability has long been studied in existing object-oriented programming languages such as Java [89, 43, 96] and C# [38]. In addition, many programs in impure functional languages tend not to mutate their data structures [41]. Recently, work has been done to study reference immutability in this context, in particular by Dort and Lhoták [28].

2.1.1 Motivating Reference Immutability

Immutable values are crucial even in impure functional programming languages because pure code is often easier to reason about. This benefits both the programmer writing the code, making debugging easier, and the compiler when applying optimizations.

Although most values, even in impure languages, are immutable by default [41], mutable values are sometimes necessary for various reasons. For example, consider a compiler for a pure, functional, language. Such a compiler might be split into multiple passes, one that first builds and generates a symbol table of procedures during semantic analysis, and one that then uses that symbol table during code generation. For efficiency, we may wish to build both the table and the procedures in that table with an impure loop.

```
object analysis {
    class Procedure(name : String) {
        val locals : mutable.Map[String, Procedure] = mutable.Map.empty
        def addLocalProcedure(name: String, proc: Procedure) = {
            locals += (name -> proc)
        }
    }

    val table : mutable.Map[String, Procedure] = mutable.Map.empty

    def analyze(ast: AST) = {
        ast.forEach((node) => { table.add(node.name, new Procedure(...)) })
    }
}
```

The symbol table and the properties of the procedure should not be mutable everywhere, though; during code generation, our compiler should be able to use the information

in the table to generate code, but it shouldn't be able to change the table nor the information in it! How do we enforce this, though?

One solution is to create an immutable copy of the symbol table for the code generator, but this can be fragile. A naive solution that merely clones the table itself will not suffice, for example:

```
object analysis {
    private val table[analysis] = ...
    def symbolTable : Map[String, Procedure] = table.toMap
    // create immutable copy of table.
}

object codegen {
    def go() = {
        analysis.symbolTable["main"].locals += ("bad" -> ...) // whoops...
    }
}
```

While this does create an immutable copy of the symbol table for the code generator, it does not create immutable copies of the procedures held in the table itself. We would need to recursively rebuild a new, immutable symbol table with new, immutable procedures to guarantee immutability, which can be expensive, both in terms of code size and in terms of running time.

Moreover, creating an immutable copy might not even work in all cases. Consider an interpreter for a pure, functional language with support for `letrec x := e in f`. The environment in which e is interpreted contains a cyclic reference to x, which necessitates mutation in the interpreter. Without mutation, or special tricks using laziness, this sort of structure cannot be constructed, let alone copied.

```
abstract class Value { }
type Env = Map[String, Value]
case class Closure(var env: Env,
    params: List[String],
        body: Exp) extends Value

def interpret_letrec(env: Env, x: String, e: Exp, f: Exp) : Value = {
    val v = interpret(env + (x -> Nothing), e)
```

```
    case v of {
        Closure(env, params, body) => v.env = v.env + (x -> v)
        // Update binding
    }
    interpret (env + (x -> v), f)
}
```

Here, the closure that `v` refers to needs to be mutable while it is being constructed, but since the underlying language is pure, it should be immutable afterwards. In particular, we should not be able to mutate the closure through the self-referential reference `v.env = env + (x -> v)`, nor should we be able to mutate the closure while interpreting `f`.

We would like a system that prevents writes to `v` from the self-referential binding in its environment and from the reference that we pass to `interpret (env + (x -> v), f)`. This is what *reference immutability* provides; an annotation `@readonly` which transitively protects the references it qualifies from being written to.

```
abstract class Value { }
type Env = Map[String, @readonly Value]
case class Closure(env: var Env, params: List[String], body: Exp)

def interpret_letrec(env: Env, x: String, e: Exp, f: Exp) : Value = {
    val v = interpret(env + (x -> Nothing), e)
    case v of {
        Closure(env, params, body) => v.env = env + (x -> v)
        // update binding
    }
    interpret (env + (x -> v), f)
}
```

Here, as `Env` maps to `@readonly Values`, `interpret` cannot mutate said values through its `env`.

2.1.2 Reference Immutability

At its core, however, regardless of language, *reference immutability* is concerned with two key ideas:

- **Read-only references:** References to values can be made read-only, so that the underlying value **cannot** be modified through that reference.
- **Transitive immutability:** An immutable reference to a *compound value* that contains other references cannot be used to obtain a mutable reference to another value. For example, if `x` is a read-only reference to a pair, the result of evaluating `x.first` should be *viewpoint adapted* [24] to be a read-only reference, even if the pair contains references that are otherwise mutable.

This requires that existing types be augmented with additional information qualifying whether a value is read-only or not – a natural application of Foster et al. [31]'s type qualifier system. For example, consider the following snippet of Scala-like code that deals with mutable pairs.

```
case class Pair[X](var first: X, var second: X)

def good(x : Pair[Int]) = { x.first = 5 }
def bad1(y : @readonly Pair[Int]) = { y.first = 7 }
def bad2(y : @readonly Pair[Pair[Int]]) = { y.first.first = 5 }
def access(z: @readonly Pair[Pair[Int]]): @readonly Pair[Int] =
    { z.first }
```

Here, `@readonly` qualifies an existing type with the additional information that instances of that type are read-only. As an example, the function `good` is well-typed because it mutates the pair through an (unqualified) mutable reference `x`. However, we would disallow `bad1` because it mutates the pair through a read-only reference `y`. Moreover, we would also disallow `bad2` because it mutates the pair 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 Pair[Int]` to the first component of the pair referenced by `z`.

(Parametric) Polymorphism

Issues arise, however, when we introduce parametric polymorphism. Polymorphism is important in all languages but especially so in functional languages. The interaction of polymorphism and reference immutability raises interesting questions. Should type variables abstract over annotated types including their immutability qualifiers (such as `@readonly`

`String`), or only over the base types without immutability annotations (such as `String`)? Should uses of type variables admit an immutability annotation like other types do? For example, should `@readonly X` be allowed, where `X` is a type variable rather than a concrete type? If yes, then how should one interpret an annotated variable itself instantiated with an annotated type? For example, what should the type `@readonly X` mean if the variable `X` is instantiated with `@mutable String`?

Recall our earlier example – a polymorphic `Pair` object.

```
case class Pair[X](var first: X, var second: X)
```

In a functional language, it is only natural to write higher-order functions that are polymorphic over the elements stored in the pair. Consider an in-place map function over pairs, which applies a function to each element in the pair, storing the result in the original pair. This naturally requires mutable access to a pair.

```
def inplace_map[X](pair: Pair[X], f: X => X): Unit = {
    pair.first = f(pair.first);
    pair.second = f(pair.second);
}
```

This is all well and good, but we may wish to restrict the behaviour of `f` over the elements of the pair. It may be safer to restrict the behaviour of `f` so that it could not mutate the elements passed to it. Note that we cannot restrict access to the pair, however, as we still need to mutate it.

```
// Is this well founded?
def inplace_map[X](pair: Pair[X], f: (@readonly X) => X): Unit = {
    pair.first = f(pair.first);
    pair.second = f(pair.second);
}
```

Now, such a definition requires the ability to further modify type variables with immutability qualifiers. Existing systems take various approaches – we survey them here.

roDOT [28]: roDOT extends the calculus of Dependent Object Types [2] 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 $M \leq \bot$, and x is read-only if $M \geq \top$. This is akin to a type qualifier scheme where qualifiers are expressed in the language of types. As types in DOT support intersections, unions, and variables, this gives roDOT's mutability qualifiers the expressiveness from intersections, unions, and variables as well.

For example, `inplace_map` from before could be expressed in roDOT using an intersection type to modify immutability on the type variable `X`; `Any` denotes the $\top$ type in Scala syntax:

```
def inplace_map[X](Pair[X]: pair, f: (X & {M :> Any}) => X): Unit
```

As another example, one could write a generic `getF` function which reads a field `f` out of any record that has `f` as a field polymorphic over *both* the mutabilities of the record `x` and the field `f`:

```
def getF[T](x: {M: *, f : T}) : T & {M :> x.M} = x.f
```

Here, the return type of `getF` will give the proper, tightest, viewpoint-adapted type for reading `x.f` depending on both the mutabilities of `x` and `f`.

This all comes at a cost, however – in order to support this expressiveness on qualifiers, Dort and Lhoták [28] build a complex calculus on top of a nontrivial system, that of DOT itself [2].

Immutability for C# [38]: Another calculus which approaches this with a similar view as roDOT is the calculus of Gordon et al. [38]. Polymorphism is possible over *both* mutabilities and types in Gordon's system, but must be done separately; type variables quantify over base types that have not been qualified with some immutability annotation, whether that be read-only or mutable. Instead of carrying mutabilities as members of types, mutabilities and types come together as stratified pairs. The `inplace_map` function that we discussed earlier would be expressed with both a base-type variable as well as a mutability variable:

```
def inplace_map[M, X](Pair[M X]: pair, f: @readonly X => M X): Unit
```

Like roDOT, Gordon's system also allows for mutability annotations to be combined in types, in effect allowing viewpoint adaptation to be expressed at the type level using their mutability operator `~>`. Their mutability operator syntactically represents mutability viewpoint adaptation; for example, `getF` could be written as the following in Gordon's system, as `MS ~> MT` gives the viewpoint adapted qualifier that the result of reading `x.f` would have:

```
def getF[MS, MT, T, S <: {f : MT T}](x: MS S) : (MS ~> MT) T = x.f
```

`~>` also simplifies concrete mutability qualifiers: `@readonly ~> @mutable` simplifies to `@readonly`. However, unlike roDOT, which allows for inferences to be drawn about the mutability of the type `(T & {M :> x.M}).M` depending on the bounds on T and S, the only allowable judgment we can draw about `MS ~> MT` is that it can be widened to `@readonly`. We cannot conclude, for example, that `MS ~> MT <: M` in the following, even though both `MS <: M` and `MT <: M`:

```
def getF[M, MS <: M, MT <: M, T, S <: {f : MT T}](x: MS S):
    (MS ~> MT) T = x.f
```

In short, while one can use `~>` to represent a mutability viewpoint-adapted qualifier, one cannot use it in any subtyping relationships unless one concretely knew the two input qualifiers.

Javari [89]**:** Reference immutability was first modelled in the context of Java; Javari is the earliest such extension. In Javari's formalization, Lightweight Javari, type variables `X` stand in for either other type variables, class types, and `readonly`-qualified class types. Unlike roDOT and Gordon et al. [38], in Lightweight Javari type variables **cannot** be further qualified by the `readonly` type qualifier. Lightweight Javari, however, does support parametric mutability polymorphism for class types, but does not support parametric mutability polymorphism directly on methods. Instead, limited parametric mutability method polymorphism in Javari, denoted with the keyword `romaybe`, is desugared using overloading into the two underlying methods handling the read-only case and the mutable case replacing `romaybe` in the source. Our earlier example, `getF`, can be written using `romaybe` as follows:

```
class HasF<T> {
    T f;
    romaybe T getF() romaybe { return f; }
}
```

This is conceptually equivalent to, and is treated as such by Javari's extended Java compiler, the following code fragment:

```
class HasF<T> {
    T f;
    @readonly T getF() @readonly { return f; }
    T getF() { return f; }
}
```

However, while this fragment is expressible in Javari, it is inexpressible in the core calculus Lightweight Javari, as `@readonly T` is ill-formed in Lightweight Javari. As for safety, immutability safety is done in Lightweight Javari through a case analysis on how typed Lightweight Javari program terms can reduce. Tschantz and Ernst [89] claim that the soundness of Lightweight Javari reduces to showing the soundness of Lightweight Java, but no formal proof is given.

ReIm: [43]**:** ReIm simplifies Javari to enable fast, scalable mutability inference and analysis. Like Javari, ReIm supports two type qualifiers – `readonly` and `polyread`, where `readonly` marks a read-only type and `polyread` is an analogue of `romaybe` from Javari. Like Lightweight Javari, and unlike roDOT, ReIm restricts how qualifiers interact with generics. ReIm's polymorphism model is similar to that of Gordon et al. [38] – type variables range over unqualified types. However, ReIm has no mechanism for full mutability polymorphism, and therefore `getF` cannot be written in ReIm at all.

Immutability Generic Java: [96]**:** Immutability Generic Java is a scheme for expressing immutability using Java's existing generics system. The type `List<Mutable>` denotes a *mutable* reference to a List, whereas the type `List<Readonly>` denotes a *read-only* reference to a list. Viewpoint adaptation is not supported, and transitive immutability must be explicitly opted into. For example, in the following snippet, the field `value` of `C` is *always* mutable. Transitive immutability must be explicitly opted into by instantiating `List` with the immutability parameter `ImmutOfC`.

```
class C<ImmutOfC> {
    List<Mutable /* ImmutOfC for transitivity */, Int> value;
}
```

Moreover, transitive immutability cannot be expressed at all over fields given a generic type. By the nature of how immutability is expressed in IGJ, type variables range over fully qualified types, and there is no mechanism for re-qualifying a type variable with a new immutability qualifier. For example, the mutability of `value` in any `Box` below depends *solely* on whether or not `T` is mutable. Hence the *value* field of a `Box` is mutable even if it was read through a read-only `Box` reference – that is, a reference of type `Box<ReadOnly>`.

```
class Box<ImmutOfBox, T> {
    T value;
}

Box<Readonly, List<Mutable,Int>> b = new Box(...)
b.value.add(10); // OK -- even though it mutates the underlying List.
```

Conclusion

As we have seen, while the ideas behind type qualifiers serve well as the framework underlying many reference immutability systems, the additional demands of polymorphism and viewpoint adaptation over polymorphic values do not play well with existing theory. This has led to an undesirable proliferation of ad-hoc approaches to solving this issue, ranging from integrating qualifiers as types themselves to support full expressiveness, to more limited approaches which support polymorphism with restrictions.

2.2 Tracking Capture

Another system related to type qualifiers, in that it augments existing types with additional information, is the *capture calculus* System $\mathtt{CC}_{<:\Box}$ by Boruch-Gruszecki et al. [10], which aims to track side effects by tracking the capabilities [22] that can perform them.

By controlling access to capabilities, one controls when and where the corresponding effectful operations can be performed. This technique allows reasoning over effects; if a piece of code cannot access the capability corresponding to the effect, it cannot perform the effect. Moreover, capabilities are lightweight – as they are just values and extra parameters passed to effectful functions, they do not require major changes to a language's type system, and existing values may be able to be repurposed as capabilities. For example, in a function that writes to a file, the file object itself can be viewed as the capability for manipulating

it. One can also imagine similar capabilities, for example, for performing file operations and for throwing exceptions.

```
def write(f: File): Unit = {
    f.print("Hello World!\n")
}
```

Moreover, these extra capability arguments can be hidden away using implicit arguments and implicit function types, as shown by Odersky et al. [69]. This allows for a very compelling and lightweight effect system as shown by Craig et al. [21], *assuming capability safety*. However, it is hard to show capability safety, as capabilities, being values, can leak and escape in very interesting ways! Consider the following fragment of code:

```
var escapedFileHandle : FileReader = null;
def badReader(f : FileReader) : Unit = {
    escapedFileReader = f;
}
withFile("hello.txt")(badReader);
escapedFileHandle.read();
```

Here, `withFile` passes `badReader` a file capability `f` which it closes once `badReader` returns. However, `badReader` *leaks* the capability by setting the variable `escapedFileHandle` to it. After `withFile` returns, the code then attempts to use the leaked capability to unknown effect.

Leaking through a mutable variable is not the only problem that could happen; `badReader` could leak the capability by returning a closure which accidentally closed over it:

```
def badReader2(f : FileReader) : Unit => String = {
    () => { f.readLine(); }
}
```

Capture tracking [10] is a mechanism for annotating captured capabilities on the objects which capture them in order to prevent unwanted leakage. At its very core, it annotates values with the *sets of free variables* they can capture. So for example, we may give the following `read` function the type {database} `() => String` as it captures a global, tracked, database object (denoted by {*}).

```
val database: {*} Database = new Database("pets.db")
def read(): String = database.query("name")
```

By annotating types of values with the variables they capture, we can statically reason about the effects that they can perform, and restrict how they leak. For example, one can restrict tracked values from appearing in mutable variable cells; this would rule out `badReader` from above.

Capture sets, much like type qualifiers, induce a natural subtyping ordering as well. Functions that capture fewer variables can be used in place of functions that capture more. In addition, however, there is a *transitivity* property in the ordering that capture sets induce. Concretely, consider the following three objects:

```
val screen = Screen()
val keyboard = Keyboard()
val console = Console(screen, keyboard)
```

Here, we have three objects: a `screen`, a `keyboard`, and a `console`. A console object captures only an incoming `screen` and an incoming `keyboard`, so we should be able to reason that a function which uses, and captures, the `console` can be used in any context which allows usage of the `screen` object and the `keyboard` object as well, as a `console` here is no more than just a `screen` and `keyboard`. Concretely, `runSK` should accept `game` as an argument.

```
def runSK(f: {screen, keyboard} () => Unit) = ???
def game(): Unit = {
    console.write("You see a fork in the road.
    val direction = con.prompt("Take the left [L] path or right [R]?")
}
// game has type {console} () => Unit
runSK(game)
```

To do so, as capture sets contain arbitrary sets of variables each with upper capture bounds, Boruch-Gruszecki et al. [10] provide ad-hoc subcapturing judgments for resolving this order, as seen in Figure 2.1, unlike Foster et al. [31]'s calculus which relies on the lattice corresponding to the qualifier being used. Rule (SC-CAP) defines $\top$ or $*$ in the subcapturing relationship, rules (SC-DISTL) and (SC-DISTR) relate capture sets with their

Subcapturing

$$\boxed{\Gamma \vdash C <: C \quad \mathrm{cv}(X) = \{X\}, \mathrm{cv}(C\ T) = C}$$

$$\frac{* \in C_2}{\Gamma \vdash C_1 <: C_2}\ (\text{SC-CAP}) \qquad \frac{\Gamma \vdash x : T \qquad \Gamma \vdash \mathrm{cv}(T) <: C}{\Gamma \vdash \{x\} <: C}\ (\text{SC-VAR})$$

$$\frac{\Gamma \vdash \{q_1\} <: C, \ldots, \Gamma \vdash \{q_n\} <: C}{\Gamma \vdash \{q_1, \ldots, q_n\} <: C}\ (\text{SC-DISTL}) \qquad \frac{\Gamma \vdash X <: T \qquad \Gamma \vdash \mathrm{cv}(T) <: C}{\Gamma \vdash \{X\} <: C}\ (\text{SC-TVAR})$$

$$\Gamma \vdash \{q_i\} <: \{q_1, q_2, \ldots, q_n\}\ (\text{SC-DISTR})$$

Figure 2.1: Subsumption by Subcapturing in System $\mathtt{CC}_{<:\Box}$ (simplified from [9])

elements, and rules (SC-VAR) and (SC-TVAR) relate variables with their capture set upper bounds.

Here, we can deduce that `game` is subsumed by its upper capture set `{console}` via (SC-VAR), which it itself subsumed by its upper capture set `{screen, keyboard}` by another application of (SC-VAR).

2.3 Tracking Effects

Finally, we examine a system, CallE by Gariano et al. [35], for tracking effects by set-like annotations on methods. In their system, much like with Java's checked exceptions, methods must be annotated with the set of effects they perform.

```
class main:
    def hello() performs[Print] = print("Hello World")
```

So a function like `hello` is annotated with the set of effects it performs – here, `Print` as it prints to the screen.

Functions that perform fewer effects can be called in contexts which can handle more: here, we can call `pure` in a context that allows `Print`ing to happen. This induces a natural subtyping relationship based on inclusion on the declared effect sets.

```
class main:
    def hello() performs[Print] = print("Hello World")
```

```
    def pure() performs[] = 0
    def main() performs[Print]:
        hello()
        pure()
```

As experience with Java's checked exceptions has shown, forcing developers to explicitly annotate in total the set of effects a method can perform is both unwieldy and unreasonable, as effect sets grow too large to manipulate. A function like `bad`, which both prints, reads, and throws an exception may need to have an entry for each of those effects in its effect set.

```
class main:
    def bad() performs[Print, Read, IOException]:
        print("Hello World")
        val c = read()
        throw(IOException)
```

Moreover, a function that calls `bad` needs to reflect `bad`'s effect set on its own effect set as well – as it will perform the same set of effects as `bad` as well.

```
class main:
    def worse() performs[Print, Read, IOException]:
        bad()
```

Gariano et al. [35]'s key insight to ameliorate this situation was to realize that *methods* can be used as proxies for the effects they perform; so instead of writing the effect set `Print, Read, Exception` for `worse`, they would write the effect set `bad`:

```
class main:
    def worse() performs[bad]:
        bad()
```

To formalize this, they proposed the novel indirect effecting rule (SUBE-LOOKUP) which formalizes the transitivity of effects through methods, as shown in Figure 2.2. However, it is worth noting that these rules attempt to encode similar transitivity properties as capture tracking systems attempt to do with subcapturing. Much like how in capture tracking where variables are (transitively) subcaptured by the set of variables they capture, here, methods are (transitively) subeffected by the other methods they call.

Subeffecting for CallE

$\Gamma \vdash \varepsilon <: \varepsilon'$
ε is an effect set

$$\frac{}{\Gamma \vdash \varepsilon <: \varepsilon} \text{(SUBE-REFL)}$$

$$\frac{\Gamma \vdash \varepsilon <: \varepsilon' \qquad \Gamma \vdash \varepsilon' <: \varepsilon''}{\Gamma \vdash \varepsilon <: \varepsilon''} \text{(SUBE-TRANS)}$$

$$\frac{\Gamma \vdash \varepsilon_1 <: \varepsilon_1' \qquad \Gamma \vdash \varepsilon_2 <: \varepsilon_2'}{\Gamma \vdash \varepsilon_1 \cup \varepsilon_2 <: \varepsilon_1' \cup \varepsilon_2'} \text{(SUBE-COMP)}$$

$$\frac{\Gamma \vdash C.m \text{ is a method with effect set } \varepsilon}{\Gamma \vdash \{C.m\} <: \varepsilon} \text{(SUBE-LOOKUP)}$$

$$\frac{}{\Gamma \vdash \{C.m\} <: \{*\}} \text{(SUBE-TOP)}$$

$$\frac{}{\Gamma \vdash \{\} <: \{C.m\}} \text{(SUBE-BOT)}$$

Figure 2.2: Subeffecting in CallE

System	Qualifier	Type
Reference Immutability	potentially `@readonly`	
Capture Tracking System $\texttt{CC}_{<:\square}$	capture set {`a, b, c`}	type T
Effects – CallE	method set {`f, g, h`}	

Table 2.1: Stratified Types in Qualifier-Like Systems

2.4 A Common, Unifying, Theme?

While these systems present many different calculi solving three different problems, we will show they share some common underlying structure in how they augment a base type system for tracking immutability, capture, and effects/exceptions.

2.4.1 Stratified Typing – Type Qualifiers

Types in all three case studies – in all of the reference immutability systems, capture tracking, and CallE are stratified into an (optional) qualifier, and a core base type.

Qualifiers are used to qualify an underlying base type with additional information. Foster et al. [31] formalize a system for enriching types with qualifiers.

In all of the reference immutability schemes, `@readonly` can be viewed as a type qualifier, similar to how C++'s `const` can be viewed as a qualifier in [31]. How that qualifier

is encoded varies – in roDOT, it is encoded as a type member, and in other systems, as a type parameter or as an annotation.

In both CallE and in System $\texttt{CC}_{<:\square}$, the underlying qualifier does not neatly fit into Foster et al. [31]'s lattice system, as the additional information is encoded as a set of names tagged onto a type. In the case of System $\texttt{CC}_{<:\square}$, these names represent captured variable; similary in the case of CallE, these names represent other directly-called methods.

2.4.2 Subtyping

Additionally, all three systems support a natural subtyping relationship on their newly-stratified types based on the qualifier component present in said types. In reference immutability, types that represent references which are mutable, and have no qualifier present, are natural subtypes of types further qualified with `@readonly`, as mutable references can be used in positions where they are not written to; i.e in `@readonly` positions.

In System $\texttt{CC}_{<:\square}$, values which capture less can be used in positions where more is acceptable. Concretely, for example, a function which captures only a capability to use the {`Screen`}, for example, can be used in contexts which allow for both capabilities {`Screen, Keyboard`} to be used.

Similarly, in CallE, methods which perform fewer effects can be used in place of methods which can perform more. Concretely, a method which only reads from the console with effect set {`read`} can be used whenever one expects a method which both reads and writes to the console, with effect set {`read, write`}.

2.4.3 Mixing Qualifiers

All three systems need support for and do support combining qualifiers. In reference immutability, this is necessitated by *viewpoint adaptation* [24], as the type of a value read from a reference `x.f` depends on the mutability of `f` as well as its enclosing object `x`; it is the most restrictive of the two mutabilities.

In System $\texttt{CC}_{<:\square}$, data structures and closures that capture multiple values have a capture set that reflects the *union* of the capture sets of the individual values, through subcapturing. The closure that `iopair` below returns has a capture set {`x, y`}, for which we can reason that all of {`x`}, {`y`} are all contained underneath {`x, y`} in the subcapturing relationship, and both are below {`IO`}.

```
def iopair[X, Y](x: {IO} X, y: {IO} Y) =
    (which: Boolean) => if which then x else y
```

Finally, in CallE, methods naturally combine and bubble up effect sets.

```
class main:
    def a() performs[read] = read()
    def b() performs[write] = write("hello")
    def c() performs[a,b]:
        a()
        b()
```

2.4.4 Combining Qualifiers and Polymorphism

Polymorphism over qualifiers is done using different, ad-hoc methods in all three systems we discussed. With the various reference immutability systems we examined, we saw a variety of different rules for dealing with the interaction between polymorphism and type qualifiers. They vary from:

- Qualifier polymorphism via type members.
- Qualifier polymorphism without support for subtyping on qualifier variables.
- Restricted, ad-hoc polymorphism via type annotations.

With capture tracking, however, System $\mathtt{CC}_{<:\Box}$ uses ad-hoc system-dependent rules in its subcapturing system to support polymorphism over qualifier variables. This is similar to the situation with CallE as well, which supports polymorphism over effects by allowing type parameter variables to show up in effect sets, but there generics are left informally discussed.

This is not an ideal situation. Ad-hoc, informally specified rules do not lend themselves to reuse in other calculi.

Feature	Ref. Immut.	Capture Tracking	CallE (Effects)
Stratified Typing	`@readonly T`	`{a, b} T`	`{meth1, meth2} T`
Subtyping	`T <: @readonly T`	`{a} T <: {a, b} T`	`{meth1} <: {meth1, meth2}`
Mixing Qualifiers	`x.f` less mutable than either `x` or `f`	objects that capture both `a` and `b` capture anything that either `a` or `b` capture.	Methods with fewer effects can be used in places where more are handled or expected.
Polymorphism	restricted – only through type polymorphism	general, but with ad-hoc rules.	general (through type parameters), but informal.

Table 2.2: Common Type System Design Requirements/Features

2.4.5 Desirable Design Qualities

In short, we want a system that generalizes over these shared design needs that show up in all three case studies, as seen in Table 2.2. We want a *design recipe* for languages and types that supports lightweight stratified typing into both base types and qualifiers on those types, natural subtyping based on qualifiers, the ability to mix qualifiers, and general polymorphism on those qualifiers. Foster et al. [31]'s design recipe nicely supports *most* of these use cases save for polymorphism, and many other related works rely on ad-hoc solutions, like System $\mathtt{F}_{\mathtt{<:M}}$ and System $\mathtt{F}_{\mathtt{<:C}}$, to fill this gap, as we discuss in Chapter 6.

Thus, we show, in Chapter 3, how to generalize Foster et al. [31]'s calculi to work with polymorphic qualifier variables, while keeping their mathematical formalism for qualifiers intact.

Chapter 3

Polymorphic Lattice-based Type Qualifiers

In this chapter, we introduce System $\texttt{F}_{\texttt{<:Q}}$, a simple calculus that unifies support for qualified types as well as type- and qualifier polymorphism. We start off with a brief explanation of what type qualifiers are (Section 3.1), introduce System $\texttt{F}_{\texttt{<:Q}}$ (Section 3.3), and show that it satisfies the standard soundness theorems (Section 3.6).

3.1 A Simply-Qualified Type System

As Foster et al. [31] 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 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. Table 3.1 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* [31]; by example, `const String` is a supertype of a unqualified `String`. Conversely, qualifiers like `noexcept` are *negative*; `noexcept` (`String => String`) is a subtype of `String => String`.

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* [12]. A *positive* 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 *negative* qualifier, as `throws` is often omitted. [65]
`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. *Positive*, 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. *Positive*, in systems with this qualifier – most values ought not to be null.

Table 3.1: Examples of type qualifiers

3.2 Qualifying a Language

[40, 31] point out that the lattice order on qualifiers naturally induce a *subqualification* relation which in turn naturally induces a subtyping relationship on qualified types. This observation that qualifiers induce subtyping relationships allows language designers to seamlessly integrate support for type qualifiers into existing languages with subtyping. Going back to our earlier example, it is easy to update the contract of a function such as `toLowerCase` with a `const` qualifier on its argument because `const String` is a supertype of `String`. `toLowerCase` accepts a superset of the arguments it used to accept while gaining desirable safety properties – namely, that it does not modifiy `in`.

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

As Foster et al. [31] 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 Table 3.1.

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:

   (a) 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 $\{l_1\}\ T_1$ `<:` $\{l_2\}\ T_2$.

   (b) Add rules for *introducing qualifiers*, typically in the introduction forms for typing values.

   (c) 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.

3.3 (Higher-Rank) Qualifier Polymorphism

Foster's original work allows one to add qualifiers to an existing type system. 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 that is polymorphic in the mutability of the string passed to it. While this is easy enough, as Foster et al. [31] 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. [31], 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 using the underlying lattice ordering in order to preserve the natural subtyping relationship that qualifiers induce. But how?

3.4 Free Lattices

As Whitman [91] observed, there is a lattice which encodes the ordering structure of lattice formulas, namely, the *free lattice* constructed over the original qualifier lattice. Free lattices capture exactly the inequalities over lattice formulas that are *true in every lattice*.

Before we introduce the notion of a free lattice, we will review the order-theoretic definition of a lattice.

Definition 1. *A lattice $\mathcal{L}$ is a ground set L coupled with binary operations $\sqcap, \sqcup$ on L and a binary relation $\sqsubseteq$ satisfying:*

- $\sqsubseteq$ *is a partial order on* L*; namely that it is reflexive, antisymmetric, and transitive.*
- *For two elements* l *and* m *in* L*,* $l \sqcap m$ *gives the greatest lower bound of* l *and* m*.*
- *For two elements* l *and* m *in* L*,* $l \sqcup m$ *gives the least upper bound of* l *and* m*.*

With lattices introduced, we are ready to introduce the notion of a free 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. *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 3. *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* $\leq$ *defined by the lattice axioms above; namely,* $f_1 \leq f_2$ *if and only if it can be derived from the basic lattice axioms above in Definition 2.*

The free lattice is by definition a lattice, and embeds into every other, as every relation in the free lattice is deriviable from the lattice axioms and hence holds in every lattice.

Lemma 1. *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 3 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 [34] and Negri and von Plato [67] give algebraic and structural proofs of this result. Jipsen [45] notes this can also be viewed as a refomulation of Whitman [91, Theorem 1]. Skolem [83] however is probably responsible for the original formulation of Theorem 1, though with *transitivity* as an additional rule (8).

Theorem 1 (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 $f_3 \leq f_1 \wedge f_2$.*

Proof. To see that $\leq$ defines a lattice over $\mathcal{E}(X)/\leq$ we need to show that $\leq$ defines a partial order and that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{E}(X)$ with respect to $\leq$.

By definition $\leq$ is anti-symmetric over $\mathcal{E}(X)/\leq$ as we are quotienting out already by terms equivalent under $\leq$. Moreover, by induction one can show that $\leq$ is reflexive and transitive as well (as shown in the mechanized proof accompanying this thesis). Hence $\leq$ defines a partial order over $\mathcal{E}(X)/\leq$. Now, to see that $\leq$ defines a lattice, we need to show that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{E}(\mathcal{X})$ with respect to $\leq$. Now, to see that for two lattice terms f_1 and f_2 that $f_1 \vee f_2$ and $f_1 \wedge f_2$ are lower and upper bounds for both Q and R, one can apply the join introduction and the meet elimination rules. It remains to show that they are the greatest lower bounds and the least upper bounds – but that follows directly from an application of rules (2), (3), and (7), as desired.

Finally, we need to show that for two well-formed lattice equations $f_1[X]$ and $f_2[X]$ in $\mathcal{E}(\mathcal{X})$ such that $f_1[X] \leq f_2[X]$, it follows that for any bounded lattice L and any instantiation of the variables $\overline{X}$ to $\overline{L}$ respecting the bounds in Γ we have that $f_1[X \to \overline{L}] \sqsubseteq f_2[X \to \overline{L}]$. But all of the rules defining $\leq$ hold in every lattice. Hence $\mathcal{F}(X)$ is the free lattice. □

It should not be surprising to see free lattices here; as Dolan [27, 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. [31]'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 $\mathtt{F}_{<:}$, instead of implicitly at prenex position with constraints, as Foster et al. [31] do.

3.5 System $\mathtt{F}_{<:\mathtt{Q}}$

We are now ready to present our recipe by constructing System $\mathtt{F}_{<:\mathtt{Q}}$, a qualified extension of System $\mathtt{F}_{<:}$ 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 $\mathtt{F}_{<:\mathtt{Q}}$ which models a free lattice over a two-point qualifier lattice to illustrate our recipe. Our main change to System $\mathtt{F}_{<:}$ is to equip types with qualifiers, extend the existing subtyping relationship to account for qualifiers, and to provide a runtime interpretation of qualifiers (as in the style of [31]) to give progress and preservation meaning.

Syntax Figure 3.1 presents the syntax of System $\mathtt{F}_{<:\mathtt{Q}}$, with additions over System $\mathtt{F}_{<:}$ highlighted in grey. Type qualifiers Q include not only $\top$ and $\bot$ as in Foster et al. [31]'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 $s\{\!\{Q\}\!\}$.

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

Types and Qualifiers We extend unqualified types in System $\mathtt{F}_{<:}$ to account for qualifiers in the same line as [31] – fully qualified types are a pair $\{Q\}\ R$. Subtyping over fully qualified types accounts for both *subqualification* over the qualifier component as well as System $\mathtt{F}_{<:}$'s inherited subtyping over unqualified types. To allow for orthogonality and to allow for expressiveness, polymorphism is done over both *base types* and *qualifiers* with first-class type and qualifier variables, and type and qualifier abstraction terms.

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

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

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

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

C	::=		**Concrete Qualifiers**
	|	$\top$ or $\bot$	binary 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 $\bot \sqcup C = C$.

Figure 3.1: The syntax of System $\texttt{F}_{<:\texttt{Q}}$. Qualified differences to System $\texttt{F}_{<:}$ highlighted in *grey*.

Subqualification for System $\mathtt{F}_{<:\mathtt{Q}}$ $\boxed{\Gamma \vdash Q <: R}$

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

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

$$\frac{\Gamma \vdash Q <: R_1}{\Gamma \vdash Q <: R_1 \vee R_2}\ \text{(SQ-JOIN-INTRO-1)}$$

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

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

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

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

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

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

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

Figure 3.2: Subqualification rules of System $\mathtt{F}_{<:\mathtt{Q}}$.

Subqualification Next we show how simple subqualification extends from a lattice inequality in a base lattice (like how `noexcept` <: `throws`) to a lattice inequality in a free lattice. Figure 3.2 captures this free lattice structure of the qualifiers of System $\mathtt{F}_{<:\mathtt{Q}}$ with a *subqualification* judgment $\Gamma \vdash Q <: R$ to make precise the partial order between two lattice formulas in a free lattice. These rules mirror the rules in Theorem 1, but they are slightly modified to support upper bounds on variables instead of leaving variables completely free. This modification can be found on (SQ-VAR), which permits transitivity through a declared upper bound on a variable. Other rules reflect the basic free lattice constructions. Reflexivity (SQ-REFL) reflects rule (1) from Theorem 1. The rules for discharging and introducing joins (SQ-JOIN-INTRO-1), (SQ-JOIN-INTRO-2), and (SQ-JOIN-ELIM) reflect rules (2), (3), and (4) from Theorem 1. Symmetrically, the rules for discharging and introducing meets (SQ-MEET-ELIM-1), (SQ-MEET-ELIM-2), and (SQ-MEET-INTRO) reflect rules (5), (6), and (7) from Theorem 1. It should not be surprising that this construction gives rise to the free lattice, which we show with the following lemma.

Lemma 2. *Let Γ be an well-formed environment mapping qualifier variables to their upper bounds, and let $\mathcal{Q}$ be the set of all qualifier terms as defined in System $F_{<:\mathtt{Q}}$ that are well-formed in Γ; that is, bounded lattice formulas. Then the subqualification rules of System $F_{<:\mathtt{Q}}$ give rise to the free lattice modulo the bounds in Γ. Namely, that:*

1. *$<:$ defines a lattice over $\mathcal{Q}/<:$, namely, the set of terms of $\mathcal{Q}$ that are not equivalent under the order $<:$.*

2. *If $Q[\overline{X}] \in \mathcal{Q}$ and $R[\overline{X}] \in \mathcal{Q}$ are well-formed lattice equations over variables $\overline{X} \in \Gamma$ such that $\Gamma \vdash Q[\overline{X}] <: R[\overline{X}]$, then for any bounded lattice L and any instantiation of the variables $\overline{X}$ to $\overline{L_X}$ respecting the bounds in Γ we have that $Q[\overline{X} \to \overline{L_X}] \sqsubseteq R[\overline{X} \to \overline{L_X}]$.*

Proof. To see that $<:$ defines a lattice over $\mathcal{Q}/<:$ we need to show that $<:$ defines a partial order and that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{Q}$ with respect to $<:$.

By definition $<:$ is anti-symmetric over $\mathcal{Q}/<:$ as we are quotienting out already by terms equivalent under $<:$. Moreover, one can show by induction (as we show in our Rocq formalization) that $<:$ is reflexive and transitive as well (as shown in the artifact). Hence $<:$ defines a partial order over $\mathcal{Q}/<:$. Now, to see that $<:$ defines a lattice, we need to show that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{Q}$ with respect to $<:$. Now, to see that for two lattice terms Q and R that $Q \vee R$ and $Q \wedge R$ are lower and upper bounds for both Q and R one can apply the join introduction and the meet elimination rules. It remains to show that they are the greatest lower bounds and the least upper bounds – but that follows directly from an application of (SQ-JOIN-ELIM) and (SQ-MEET-INTRO), as desired.

Finally, we need to show that for two well-formed lattice equations $Q[\overline{X}] \in \mathcal{Q}$ and $R[\overline{X}] \in \mathcal{Q}$ over variables $\overline{X} \in \Gamma$ such that $\Gamma \vdash Q[\overline{X}] <: R[\overline{X}]$, then for any bounded lattice L and any instantiation of the variables $\overline{X}$ to $\overline{L_X}$ respecting the bounds in Γ we have that $Q[\overline{X} \to \overline{L_X}] \sqsubseteq R[\overline{X} \to \overline{L_X}]$. This we will do by structural induction over the subqualification derivation $\Gamma \vdash Q[\overline{X}] <: R[\overline{X}]$. Most of the rules hold directly, as they do not involve qualifier variables. The only two interesting cases are the rules (SQ-VAR) and (SQ-REFL-VAR). Now, for the (SQ-REFL-VAR) case we have that for any lattice term l that $l \sqsubseteq l$ so $X[X \to l] \sqsubseteq X[X \to l]$, as desired. Finally, for the (SQ-VAR) case we have that $Y <: Q \in \Gamma$ and $\Gamma \vdash Q <: R$. By induction and assumptions we have that X has been instantiated to a lattice term $l \in L$ such that $l \sqsubseteq Q[\overline{X} \to \overline{L_x}]$, and that $Q[\overline{X} \to \overline{L_X}] \sqsubseteq R[\overline{X} \to \overline{L_X}]$. So by transitivity we have that $l \sqsubseteq R[\overline{X} \to \overline{L_X}]$, hence $X[\overline{X} \to \overline{L_X}] \sqsubseteq R[\overline{X} \to \overline{L_X}]$, as desired. □

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.

$$X <: A \vee B \text{ by } (\text{SQ-JOIN-INTRO-1})$$
$$Y <: A \vee B \text{ by } (\text{SQ-JOIN-INTRO-2})$$
$$X \vee Y <: A \vee B \text{ by } (\text{SQ-JOIN-ELIM})$$

Subqualification, as one might expect, satisfies both transitivity and reflexivity. Note that reflexivity and transitivity are not given as explicit subqualification rules, similar to Negri and von Plato [67] and our presentation of the free lattice above. Here, we have transitivity and reflexivity integrated into the premises of the other subqualification rules.

Lemma 3. $\Gamma \vdash R$ `<:` Q *is transitive and reflexive.*

Proof. Can be found in the mechanization accompanying this thesis. □

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. [31, Section 2.2]. For example, the value $\lambda(x : \texttt{Int})_\top x$ would represent the integer function qualified at $\top$.

While System $\texttt{F}_{\texttt{<:Q}}$ 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. Conversely positive (or covariant) qualifiers like `const` would inform a default qualifier tag choice of $\bot$, as by default, in mutable languages, values should be mutable. Put simply, the default value tag should correspond to the default, omitted, qualifier.

Semantics To implement meaningful runtime semantics in System $\texttt{F}_{\texttt{<:Q}}$, we make some minor changes to the evaluation rules of System $\texttt{F}_{\texttt{<:}}$. These changes are defined in Figure 3.3. 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 and upcasting qualifier tags: they coerce qualifier expressions to concrete qualifiers when possible and ensure that the concrete qualifiers are compatible before successfully reducing. As (UPQUAL) and (ASSERT) need to compare qualifier expressions to determine if evaluation should continue, we introduce a helper function `eval` which evaluates qualifier expressions to concrete terms in the underlying lattice. This

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

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

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

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

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

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

$$\frac{v \text{ tagged with } P \qquad \mathtt{eval}(P) \sqsubseteq \mathtt{eval}(Q)}{\mathtt{assert}\, P\ v \longrightarrow v} \quad \text{(ASSERT)}$$

$$\frac{s \longrightarrow t}{E[s] \longrightarrow E[t]} \quad \text{(CONTEXT)}$$

E	::=	**Evaluation Context**
	|	$[]$
	|	$E(t) \mid v(E)$
	|	$E[S] \mid E[Q]$
	|	$\mathtt{upqual}\, P\ E$
	|	$\mathtt{assert}\, P\ E$

$\mathtt{eval}(P)$	::=		**Partial Qualifier Evaluation**
	| C	=>	$\mathtt{Some}\ C$
	| $P \wedge R$	=>	$\mathtt{Some}\ \mathtt{eval(P)} \sqcap \mathtt{eval(R)}$
	| $P \vee R$	=>	$\mathtt{Some}\ \mathtt{eval}(P) \sqcup \mathtt{eval}(R)$
	| _	=>	Nothing, otherwise.

Figure 3.3: Reduction rules for System $\mathtt{F}_{<:\mathtt{Q}}$

Subtyping for System $\mathtt{F}_{<:\mathtt{Q}}$– Base Types $\boxed{\Gamma \vdash S_1 <: S_1}$

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

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

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

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

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

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

Subtyping for System $\mathtt{F}_{<:\mathtt{Q}}$– Qualified Types $\boxed{\Gamma \vdash T_1 <: T_2}$

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

Figure 3.4: Subtyping rules of System $\mathtt{F}_{<:\mathtt{Q}}$. New rules highlighted in *grey*.

function is undefined, and hence is partial, for terms which contain qualifier variables, but is perfectly well defined for well-typed terms at evaluation, as evaluation proceeds by substitution in the empty environment.

With our major changes out of the way, we discuss how our changes fit into the rest of System $\mathtt{F}_{<:}$.

Subtyping System $\mathtt{F}_{<:\mathtt{Q}}$ inherits most of its rules for subtyping from System $\mathtt{F}_{<:}$, with two changes made (Figure 3.4). 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 another qualified type $\{Q_2\}\ S_2$ only if the qualifiers

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

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

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda(x)_P.t : \{\, P \,\}\ T_1 \to T_2} \text{ (ABS)} \qquad \frac{\Gamma \vdash t : \{Q\}\ \forall(X <: S).T \qquad \Gamma \vdash S' <: S}{\Gamma \vdash t[S'] : T[X \mapsto S']} \text{ (T-APP)}$$

$$\frac{\Gamma, X <: S \vdash t : T}{\Gamma \vdash \Lambda(X <: S)_P.t : \{\, P \,\}\ \forall(X <: S).T} \text{ (T-ABS)} \qquad \frac{\Gamma \vdash t : \{R\}\ \forall(Y <: Q).T \qquad \Gamma \vdash Q' <: Q}{\Gamma \vdash t\{\!\{Q'\}\!\} : T[Y \mapsto Q']} \text{ (Q-APP)}$$

$$\frac{\Gamma, X <: S \vdash t : T}{\Gamma \vdash \Lambda(Y <: Q)_P.t : \{\, P \,\}\ \forall(Y <: Q).T} \text{ (Q-ABS)} \qquad \frac{\Gamma \vdash s : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash s : T_2} \text{ (SUB)}$$

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

Figure 3.5: Typing rules for System $\mathtt{F}_{<:\mathtt{Q}}$

are in a subqualification relationship $Q_1 <: Q_2$, and the simple types are as well: $S_1 <: S_2$.

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 3.1) to use qualified types T instead of simple types S.

Typing Finally, Figure 3.5 defines the typing rules of System $\mathtt{F}_{<:\mathtt{Q}}$. The typing judgment assigns qualified types T to expressions, and can be viewed as $\Gamma \vdash t : \{Q\}\ S$. As System $\mathtt{F}_{<:\mathtt{Q}}$ does not assign an interpretation to qualifiers, the introduction rules for typing values from System $\mathtt{F}_{<:}$, (ABS) and (T-ABS), simply introduce qualifiers by typing values with their tagged qualifier, and their corresponding elimination/application rules remain unmodified.

The only changes from System $\mathtt{F}_{<:}$ are the new elimination rules that deal with qualifiers: the rules (TYP-ASSERT) and (TYP-UPQUAL), and the rules (Q-ABS) and (Q-APP) to support qualifier polymorphism.

3.6 Metatheory

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

Theorem 2 (Preservation)**.** *Suppose* $\Gamma \vdash s : T$*, and* $s \longrightarrow t$*. Then* $\Gamma \vdash t : T$ *as well.*

Theorem 3 (Progress)**.** *Suppose* $\varnothing \vdash s : T$*. Then either* s *is a value, or* $s \longrightarrow t$ *for some term* t*.*

Proof. These proofs can be found in the mechanization accompanying this thesis. □

While System $\texttt{F}_{\texttt{<:Q}}$ does not place any interpretation on qualifiers, it is already useful for enforcing safety constraints. By preservation we have that the static type of a value will always be greater than the tag annotated on it. As Foster et al. [31] point out, this safety property can be used to enforce invariants on values – in their example, on lists. For example, 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.

Moreover, `upqual` and `assert` can be used to check these constraints at runtime – and by progress, a well typed program will not fail these checks.

3.7 Generalizing Qualifiers to General Lattices

Qualifiers often come in more complicated lattices: for example, *protection rings* [47] 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 $\texttt{F}_{\texttt{<:Q}}$ using the free lattice based on two-point lattices to support general (countable, bounded) qualifier lattices $\mathcal{L}$ as well, using the free lattice based on $\mathcal{L}$.

Lemma 4. *Let* M *be an arbitrary base bounded lattice, let* Γ *be an well-formed environment mapping qualifier variables to their upper bounds, and let* $\mathcal{Q}$ *be the set of all qualifier terms*

as defined in extended System $\mathsf{F}_{<:\mathcal{Q}}$ *that are well-formed in* Γ*; that is, lattice formulas involving lattice values in* M*.*

Then the subqualification rules of extended System $\mathsf{F}_{<:\mathcal{Q}}$ *give rise to the free lattice of extensions of* M *modulo the bounds in* Γ*. Namely, that:*

1. $<:$ *defines a lattice over* $\mathcal{Q}/\ <:$ *, namely, the set of terms of* $\mathcal{Q}$ *that are not equivalent under the order* $<:$ *.*
2. *If* $Q[\overline{X}] \in \mathcal{Q}$ *and* $R[\overline{X}] \in \mathcal{Q}$ *are well-formed lattice equations over variables* $\overline{X} \in \Gamma$ *such that* $\Gamma \vdash Q[\overline{X}]\ <:\ R[\overline{X}]$*, then for any bounded lattice* L *extending* M *and any instantiation of the variables* $\overline{X}$ *to* $\overline{L_X}$ *respecting the bounds in* Γ *we have that* $Q[\overline{X} \to \overline{L_X}] \sqsubseteq R[\overline{X} \to \overline{L_X}]$*.*

Proof. To see that $<:$ defines a lattice over $\mathcal{Q}/\ <:$ we need to show that $<:$ defines a partial order and that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{Q}$ with respect to $<:$.

By definition $<:$ is anti-symmetric over $\mathcal{Q}/\ <:$ as we are quotienting out already by terms equivalent under $<:$. Moreover, one can show by induction (as we show in our Rocq formalization) that $<:$ is reflexive and transitive as well (as shown in the artifact). Hence $<:$ defines a partial order over $\mathcal{Q}/\ <:$. Now, to see that $<:$ defines a lattice, we need to show that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{Q}$ with respect to $<:$. Now, to see that for two lattice terms Q and R that $Q \vee R$ and $Q \wedge R$ are lower and upper bounds for both Q and R one can apply the join introduction and the meet elimination rules. It remains to show that they are the greatest lower bounds and the least upper bounds – but that follows directly from an application of (SQ-JOIN-ELIM) and (SQ-MEET-INTRO), as desired.

Finally, we need to show that for two well-formed lattice equations $Q[\overline{X}] \in \mathcal{Q}$ and $R[\overline{X}] \in \mathcal{Q}$ over variables $\overline{X} \in \Gamma$ such that $\Gamma \vdash Q[\overline{X}]\ <:\ R[\overline{X}]$, then for any bounded lattice L and any instantiation of the variables $\overline{X}$ to $\overline{L_X}$ respecting the bounds in Γ we have that $Q[\overline{X} \to \overline{L_X}] \sqsubseteq R[\overline{X} \to \overline{L_X}]$. This we will do by structural induction over the subqualification derivation $\Gamma \vdash Q[\overline{X}]\ <:\ R[\overline{X}]$. Most of the rules hold directly, as they do not involve qualifier variables nor partial evaluation; this includes the case for the rule (SQ-LIFT). The only four interesting cases are the rules (SQ-VAR), (SQ-REFL-VAR), (SQ-LIFT), and the two evaluation rules (SQ-EVAL-ELIM) and (SQ-EVAL-INTRO). The first two rules can be shown to hold via the same proof as in Lemma 2. So it remains to deal with the two cases for (SQ-EVAL-ELIM) and (SQ-EVAL-INTRO). As these cases are symmetric

$$
\begin{array}{rcll}
P, Q, R & ::= & & \textbf{Qualifiers in extended System } \mathtt{F}_{<:\mathtt{Q}} \\
& | & \colorbox{gray!10}{l} & \text{Base lattice elements } l \in L \\
& | & Y & \text{Qualifier variables} \\
& | & Q \wedge R \mid Q \vee R & \text{Meets and joins} \\
C & ::= & & \textbf{Concrete Qualifiers} \\
& | & \colorbox{gray!10}{l} & \text{Base lattice elements } l \in L
\end{array}
$$

Figure 3.6: The syntax of System $\mathtt{F}_{<:\mathtt{Q}}$ extended over a bounded lattice $\mathcal{L}$. Differences to System $\mathtt{F}_{<:\mathtt{Q}}$ highlighted in *grey*.

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

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

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

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

Figure 3.7: Extended sub-qualification rules for System $\mathtt{F}_{<:\mathtt{Q}}$.

we may assume without loss of generality that we are working with the rule (SQ-EVAL-ELIM). Hence we have Q, Q', and R, lattice formulas over lattice values in M such that $\Gamma \vdash Q <: Q'$, $\mathtt{eval}\, Q' = l$ for some lattice value $l \in M$, and $\Gamma \vdash l <: R$. By structural induction we have that $Q[\overline{X} \to \overline{L_X}] \sqsubseteq Q'[\overline{X} \to \overline{L_X}]$ and that $l \sqsubseteq R[\overline{X} \to \overline{L_X}]$. However, as $\mathtt{eval}\, Q' = l$, we have that Q' is simply the lattice meet and join of lattice values in L that evaluate to l; hence $Q' = l$ in M. Hence we have that $Q[\overline{X} \to \overline{L_X}] \sqsubseteq R[\overline{X} \to \overline{L_X}]$, as desired. □

Syntax The syntax changes needed to support this construction are listed in Figure 3.6. 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 3.7. 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 (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 $\Gamma \vdash l_1 \vee l_2 \texttt{ <: } l_1 \sqcup l_2$ and that $\Gamma \vdash l_1 \sqcup l_2 \texttt{ <: } l_1 \vee l_2$. However, without the two evaluation rules (SQ-EVAL-ELIM) and (SQ-EVAL-INTRO), we would only be able to conclude that $\Gamma \vdash l_1 \vee l_2 \texttt{ <: } l_1 \sqcup l_2$, but not the other desired inequality $\Gamma \vdash l_1 \sqcup l_2 \texttt{ <: } l_1 \vee l_2$.

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 appendix.

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

Theorem 4 (Preservation for Extended System $\texttt{F}_{\texttt{<:Q}}$)**.** *Suppose* $\Gamma \vdash s : T$*, and* $s \longrightarrow t$*. Then* $\Gamma \vdash t : T$ *as well.*

Theorem 5 (Progress for Extended System $\texttt{F}_{\texttt{<:Q}}$)**.** *Suppose* $\varnothing \vdash s : T$*. Then either* s *is a value, or* $s \longrightarrow t$ *for some term* t*.*

Proof. Can be found in the mechanization accompanying this thesis. □

3.8 Mechanization

The mechanization of System $\texttt{F}_{\texttt{<:Q}}$ (its derived calculi, System $\texttt{F}_{\texttt{<:QM}}$, System $\texttt{F}_{\texttt{<:QA}}$, and System $\texttt{F}_{\texttt{<:QC}}$, (all ahead in Chapter 4), and extended System $\texttt{F}_{\texttt{<:Q}}$), is derived from the mechanization of System $\texttt{F}_{\texttt{<:}}$ by Aydemir et al. [3]. While the proofs of soundness have been formally mechanized in Rocq, the theorems relating the subqualification structure of System $\texttt{F}_{\texttt{<:Q}}$ to free lattices are only proven in text.

The proof artifacts can be found at https://github.com/e45lee/phd-thesis-proofs.

3.9 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 found earlier, such as Lee et al. [52]. 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 easy. Many systems, like those of Zibin et al. [96] and Huang et al. [43], 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 [51], is to show that these equivalences can be discharged through subtyping rules which *normalize* equivalent types. However, that approach leads to complexities in proofs of soundness and it is unclear if that system admits practical 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. [90]. 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 => {Xq} 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 [72]. Alternatively, we could instead merge the qualifiers together:

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

In any case, however, these are all considerations for a surface language; the underlying calculus System $F_{<:Q}$ however can serve as a sound foundation for that surface language regardless of the design choices made.

Chapter 4

Applications of Polymorphic Type Qualifiers

In this chapter, we re-examine the ad-hoc qualifier systems we first visited in Chapter 2, and re-cast them in our general framework we introduced in Chapter 3. We also show how *coloured functions* [68] can be set in our general framework for polymorphic qualifiers as well.

4.1 Reference Immutability, Redux

We start by re-examining *reference immutability* from Chapter 2. Recall that 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 transitively reached values from it, so a value read through a compound object or reference qualified with `readonly` is also `readonly`.

```
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 [76, 96]. 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
```

To formalize reference immutability, we need to formalize exactly when references are used to update the values they refer to. For example, how do we check that the `access` function defined earlier does what it *claims* to do?

```
def access(z: @readonly Pair[Pair[Int]]): @readonly Pair[Int]
    = { z.first }
```

How do we check that `access` returns a reference to `z.first` that, at runtime, is never used to write to `z.first` or any other values transitively reachable from it through other references? How do we even express this guarantee precisely?

4.1.1 Dynamic Immutability Safety

If we consider a reference as a collection of getter and setter methods for the fields of the object it refers to, we could ensure that a reference is immutable by dropping all the setter methods. To ensure that immutability is transitive, we would also need to ensure that the result of applying a getter method is also immutable, i.e. by also dropping its setter methods and recursively applying the same modification to *its* getter methods. We will

make this precise by introducing the System $\lambda_{\texttt{M}}$ calculus with a notion of *sealed* references; references protected with wrapping forbidding writes through them.

System $\lambda_{\texttt{M}}$ is adapted from the CS-machine of Felleisen and Friedman [30] and extended with rules for dealing with sealed references.

Sealed references: To address the question about dynamic, runtime safety – can we ensure that read-only references are never used to mutate values – references can be explicitly *sealed* so that any operation that will mutate the cell referenced will fail to evaluate; see Figure 4.1.

The `seal` form protects its result from writes. A term under a `seal` form reduces until it becomes a value. At that point, values that are not records, like functions and type abstractions, are just transparently passed through the `seal` construct. However, values that are, such as the `record` form, remain protected by the `seal` form, and do not reduce further. For example:

$$\texttt{seal}\ (\{y : \texttt{0x0001}\})$$

is an irreducible value – a sealed record where the first field is stored at location 1 in the store. Intuitively, this can be viewed as removing the setter methods from an object reference. A sealed reference `seal` v behaves *exactly* like its unsealed variant v except that writes to `seal` v are forbidden and reads from `seal` v return `sealed` results.

Rules that mutate the cells corresponding to a record explicitly require an unsealed open record; see (WRITE-FIELD). This ensures that any ill-behaved program that mutates a store cell through a sealed record will get stuck, while an unsealed record can have its fields updated:

$$\begin{array}{rcl} \langle \{x : 10\}.x = 5, [] \rangle & \longrightarrow & \langle \{x : \texttt{0x0001}\}.x = 5, [\texttt{0x0001} : 10] \rangle \\ & \longrightarrow & \langle 10, [\texttt{0x0001} : 5] \rangle \end{array}$$

A sealed record cannot have its fields written to. Unlike record field reads, for which there is a sealed (SEALED-FIELD) counterpart to the standard record read rule (FIELD), there is no corresponding rule for writing to a sealed record for (WRITE-FIELD). Recall that (WRITE-FIELD) requires an *open, unsealed* record as input:

$$\frac{l : v \in \sigma}{\langle \{\ldots x : l \ldots\}.x = v', \sigma \rangle \longrightarrow \langle v, \sigma[l \mapsto v'] \rangle}$$

The calculus does not contain any rule like the following, which would reduce writes on a sealed record:

$$
\begin{array}{rcll}
s, t & ::= & & \textbf{Terms} \\
& | & \lambda x.t & \text{abstraction} \\
& | & x & \text{variable} \\
& | & s(t) & \text{application} \\
& | & \{f_1 : s_1, f_2 : s_2, \ldots\} & \text{records} \\
& | & s.f & \text{field read} \\
& | & s.f = t & \text{field write} \\
& | & \texttt{seal}\ s & \text{sealing}
\end{array}
$$

$$
\begin{array}{rcll}
 & & l & \textbf{Location} \\
s, t & ::= & & \textbf{Runtime Terms} \\
& | & \{f_i : l_i, \ldots\} & \text{runtime record} \\
v & ::= & & \textbf{Runtime Values} \\
& | & \lambda x.t & \\
& | & \{f_i : l_i\} & \\
& | & \texttt{seal}\ \{f_i : l_i, \ldots\} &
\end{array}
$$

$$\langle (\lambda x.t)(v), \sigma \rangle \longrightarrow \langle t[x \mapsto v], \sigma \rangle \ (\textsc{beta-v})$$

$$\frac{l : v \in \sigma}{\langle \left(\texttt{seal}\ \{\ldots x : l \ldots\}\right).x, \sigma \rangle \longrightarrow \langle\ \texttt{seal}\ v\ , \sigma \rangle} \ (\textsc{sealed-field})$$

$$\frac{l_i \notin \sigma}{\langle \{x_i : v_i\}, \sigma \rangle \longrightarrow \langle \{x_i : l_i\}, (\sigma, l_1 : v_1, l_2 : v_2, \ldots) \rangle} \ (\textsc{record-store})$$

$$\langle\ \texttt{seal}\ (\lambda x.t)\ , \sigma \rangle \longrightarrow \langle \lambda x.t, \sigma \rangle \ (\textsc{seal-elim-abs})$$

$$\frac{l : v \in \sigma}{\langle \{\ldots x : l \ldots\}.x, \sigma \rangle \longrightarrow \langle v, \sigma \rangle} \ (\textsc{field})$$

$$\langle\ \texttt{seal seal}\ v\ , \sigma \rangle \longrightarrow \langle \texttt{seal}\ v, \sigma \rangle \ (\textsc{seal-elim-multiple})$$

$$\frac{l : v \in \sigma}{\langle \{\ldots x : l \ldots\}.x = v', \sigma \rangle \longrightarrow \langle v, \sigma[l \mapsto v'] \rangle} \ (\textsc{write-field})$$

$$\frac{\langle s, \sigma \rangle \longrightarrow \langle t, \sigma' \rangle}{\langle E[s], \sigma \rangle \longrightarrow \langle E[t], \sigma' \rangle} \ (\textsc{context})$$

$$
\begin{array}{rcll}
E & ::= & [] \mid E(t) \mid v(E) & \textbf{Evaluation Context} \\
& | & \{x_0 : v_0, \ldots, x_i : E, x_{i+1} : t_{i+1}, \ldots\} & \\
& | & E.x & \\
& | & E.x = t \mid v.x = E & \\
& | & \texttt{seal}\ E &
\end{array}
$$

Figure 4.1: The syntax and semantics of λ_m.

$$\frac{l : v \in \sigma}{\langle\, (\texttt{seal}\ \{\ldots x : l \ldots\})\, .x = v', \sigma\rangle \longrightarrow \langle v, \sigma[l \mapsto v']\rangle}$$

So a term like:

$$\begin{array}{rcl} \langle(\texttt{seal}\ \{x : 10\}).x = 5, []\rangle & \longrightarrow & \langle \texttt{seal}\ (\{x : \texttt{0x0001}\}).x = 5, [\texttt{0x0001} : 10]\rangle \\ & \longrightarrow & \text{gets stuck.} \end{array}$$

Dynamic viewpoint adaptation: After reading a field from a sealed record, the semantics *seals* that value, ensuring transitive safety – see (SEALED-FIELD).

$$\frac{l : v \in \sigma}{\langle\, (\texttt{seal}\ \{\ldots x : l \ldots\})\, .x, \sigma\rangle \longrightarrow \langle\, \texttt{seal}\ v\, , \sigma\rangle}$$

For example:

$$\begin{array}{rcl} \langle(\texttt{seal}\ \{y : \{x : 10\}\}).y, []\rangle & \longrightarrow & \langle \texttt{seal}\ (\{y : \{x : \texttt{0x001}\}\}).y, [\texttt{0x001} : 10]\rangle \\ & \longrightarrow & \langle \texttt{seal}\ (\{y : \texttt{0x002}\}).y, [\texttt{0x001} : 10, \texttt{0x002} : \{x : \texttt{0x001}\}]\rangle \\ & \longrightarrow & \langle \texttt{seal}\ (\{x : \texttt{0x001}\}), [\texttt{0x001} : 10, \texttt{0x002} : \{x : \texttt{0x001}\}]\rangle \end{array}$$

Sealed references and *dynamic viewpoint adaptation* allow for a succinct guarantee of *dynamic transitive immutability safety*. Namely, no value is ever mutated through a read-only reference or any other references transitively derived from it.

Aside from preventing writes through sealed references, we should show that sealing does not otherwise affect reduction. For this we need a definition that relates pairs of terms that are essentially equivalent except that one has more seals than the other.

Definition 4. *Let s and t be two terms. We say $s \leq t$ if t can be obtained from s by repeatedly replacing sub-terms s' of s with sealed subterms* `seal` s'.

This implies a similar definition for stores:

Definition 5. *Let σ and σ' be two stores. We say $\sigma \leq \sigma'$ if and only if they have the same locations and for every location $l \in \sigma$, we have $\sigma(l) \leq \sigma'(l)$.*

The following three lemmas formalize how reduction behaves for terms that are equivalent modulo seals. The first one is for a term t that is equivalent to a value – it states that if t reduces, the resulting term is still equivalent to the same value. It also shows that the resulting term has fewer seals than t, which we'll need later for an inductive argument.

Definition 6. *Let s be a term. Then $|s|$ is the number of seals in s.*

Lemma 5. *Let v be a value, σ_v be a store, t be a term such that $v \leq t$, and σ_t be a store such that $\sigma_v \leq \sigma_t$.*

If $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$ then $v \leq t'$, $\sigma_v \leq \sigma'_t$, and $|t'| < |t|$.

The next lemma is an analogue of Lemma 5 for terms. Given two equivalent terms s and t, if s steps to s' and t steps to t', then either s and t' are equivalent or s' and t' are equivalent. Moreover, again, to show that reduction in t is equivalent to reduction in s, we have that $|t'| < t$ if $s \leq t'$.

Lemma 6. *Let s, t be terms such that $s \leq t$ and let σ_s, σ_t be stores such that $\sigma_s \leq \sigma_t$. If $\langle s, \sigma_s \rangle \longrightarrow \langle s', \sigma'_s \rangle$ and $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$ then:*

1. *Either $s \leq t'$, $\sigma_s \leq \sigma'_t$, and $|t'| < |t|$, or*
2. *$s' \leq t'$ and $\sigma'_s \leq \sigma'_t$.*

Together, Lemmas 5 and 6 relate how terms s and t reduce when they are equivalent modulo seals. Assuming that both s and t reduce, every step of s corresponds to finitely many steps of t, and they reduce to equivalent results as well. This shows that sealing is transparent when added onto references that are never written to, allowing for a succinct guarantee of immutability safety.

Finally, the last lemma states that erasing `seal`s will never cause a term to get stuck. Seals can be safely erased without affecting reduction.

Lemma 7. *Let s, t be terms such that $s \leq t$ and let σ_s, σ_t be stores such that $\sigma_s \leq \sigma_t$. If $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$ then:*

1. *Either $s \leq t'$, $\sigma_s \leq \sigma'_t$, and $|t'| < |t|$, or*
2. *There exists s' and σ'_s such that $\langle s, \sigma_s \rangle \longrightarrow \langle s', \sigma'_s \rangle$, $s' \leq t'$ and $\sigma'_s \leq \sigma'_t$.*

From this we can derive the following multi-step analogue, after observing the following lemma:

Lemma 8. *If s is a term and v is a value such that $s \leq v$, then s is also a value.*

Hence:

Lemma 9. *Suppose s and t are terms such that $s \leq t$. If $\langle t, \sigma_t \rangle \longrightarrow^* \langle v_t, \sigma'_t \rangle$ for some value v_t, then for any $\sigma_s \leq \sigma_t$ we have $\langle s, \sigma_s \rangle \longrightarrow^* \langle v_s, \sigma'_s \rangle$ such that $v'_s \leq v'_s$ and $\sigma'_s \leq \sigma'_t$.*

Finally, it can be shown that the seals are to blame when two equivalent terms s and t reduce differently – in particular, when one reduces but the other gets stuck.

Lemma 10. *Let s, t be terms such that $s \leq t$, and let σ_s, σ_t be stores such that $\sigma_s \leq \sigma_t$. If $\langle s, \sigma_s \rangle \longrightarrow \langle s', \sigma'_s \rangle$ and t gets stuck, then the reduction performed on s was a write to a record using rule (WRITE-FIELD).*

Proof. (Sketch) As s cannot further reduce, the evaluation context of s and t must match; there are no extraneous `seal`s that need to be discharged. As such, from inspection of the reduction rules, we see that in all cases except for (WRITE-FIELD), for every possible reduction that s could have taken, there is a possible reduction that t could have taken as well, as desired. □

4.1.2 Qualifiers for Reference Immutability

Seals have a natural analogue in a qualifier system – the presence of a seal corresponds to a value tagged at $\top$ or `readonly`, and the absence of a seal corresponds to a value tagged at $\bot$. Here, with this intuition in mind, 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 $\mathtt{F}_{<:\mathtt{QM}}$. Concretely, we will prove analogues of Lemmas 5, 6 and 9 replacing seals with tags on values (qualifiers).

Assigning Qualifiers We need to define how qualifiers `mutable` and `readonly` are assigned to $\top$ and $\bot$ in System $\mathtt{F}_{<:\mathtt{QM}}$. 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 4.2.

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 $\langle t, \sigma \rangle$:

			Terms				**Types**
s, t	$::=$				$::=$		
	$\ldots$				$\ldots$		
	$\mid$	$\texttt{box}_P\, t$	reference cell		$\mid$	$\texttt{box}\, S$	reference type
	$\mid$	$\texttt{unbox}\ s$	deferencing	P, Q, R	$::=$		**Qualifiers**
	$\mid$	$\texttt{set-box!}\ s\ t$	reference update		$\ldots$		as before, except:
					$\mid$	$\texttt{readonly}$	as $\top$
					$\mid$	$\texttt{mutable}$	as $\bot$

		l	**Location**	σ	$::=$		**Store**
s, t	$::=$		**Runtime Terms**		$\mid$	$\cdot$	empty
	$\mid$	$\texttt{box}_P\, l$	runtime reference		$\mid$	$\sigma,\ l : v$	cell l with value v
v	$::=$		**Runtime Values**	Σ	$::=$		**Store Environment**
	$\ldots$				$\mid$	$\cdot$	empty
	$\mid$	$\texttt{box}_P\, l$			$\mid$	$\sigma,\ l : T$	cell binding

Figure 4.2: The syntax of System $\texttt{F}_{\texttt{<:QM}}$.

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

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.

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

One can explicitly mark a value immutable by `upqual`-ing to $\top$. The elimination form for reading from a reference, (DEREF), ensures that a value read from a reference tagged `readonly`, or at $\top$, remains `readonly`. This is reflected in the updated operational semantics (Figure 4.3).

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

$$\frac{l \notin \sigma}{\langle \texttt{box}_P\, v, \sigma\rangle \longrightarrow \langle \texttt{box}_P\, l, (\sigma, l : v)\rangle}\ (\textsc{ref-store})$$

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

$$\frac{l : v \in \sigma \qquad v \text{ tagged with } Q}{\langle \texttt{unbox box}_P\, l, \sigma\rangle \longrightarrow \langle v \text{ retagged at } P \vee Q, \sigma\rangle}\ (\textsc{deref})$$

$$\frac{\langle s, \sigma\rangle \longrightarrow \langle t, \sigma'\rangle}{\langle E[s], \sigma\rangle \longrightarrow \langle E[t], \sigma'\rangle}\ (\textsc{context})$$

$$\begin{array}{rcll} E & ::= & \ldots & \textbf{Evaluation Context} \\ & | & \texttt{box}_P\, E & \\ & | & \texttt{unbox}\ E & \\ & | & \texttt{set-box!}\ E\ \ t \mid \texttt{set-box!}\ v\ \ E & \end{array}$$

Figure 4.3: Reduction rules for System $\texttt{F}_{\texttt{<:QM}}$

Additional Typing and Runtime Typing for System $\texttt{F}_{\texttt{<:QM}}$ $\boxed{\Gamma \mid \Sigma \vdash t : T \text{ and } \Gamma \mid \Sigma \vdash \sigma}$

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

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

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

Figure 4.4: Typing rules for System $\texttt{F}_{\texttt{<:QM}}$; notable changes highlighted in *grey*.

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 4.4. To ensure immutability safety, the standard reference update elimination form (REF-UPDATE) is augmented to check that a reference can be written to if and only if it can be typed as some `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 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 6 (Preservation of System $\mathtt{F}_{<:\mathtt{QM}}$)**.** *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 7 (Progress for System $\mathtt{F}_{<:\mathtt{QM}}$)**.** *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 $\mathtt{F}_{<:\mathtt{QM}}$: 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.

This we can formalize as well using the intuition we developed earlier with seals and sealing.

Definition 7. *A term* s *is the stripped version of a term* t*, denoted* $s \triangleleft t$*, if* s *is equal to* t *modulo qualifiers in tag position and in qualifier upcasts; in those positions, qualifiers are set at* $\bot$*.*

This implies a similar definition for stores:

Definition 8. *Let σ and σ' be two stores. We say $\sigma \triangleleft \sigma'$ if and only if they have the same locations and for every location $l \in \sigma$, we have $\sigma(l) \triangleleft \sigma'(l)$.*

Stripping tags will never disturb evaluation. Given two terms s and t where $s \triangleleft t$, then if s and t step to s' and t', we have that $s' \triangleleft t'$.

Lemma 11. *Let s, t be terms such that $s \triangleleft t$ and let σ_s, σ_t be stores such that $\sigma_s \triangleleft \sigma_t$. If $\langle s, \sigma_s \rangle \longrightarrow \langle s', \sigma'_s \rangle$ and $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$ then*

$s' \triangleleft t'$ and $\sigma'_s \triangleleft \sigma'_t$.

Moreover, stripping tags will never cause a term to get stuck.

Lemma 12. *Let s, t be terms such that $s \triangleleft t$ and let σ_s, σ_t be stores such that $\sigma_s \triangleleft \sigma_t$. If $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$ then there exists s' and σ'_s such that $\langle s, \sigma_s \rangle \longrightarrow \langle s', \sigma'_s \rangle$, $s' \leq t'$ and $\sigma'_s \leq \sigma'_t$.*

Multistep immutability follows naturally in this setting as s and t in both lemmas step in lockstep.

4.2 Capture Tracking, Redux

Our design recipe can also 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 [22], 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 [1]:

```
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. Table 4.1 summarizes the different variables in the above examples and the qualifiers we would like to assign to their types.

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`.

Table 4.1: Qualifier assignments in Capture Tracking

There is an ongoing effort to incorporate such a system into Scala, and a working prototype has been implemented in the Dotty compiler – see Boruch-Gruszecki et al. [10]. Formalizing such a system has been difficult. The problem is that a free variable tracking system is inherently dependently typed, in that types track (term-level) variables. Consider an example Scala definition:

```
def delayedPrint(s: {*} String) = () => print(s)
```

Here, the `*` in the type of `s` specifies that capturing `s` is to be tracked by the type system. Thus, the return type of `delayedPrint` could be `{s} Unit => Unit` to indicate that the closure that is returned captures `s`. A type like `print (s : {*} String) => {s} Unit => Unit` will always unfortunately contain term variables.

An early attempt to formalize a tracking system by Boruch-Gruszecki et al. [8] runs into complications concerning dependent types, which they solve by enforcing variance restrictions on where type variables can occur. More recently, Boruch-Gruszecki et al. [10] sidestep these issues by working in administrative normal form (ANF), a technique previously used by Rapoport and Lhoták [77] and Amin et al. [2] to work around similar complications in formalizing type dependency in DOT, Scala's core calculus. However, the indirection of ANF makes it difficult to acquire an intuition for the properties of the system.

This does not need to be the case, though. As Brachthäuser et al. [14] show, a capture tracking system can be formalized with a simple core calculus even in the presence of dependent types. While their system lacks subtyping, their system differs in one key way compared to Boruch-Gruszecki et al. [8] and Boruch-Gruszecki et al. [10] which enables a simple soundness proof to go through while still keeping track of captured variables

in types. They annotate their function applications with the capture information which should be substituted instead of relying on the computed capture information on the actual value being substituted in.

Inspired by Brachthäuser et al. [14], we show that a similar technique also works in systems like Boruch-Gruszecki et al. [8] and Boruch-Gruszecki et al. [10]. We present a simple calculus System $\mathtt{F}_{<:\mathtt{C}}$ that extends System $\mathtt{F}_{<:}$ with capture tracking without resorting to ANF nor to variance restrictions on type variables, for which we have mechanized a soundness proof, and we sketch out how System $\mathtt{F}_{<:\mathtt{C}}$ can be used as a basis of soundness proofs for a surface syntax in a practical programming language. We then show that System $\mathtt{F}_{<:\mathtt{C}}$ can be recast using System $\mathtt{F}_{<:\mathtt{Q}}$ as a foundation to demonstrate the expressiveness of System $\mathtt{F}_{<:\mathtt{Q}}$.

4.2.1 Capture Tracking, A Core Presentation

At its core, capture tracking is concerned with:

1. Tracking what free variables are present in values, and,
2. For functions, tracking how arguments flow into results.

For example, in our earlier example:

```
def delayedPrint(s: {*} String) = () => print(s)
```

The above term would receive the following type:

```
{print} (s : {*} String) => {s} Unit => Unit
```

This type describes that `delayedPrint` is a function that returns a thunk which captures the parameter `s` passed to `delayedPrint`.

Applying delayedPrint to a string returns a value whose type reflects that it captures that string. For example, in the following listing, `delayedPrint(hello)` has type `{hello} Unit => Unit`.

```
val hello: {*} String = "Hello World".
delayedPrint(hello) // returns thunk with type {hello} Unit => Unit
```

Dependent Types

Dependent types pose a problem. Consider the following example.

```
val goodbye: {*} String = "Goodbye World".
def wantsEqual = (s: {*} () => String) =>
                 (t: {s} () => String) => { ... }
wantsEqual
   (if true then () => hello else () => goodbye)
   (() => goodbye)
```

Here, we declare that `goodbye` is a `String` which should be tracked, and `wantsEqual` is a curried function that expects a value `s` that can capture anything and a value `t` that captures the same variables as `s` does. We then apply `wantsEqual` to a conditional expression that returns two thunks: one returning `hello` and one returning `goodbye`:

```
if true then () => hello else () => goodbye
```

Now, what should the type of this conditional term be? Both branches return a thunk returning a `String`, but one captures `hello` and the other `goodbye`. All we can say is that the conditional could possibly capture `hello` or `goodbye`, giving the conditional the type `{hello, goodbye} () => String`. Thus, `wantsEqual(if true then () => hello else () => goodbye)` has type `({hello, goodbye}() => String) => T` for some `T`. Accordingly, we widen the type of the second argument of `wantsEqual` from `{goodbye}() => String` to `{hello, goodbye}() => String`. The overall program is well typed.

Now, consider what happens after a single step of reduction, which reduces the conditional term to just the thunk `() => hello`. The overall program reduces to:

```
wantsEqual(() => hello)(() => goodbye)
    // what type does this function have?
```

The next step will be to reduce the application `wantsEqual(() => hello)`. As is standard for function application, we substitute the argument `() => hello` for the parameter `s` in the body of `wantsEqual`. But `s` occurs not only in the body of the function, but also in the type of the second parameter `t`, which is `{s} () => String`. What should be substituted for `s` in the type of `t`?

There are a few options. One option is to substitute with the actual set of variables that are captured by the value, namely the free variables of `() => hello`. This is given by the substitution rule:

$$(\lambda(x).Tt)v \longrightarrow t\,[x \mapsto_{\texttt{type}} fv(v)]\,[x \mapsto_{\texttt{term}} v] \qquad \text{(BETA-V-BAD)}$$

However, this cannot be done so easily. In our example, if we replaced `s` with `hello` in the types of the body of `wantsEqual`, we end up with the term:

```
((t: {hello} () => String) => ...)(() => goodbye)
```

This is obviously ill-typed, since the argument captures `goodbye` but is required by the parameter type to capture `hello`. The reduction in this example violates type preservation. In this example two things went wrong for this to transpire:

1. The capture set that we had "assigned" to `s` shrank under reduction from {`hello, goodbye`} to just {`hello`}, and,
2. We used `s` in a negative, contravariant position – namely, in the type annotation for what we expect to be passed in the second parameter `t`.

This breaks preservation under reduction, as contravariance flips the subtyping hierarchy. It would be unsound for `s` to shrink in this manner. It must either stay static or expand. In order for preservation to hold, we need to prevent this from happening.

One possibility is to simply disallow contravariant occurrences of term variables in types. However, this is restrictive, and complicates the proof of soundness, which needs to enforce and check that term variables occur only covariantly in types, even under reduction.

Another possibility is to make what we "assign" to `s` static, by putting programs in administrative normal form, in effect assigning a name to every (sub)expression in a program:

```
val tmp =
    if true then () => hello else () => goodbye
wantsEqual(tmp) // expects {tmp} String
```

Now, since we have named the conditional term `tmp`, we can give `wantsEqual(tmp)` the type `({tmp} () => String) => T`. Since `tmp` is named, we can use the capture set {`tmp`} to represent exactly what `tmp` captures, even as what is stored in `tmp` shrinks under reduction from something capturing {`hello, goodbye`} to just `hello`.

```
val tmp = () => hello // reduced
wantsEqual(tmp) // still expects {tmp} String.
```

This is the approach taken by Boruch-Gruszecki et al. [10]. Similar approaches have been used to handle variance issues in other contexts – for example, Rapoport and Lhoták [77] and Amin et al. [2]. However, administrative normal form adds complexity to the reduction rules. Moreover, it does not even improve expressiveness for this example. While `wantsEqual` can be typed, the only variable that can be passed to `wantsEqual(tmp)` is either `tmp` itself or a variable that does not capture any other tracked variables. In particular, the ANF-converted version of the original term `wantsEqual(tmp)(() => goodbye)` remains ill-typed.

In Section 4.2.2, we explore a third approach, which is to annotate each term application in the original program with the *capture set* that will be substituted in place of that term variable in capture set position. That set is fixed in the original program and does not shrink with reduction. Formally, we replace (BETA-V-BAD) with the following:

$$(\lambda(x).Tt)\,[C]\,(v) \longrightarrow t\,[x \mapsto_{\texttt{type}} C]\,[x \mapsto_{\texttt{term}} v] \qquad \text{(BETA-V)}$$

This neatly sidesteps the issue of shrinking variables and variance, at the cost of annotating each term application with a capture set. The resulting system satisfies type preservation and allows us to type the `wantsEqual` example with the capture set annotation `{hello,goodbye}`:

```
wantsEqual
  {hello, goodbye}
    (if true then () => hello else () => goodbye)
  {hello, goodbye}(() => goodbye)
-->
wantsEqual
  {hello, goodbye}(() => hello)
  {hello, goodbye}(() => goodbye)
-->
(t: {hello, goodbye} () => String)
  {hello, goodbye}(() => goodbye)
```

The System C calculus of Brachthäuser et al. [14] also uses this approach, but in the context of a rather different calculus.

Types, Shapes, and Polymorphism

Another issue that arises is that of parametric polymorphism. What should a type variable `X` quantify over, now that types contain both information describing the *shape* of a value, as well as what that value *captures*. The seemingly obvious answer is that `X` should stand for both the shape and the capture information of a value, but this imposes some restrictions, as we discussed earlier in Section 3.9.

To sidestep this issue, Boruch-Gruszecki et al. [10] require generic data structures to hold only pure elements with empty capture sets. This would be impractically restrictive, however, so they add a special mechanism called *boxing* to "tunnel" captures, by temporarily making impure values pure so they can be stored in a generic data structure, but marking their type to ensure that the necessary captures are covered by the type when the element is later retrieved from the data structure.

Here, inspiring the choices we made in Section 3.9, we use a more familiar mechanism, for which soundness proofs are well understood: an additional binding form for variables that stand for capture sets. Thus, we separate the binders for the shape part and the capture part of a type. In this setting, we could give `map` the following signature:

```
def map[C,X,Y](l: {*} List[{C} X], f: {*} X => Y)
  : {C, f} List[{C, f} Y]
```

Here, the capture set binder allows us to introduce a name `C` for the capture set of the elements of `l`.

4.2.2 A Simpler Calculus

To this end, we develop a simpler capture tracking calculus System $\mathtt{F_{<:C}}$ inspired by these observations. As discussed earlier, we explicitly annotate term applications with the capture sets which should be substituted in. Specifically, we define application reduction to be as below:

$$(\lambda(x).Tt)\,[C]\,(v) \;\longrightarrow\; t\,[x \mapsto_{\mathtt{type}} C]\,[x \mapsto_{\mathtt{term}} v] \qquad \text{(BETA-V)}$$

Obviously, though, `C` needs to bound to what could be carried in `v` – this is handled through the typing rule (APP).

$$\frac{\Gamma \vdash t : \{D\}\ x \to \{C\}\ ST \qquad \Gamma \vdash s : \{C\}\ S}{\Gamma \vdash t(C)s : T[x \mapsto C]} \qquad \text{(APP)}$$

s, t	::=		**Terms**	S	::=		**Shapes**
	$\mid$	$\lambda(x).Tt$	term abstraction		$\mid$	X	shape variable
	$\mid$	$\Lambda(X <: S).t$	shape abstraction		$\mid$	$x \to T_1 T_2$	function shapes
	$\mid$	$\Lambda(Y <: C).t$	capture abstraction		$\mid$	$\forall(X <: S).T$	for-all shapes
	$\mid$	x	term variable		$\mid$	$\forall(Y <: C).T$	capture for-all shapes
	$\mid$	$s(C)t$	application	T	::=		**Types**
	$\mid$	$s[S]$	shape application		$\mid$	$\{C\}\ S$	capturing type
	$\mid$	$s[C]$	capture application	C, D	::=		**Capture Sets**
Γ	::=		**Environment**		$\mid$	$\{x_1, \ldots, Y_1, \ldots\}$	proper set
	$\mid$	$\cdot$	empty		$\mid$	$\top$	universal set
	$\mid$	$\Gamma,\ x : T$	term binding		where	Y	capture variable
	$\mid$	$\Gamma,\ X <: S$	shape binding				
	$\mid$	$\Gamma,\ Y <: C$	capture binding				

Figure 4.5: Syntax of System $\mathtt{F}_{<:\mathtt{C}}$.

To handle parametric polymorphism, as discussed in Section 2.2, we adopt a split shape-type system with polymorphism on shape variables. We support capture polymorphism directly using a third binder $\Lambda(Y <: C).t$ for capture set variables.

This allows for straightforward mechanized proofs of Progress and Preservation, without requiring variance restrictions or administrative normal form.

Theorem 8 (Preservation)**.** *Suppose* $\Gamma \vdash s : T$*, and* $s \longrightarrow t$*. Then* $\Gamma \vdash t : T$ *as well.*

Theorem 9 (Progress)**.** *Suppose* $\varnothing \vdash s : T$*. Either* s *is a value, or* $s \longrightarrow t$ *for some term* t*.*

Our development has been mechanized in Rocq, based on the mechanization System $\mathtt{F}_{<:}$ of Aydemir et al. [3]. While there are are many lemmas in our mechanization, due to our three binding forms, all of those proofs were straightforward and were not difficult to discharge. The only proofs that were difficult were those that dealt with sets, which is mainly due to lack of good automation for working with sets in Rocq.

The astute reader might recognize that the rules defining System $\mathtt{F}_{<:\mathtt{C}}$ seem reminiscent of System $\mathtt{F}_{<:\mathtt{Q}}$. The author made the same realization when developing System $\mathtt{F}_{<:\mathtt{C}}$ which predates System $\mathtt{F}_{<:\mathtt{Q}}$, and is foreshadowing the common structure that we observed and will adopt later when we recast System $\mathtt{F}_{<:\mathtt{C}}$ in System $\mathtt{F}_{<:\mathtt{Q}}$ in Section 4.3.

Subcapturing $\boxed{\Gamma \vdash C <: D}$

$$\Gamma \vdash C <: \top \quad \text{(SC-UNIV)}$$

$$\frac{x \in C}{\Gamma \vdash \{x\} <: C} \quad \text{(SC-IN)}$$

$$\frac{\forall x \in C,\ \Gamma \vdash \{x\} <: D}{\Gamma \vdash C <: D} \quad \text{(SC-EXPAND)}$$

$$\frac{x : \{C\}\ S \in \Gamma \qquad \Gamma \vdash C <: D}{\Gamma \vdash \{x\} <: D} \quad \text{(SC-VAR)}$$

$$\frac{x <: C \in \Gamma \qquad \Gamma \vdash C <: D}{\Gamma \vdash \{x\} <: D} \quad \text{(SC-CVAR)}$$

Figure 4.6: Subcapturing rules of System $\mathtt{F}_{<:\mathtt{C}}$.

Subtyping $\boxed{\Gamma \vdash S_1 <: S_1 \text{ and } \Gamma \vdash T_1 <: T_2}$

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

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

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

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash \{C_1\}\ S_1 <: \{C_2\}\ S_2} \quad \text{(SUB-TYPE)}$$

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

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

$$\frac{\Gamma \vdash C_1 <: C_2 \qquad \Gamma, Y <: C_1 \vdash T_1 <: T_2}{\Gamma \vdash \forall(Y <: C_1).T_1 <: \forall(X <: C_2).T_2} \quad \text{(SUB-CALL)}$$

Figure 4.7: Subtyping rules of System $\mathtt{F}_{<:\mathtt{C}}$.

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x : \{C\}\ S \in \Gamma}{\Gamma \vdash x : \{x\}\ S} \quad (\textsc{var})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda(x).T_1 t : \{fv(t) - x\}\ (x : T_1) \to T_2} \quad (\textsc{abs})$$

$$\frac{\Gamma, X <: S \vdash t : T}{\Gamma \vdash \Lambda(X <: S).T : \{fv(t)\}\ \forall(X <: S).T} \quad (\textsc{s-abs})$$

$$\frac{\Gamma, Y <: C \vdash t : T}{\Gamma \vdash \Lambda(Y <: C).T : \{fv(t)\}\ \forall(Y <: C).T} \quad (\textsc{c-abs})$$

$$\frac{\Gamma \vdash t : \{D\}\ (x : \{C\}\ S) \to T \qquad \Gamma \vdash s : \{C\}\ S}{\Gamma \vdash t(C)s : T[x \mapsto C]} \quad (\textsc{app})$$

$$\frac{\Gamma \vdash t : \{D\}\ \forall(X <: S).T \qquad \Gamma \vdash S' <: S}{\Gamma \vdash t[S'] : T[X \mapsto S']} \quad (\textsc{s-app})$$

$$\frac{\Gamma \vdash t : \{D\}\ \forall(Y <: C).T \qquad \Gamma \vdash C' <: C}{\Gamma \vdash t[C'] : T[Y \mapsto C']} \quad (\textsc{c-app})$$

Figure 4.8: Typing rules for System $\mathtt{F}_{<:\mathtt{C}}$

Evaluation $\boxed{s \longrightarrow t}$

$$(\lambda(x).Tt)(C)v \longrightarrow t\,[x \mapsto_{\mathtt{type}} C]\,[x \mapsto_{\mathtt{term}} v] \quad (\textsc{beta-v})$$

$$(\Lambda(X <: S_b).t)[S] \longrightarrow t[X \mapsto S] \quad (\textsc{beta-s})$$

$$(\Lambda(Y <: C_b).t)[C] \longrightarrow t[Y \mapsto C] \quad (\textsc{beta-c})$$

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

$$\begin{array}{rcll} E & ::= & [] \mid E(C)t \mid v(C)E & \textbf{Evaluation Context} \\ & \mid & E[S] & \\ & \mid & E[C] & \end{array}$$

Figure 4.9: Reduction rules for System $\mathtt{F}_{<:\mathtt{C}}$

The Burden of Annotations?

One might object to the extra annotation required on term application. After all, one has to find a `C` – a capture set – to fill in there. However, this is not such an onerous requirement. As Brachthäuser et al. [14] observe, one infers the appropriate annotation anyways while type checking. Concretely, with the same typing rules, we could devise a calculus System $\texttt{F}_{\texttt{<:CO}}$, without reduction rules, but with almost the same syntax and typing rules, with the following two changes to syntax and typing:

1. Term application omits the capture set annotation – $f(v)$ instead of $f(C)v$.
2. The typing judgment is updated to reflect this:

$$\frac{\Gamma \vdash t : \{D\}\ x \to \{C\}\ ST \qquad \Gamma \vdash s : \{C\}\ S}{\Gamma \vdash t(s) : T[x \mapsto C]} \qquad \text{(APP-OMIT)}$$

At the surface level, System $\texttt{F}_{\texttt{<:CO}}$ looks almost like a dependently-typed calculus without capture annotations in the same style as Boruch-Gruszecki et al. [8]. This allows System $\texttt{F}_{\texttt{<:CO}}$ to serve as the basis for the surface-level syntax of a capture tracking system in a programming language, like Scala. It does not contain annotations like System $\texttt{F}_{\texttt{<:C}}$. But System $\texttt{F}_{\texttt{<:C}}$ can be used to show that well-typed System $\texttt{F}_{\texttt{<:CO}}$ programs behave properly as well. A typing derivation for a System $\texttt{F}_{\texttt{<:CO}}$ program is in one-to-one correspondence with a typing derivation for a System $\texttt{F}_{\texttt{<:C}}$ program – simply lift out the `C`s that appear in the (APP-OMIT) rule into the program itself. Progress and preservation in System $\texttt{F}_{\texttt{<:C}}$ guarantee that the resulting System $\texttt{F}_{\texttt{<:C}}$ program will reduce properly and not get stuck. Note, however, that the program will reduce according to the *inferred capture sets* on term application, unlike previous dependently-typed capture tracking calculi. This avoids the restrictions that Boruch-Gruszecki et al. [10] and Boruch-Gruszecki et al. [8] imposed on their systems.

4.3 Capture Sets as Qualifiers

As Boruch-Gruszecki et al. [10] and we have just shown, a capture tracking system, like the one presented earlier in Section 4.2.1, could be used to guarantee desirable and important safety invariants. These systems model capture tracking using sets of variables, but a set is just a lattice join of the singletons in that set! For example, Boruch-Gruszecki et al. [10] would give the following `evil_monologue`[1] function the capture set

[1] Scene from James Bond: The Travelling Salesman.

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

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

$$
(\lambda(x)_P.t)\{\!\{\ Q\ \}\!\}(s) \longrightarrow t[x \mapsto_{\texttt{type}} Q][x \mapsto_{\texttt{term}} s] \quad \text{(C-BETA-V)}
$$

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

$$
\frac{x : \{Q\}\ S \in \Gamma \qquad \Gamma \vdash Q \texttt{ <: } R}{\Gamma \vdash x \texttt{ <: } R}\ \text{(SQ-TVAR)}
$$

$$
\frac{x : \{Q\}\ S \in \Gamma}{\Gamma \vdash x \texttt{ <: } x}\ \text{(SQ-REFL-TVAR)}
$$

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

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

Figure 4.10: Evaluation, Syntax, Subtyping for System $\texttt{F}_{\texttt{<:QC}}$

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()
}
```

Perhaps this is not too surprising; after all, the subcapturing rules presented in Figure 2.1 are very similar to the rules we derived for System $\texttt{F}_{\texttt{<:Q}}$ in Figure 5.3. Using this insight, we can model capture tracking as an extension System $\texttt{F}_{\texttt{<:QC}}$ of System $\texttt{F}_{\texttt{<:Q}}$.

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.

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

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

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

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

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

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

Figure 4.11: Typing rules for System $\mathtt{F}_{<:\mathtt{QC}}$

Syntax – Tracking Variables Figure 4.10 defines the syntax of System $\mathtt{F}_{<:\mathtt{QC}}$. To reflect the underlying term-variable-based nature of capture tracking, term bindings in System $\mathtt{F}_{<:\mathtt{QC}}$ 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 typed $\{\bot\}\ (x : \{Q\}\ S) \to \{x\}\ S$ 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 $\mathtt{F}_{<:\mathtt{QC}}$ now requires a qualifier argument to be substituted for that variable in qualifier position. As such, term application in System $\mathtt{F}_{<:\mathtt{QC}}$ now has three arguments $s\{\!\{Q\}\!\}(t)$ – a function s, a qualifier Q, and an argument t; see Figure 4.10. In this sense, term abstractions in System $\mathtt{F}_{<:\mathtt{QC}}$ 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 $\texttt{F}_{\texttt{<:QC}}$'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)!

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

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 4.11.

Soundness Again, we can prove the standard soundness theorems for System $\texttt{F}_{\texttt{<:QC}}$, using the same ideas borrowed from System $\texttt{F}_{\texttt{<:C}}$.

Theorem 10 (Preservation for System $\texttt{F}_{\texttt{<:QC}}$)**.** *Suppose* $\Gamma \vdash s : T$*, and* $s \longrightarrow t$*. Then* $\Gamma \vdash t : T$ *as well.*

Theorem 11 (Progress for System $\texttt{F}_{\texttt{<:QC}}$)**.** *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* [10] 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 13 (Capture Prediction for System $\mathtt{F}_{<:\mathtt{QC}}$)**.** *Let* Γ *be an environment and* v *be a value such that* $\Gamma \vdash v : \{Q\}\ S$*. Then* $\Gamma \vdash \left\{\bigvee_{y \in fv(v)} y\right\}$ `<:` Q*.*

4.4 Function Colouring, Redux

Function colouring [68] 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 time 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 [68]. 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 $\mathtt{F}_{<:\mathtt{QA}}$, a calculus that enforces these restrictions. Additions to System $\mathtt{F}_{<:\mathtt{Q}}$ are highlighted in *grey*.

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

Figure 4.12: The syntax of System $\texttt{F}_{\texttt{<:QA}}$.

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`.

Syntax Figure 4.12 presents the modified syntax of System $\texttt{F}_{\texttt{<:QA}}$. 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 $\lambda(x)_{\texttt{async}}.\, x$, and an example of a function that is polymorphic in its qualifier is $\Lambda(Y \texttt{ <: sync})_{\texttt{async}}.\lambda(f)_Y.\, f(1)$, describing a function that should run in the same synchronicity context as its argument f.

Evaluation To model synchronicity safety, Figure 4.13 describes the operational semantics of System $\texttt{F}_{\texttt{<:QA}}$ using Felleisen and Friedman [30]-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.

$$
\langle 1, \texttt{app}\, \lambda(x)_{\bot}.x :: \kappa \rangle \longrightarrow \langle 1, \texttt{barrier}\, \bot :: \kappa \rangle
$$

reduces by placing a barrier marking a synchronous function on the stack.

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

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

$$\langle s(t), \kappa\rangle \longrightarrow \langle s, \mathtt{arg}\, t :: \kappa\rangle \ (\textsc{cong-app})$$

$$\langle v, \mathtt{arg}\, t :: \kappa\rangle \longrightarrow \langle t, \mathtt{app}\, v :: \kappa\rangle \ (\textsc{cong-arg})$$

$$\langle s[S], \kappa\rangle \longrightarrow \langle s, \mathtt{targ}\, S :: \kappa\rangle \ (\textsc{cong-tapp})$$

$$\langle s\{\!\{Q\}\!\}, \kappa\rangle \longrightarrow \langle s, \mathtt{qarg}\, Q :: \kappa\rangle \ (\textsc{cong-qapp})$$

$$\langle v, \mathtt{barrier}\, C :: \kappa\rangle \longrightarrow \langle v, \kappa\rangle \ (\textsc{break-barrier})$$

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

$$\frac{C \leq C_i \text{ for all } \mathtt{barrier}\ C_i \text{ frames on } \kappa \qquad \mathtt{eval}\, P = C}{\langle \Lambda(X \mathtt{<:}\ S)_P.t, \mathtt{targ}\, S' :: \kappa\rangle \longrightarrow \langle t[X \mapsto S'], \mathtt{barrier}\, C :: \kappa\rangle} \ (\textsc{reduce-tapp})$$

$$\frac{C \leq C_i \text{ for all } \mathtt{barrier}\ C_i \text{ frames on } \kappa \qquad \mathtt{eval}\, P = C}{\langle \Lambda(Y \mathtt{<:}\ Q)_P.t, \mathtt{qarg}\, Q' :: \kappa\rangle \longrightarrow \langle t[Y \mapsto Q'], \mathtt{barrier}\, C :: \kappa\rangle} \ (\textsc{reduce-qapp})$$

Figure 4.13: Operational Semantics (CK-style) for System $\mathtt{F}_{<:\mathtt{QA}}$

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

The other evaluation contexts are standard.

Typing To guarantee soundness, Figure 4.14 endows the typing rules of System $\mathtt{F}_{<:\mathtt{QA}}$ 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 change is again hilighted in *grey*. The 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.

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

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

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

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

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

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

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

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

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

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

Figure 4.14: Typing rules for System $\mathtt{F}_{<:\mathtt{QA}}$

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

Theorem 12 (Progress of System $\mathtt{F}_{<:\mathtt{QA}}$)**.** *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 13 (Preservation of System $\mathtt{F}_{<:\mathtt{QA}}$)**.** *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 $\mathtt{F}_{<:\mathtt{QA}}$: 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 $\mathtt{F}_{\mathtt{<:QA}}$ 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 $\mathtt{F}_{\mathtt{<:QA}}$ 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 later in this thesis in Chapter 6.

Chapter 5

Boolean-algebra-based Type Qualifiers

Now while lattices capture many applications for which type qualifiers are used, we may want to use type qualifiers in situations where negation, or *exclusion* is required. For example, in a system for effects, we may wish to allow all effects *except* a certain set which is unsafe; in short, to *exclude* a certain set of effects. Here, we may have a function which handles one specific exception, say, `IOException`, but lets others from its argument `f` pass through. `f` has exception set {`effs`}, denoted `Unit -> T  {effs}`.

```
def handleIOException[T][effs](f: Unit -> T \ {effs}):
    try:
        ....
    catch (IOException):
        ....
```

What should the type of `handleIOException` be? It will have the same effects as `f`, namely `effs`, except `IOException`. This is not expressible in a lattice – it requires *negation*, so we can write down the desired effect set of `handleIOException`, `effs - IOException`.

Lattices are not expressive enough to support this use case, but lattices coupled with support for distributivity and negation could. Previous work [60] has shown that we can support negation/exclusion but without the natural *subeffecting* that the lattice order gives rise to; clearly the set of effects that `handleIOException` could perform is contained within the set of effects that `f` can perform. Our effect system should handle both subeffecting

and negation – to reason that `effs - IOException` $\sqsubseteq$ `effs`. What we want is the order structure of a lattice coupled with *negation*; in other words, we want type qualifier systems grounded over *Boolean algebras*.

In this chapter, we introduce System $\mathtt{F}_{\mathtt{<:B}}$, a typed polymorphic calculus with support for Boolean algebra-based type qualifiers building on our polymorphic lattice-based qualifier System $\mathtt{F}_{\mathtt{<:Q}}$, and show how an effect system with support for effect exclusion can be encoded as an extension of our system.

5.1 Boolean Algebras

Before we discuss our System $\mathtt{F}_{\mathtt{<:B}}$ calculus extending System $\mathtt{F}_{\mathtt{<:Q}}$, we will first introduce the notion of a *boolean algebra* and the *free boolean algebra* of boolean expressions over that algebra. First, we will introduce the order-theoretic definition of a boolean algebra.

Definition 9. *A boolean algebra $\mathcal{B}$ is a ground set B, coupled with binary operations $\sqcap, \sqcup, \neg$ as well as a binary relation $\sqsubseteq$ satisfying:*

- *$\mathcal{B}$ is a lattice.*
- *$\mathcal{B}$ is a distributive lattice, namely for all $b_1, b_2, b_3 \in B$ we have that $b_1 \sqcap (b_2 \sqcup b_3) = b_1 \sqcap b_2 \sqcup b_1 \sqcap b_3$.*
- *For all elements b_1 in B, $b_1 \sqcap \neg b_1 \sqsubseteq \bot$, and $\top \sqsubseteq b_1 \sqcup \neg b_1$.*

Definition 10. *Let X be a set of variables. Then $\mathcal{E}_B(X)$, the set of boolean algebra formulas generated by X, is recursively defined by:*

1. *If $x \in X$ then $x \in \mathcal{E}(X)$.*
2. *If $f \in \mathcal{E}_B(X)$ then so is $\neg f_1 \in \mathcal{E}_B(X)$.*
3. *If $f_1 \in \mathcal{E}_B(X)$ and $f_2 \in \mathcal{E}(X)$ then $f_1 \wedge f_2 \in \mathcal{E}(X)$.*
4. *If $f_1 \in \mathcal{E}_B(X)$ and $f_2 \in \mathcal{E}(X)$ then $f_1 \vee f_2 \in \mathcal{E}(X)$.*

Definition 11. *Let X be a set of variables. Then $\mathcal{F}_B(X)$, the free boolean algebra generated by X, is the lattice over $\mathcal{E}_B(X)$ 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 $f_1[X \to \overline{B}] \sqsubseteq f_2[X \to \overline{B}]$ in every boolean algebra $\mathcal{B}$ and instantiation $\overline{B}$ of the variables in X to elements of $\mathcal{B}$, up to equivalence modulo $\leq$.*

Now while Definition 11 defines the free boolean algebra extrinsically by its universal property, it unfortunately does not give a constructive definition for the free algebra. However, it is folklore that the following explicit construction gives rise to the free boolean algebra as well, using techniques similar to what [67] used to show a similar result for free lattices.

Theorem 14 (Folklore)**.** *$\mathcal{F}_B(X)$ is the lattice over $(\mathcal{E}_B(X)/\leq, \leq)$, where $\leq$ is a binary relation over $\mathcal{E}(X)$ defined by:*

1. *For every $f \in \mathcal{E}_B(X)$, $f \leq f$.*
2. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$, if $f_3 \leq f_1$ then $f_3 \leq f_1 \vee f_2$.*
3. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$, if $f_3 \leq f_2$ then $f_3 \leq f_1 \vee f_2$.*
4. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$, if $f_3 \leq f_1$ and $f_3 \leq f_2$ then $f_1 \vee f_2 \leq f_3$.*
5. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$, if $f_1 \leq f_3$ then $f_1 \wedge f_2 \leq f_3$.*
6. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$, if $f_2 \leq f_3$ then $f_1 \wedge f_2 \leq f_3$.*
7. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$, if $f_3 \leq f_1$ and $f_3 \leq f_2$ then $f_3 \leq f_1 \wedge f_2$.*
8. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$, if $f_1 \leq f_2$ and $f_2 \leq f_3$ then $f_1 \leq f_3$.*
9. *For every $f_1, f_2, f_3 \in \mathcal{E}_B(X)$ we have that $f_1 \wedge (f_2 \vee f_3) \leq f_1 \wedge f_2 \vee f_1 \wedge f_3$.*
10. *For every $f \in \mathcal{E}_B(X)$, $f \wedge \neg f \leq \bot$.*
11. *For every $f \in \mathcal{E}_B(X)$, $\top \leq f \vee \neg f$.*

Here, the additional rules defining a Boolean algebra over a lattice are highlighted in grey.

Proof. To see that $\leq$ defines a boolean algebra over $\mathcal{E}_B(X)/\leq$ we need to show that $\leq$ defines a partial order and that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{E}_B(X)$ with respect to $\leq$.

By definition, $\leq$ is antisymmetric over $\mathcal{E}_B(X)/\leq$ as we are quotienting out terms equivalent under $\leq$. Moreover, by definition – rules (1) and (8) – $\leq$ is reflexive and transitive. Hence $\leq$ defines a partial order over $\mathcal{E}_B(X)/\leq$.

Now, to see that $\leq$ defines a boolean algebra, we need to show that it defines a distributive lattice as well as that negation is well behaved – namely, for any $f \in \mathcal{E}_B(X)$, $\top \leq f \vee \neg f$ and that $\bot \leq f \wedge \neg f$.

To show that $\leq$ defines a lattice, we need to show that $\wedge$ and $\vee$ define the meet and join respectively of any two terms in $\mathcal{E}(\mathcal{X})$ with respect to $\leq$. Now, to see for two lattice terms f_1 and f_2 that $f_1 \vee f_2$ and $f_1 \wedge f_2$ are lower and upper bounds for both Q and R, one can apply the join introduction and the meet elimination rules. It remains to show that they are the greatest lower bounds and the least upper bounds – but that follows directly from an application of rules (2), (3), and (7), as desired.

To show that $\leq$ defines a distributive lattice, we need to show for all terms that $f_1 \wedge (f_2 \vee f_3) = f_1 \wedge f_2 \vee f_1 \wedge f_3$. One half of this equality holds, by rule (9). The other half holds as $f_1 \wedge f_2 \leq f_1$ and $f_1 \wedge f_2 \leq (f_2 \vee f_3)$, and symmetrically $f_1 \wedge f_3 \leq f_1$ and $f_1 \wedge f_3 \leq (f_2 \vee f_3)$.

Finally, to show that $\leq$ defines a boolean algebra, we need to show that for all f, $\bot \leq f \wedge \neg f$ and $\top \leq f \vee \neg f$ – but these are rules (10) and (11).

Thus, we have that $(\mathcal{E}_B(X)/\leq, \leq)$ forms a boolean algebra. Finally, to show that it is the free Boolean algebra over X, we need to show that for two well-formed boolean algebra equations $f_1[X]$ and $f_2[X]$ in $\mathcal{E}_B(X)$ such that $f_1[X] \leq f_2[X]$, it follows that for any Boolean algebra B and any instantiation of the variables $\overline{X}$ to $\overline{B}$ respecting the bounds in Γ, we have that $f_1[X \to \overline{B}] \sqsubseteq f_2[X \to \overline{B}]$. But all of the rules defining $\leq$ hold in every Boolean algebra. Hence $\mathcal{F}_B(X)$ is the free boolean algebra. □

5.2 Qualified Types with Boolean Algebras

With the free Boolean algebra defined via its inference rules from Theorem 14, we are now ready to define System $\mathtt{F}_{\mathtt{<:B}}$, which extends the subqualification rules of System $\mathtt{F}_{\mathtt{<:Q}}$ using the additional inference rules from Theorem 14.

Syntax Figure 5.1 presents the syntax of System $\mathtt{F}_{\mathtt{<:B}}$. Importantly, we now allow negation and base qualifiers from a base Boolean algebra B, equipped with distinguished values $\top$ and $\bot$, operations meet ($\sqcap$), join ($\sqcup$), negation ($\neg$), and with ordering $\sqsubseteq$. We reflect these operations in the syntax of qualifiers with $\wedge$, $\vee$, and $\sim$ respectively.

Values and Qualifiers Following (and unchanged from) System $\mathtt{F}_{\mathtt{<:Q}}$, we *tag* values with a qualifier expression P denoting the qualifier that value should be typed at and we add

s, t	::=	**Terms**	
|	n_P	integer values	
|	$\lambda(x)_P.t$	term abstraction	
|	x	term variable	
|	$s(t)$	term application	
|	$\Lambda(X \texttt{<:}\ S)_P.t$	type abstraction	
|	$\Lambda(Y \texttt{<:}\ Q)_P.t$	qualifier abstraction	
|	$s[S]$	type application	
|	$s\{\!\{Q\}\!\}$	qualifier application	
|	$\texttt{upqual}\ Q\ s$	qualifier upcast	
|	$\texttt{assert}\ Q\ s$	qualifier assertion	
Γ	::=	**Environment**	
|	$\cdot$	empty	
|	$\Gamma,\ x : T$	term binding	
|	$\Gamma,\ X <: S$	type binding	
|	$\Gamma,\ Y <: Q$	qualifier binding	
v	::=	**Values**	
|	n_P		
|	$\lambda(x)_P.t$		
|	$\Lambda(X \texttt{<:}\ S)_P.t$		
|	$\Lambda(Y \texttt{<:}\ Q)_P.t$		

S	::=	**Simple Types**
|	$\top$	top type
|	`int`	integer type
|	$T_1 \to T_2$	function type
|	X	type variable
|	$\forall(X \texttt{<:}\ S).T$	for-all type
|	$\forall(Y \texttt{<:}\ Q).T$	qualifier for-all type
T	::=	**Qualified Types**
|	$\{Q\}\ S$	qualified type
P, Q, R	::=	**Qualifiers**
|	b	base elements
|	Y	qualifier variables
|	$Q \wedge R \mid Q \vee R$	meets and joins
|	$\sim Q$	negation
C	::=	**Concrete Qualifiers**
|	b	base elements

Figure 5.1: The syntax of System $\texttt{F}_{\texttt{<:B}}$, as an extension of System $\texttt{F}_{\texttt{<:Q}}$. Changes are highlighted in grey.

Evaluation for System $\mathtt{F}_{<:\mathtt{B}}$ $\boxed{s \longrightarrow t}$ and $\boxed{\mathtt{eval}\, Q}$

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

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

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

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

$$\frac{v \text{ tagged with } P \qquad \mathtt{eval}(P) \sqsubseteq \mathtt{eval}(Q)}{\mathtt{assert}\, Q\ v \longrightarrow v} \quad \text{(ASSERT)}$$

$$\frac{s \longrightarrow t}{E[s] \longrightarrow E[t]} \quad \text{(CONTEXT)}$$

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

$$\begin{array}{rcll} \mathtt{eval}(P) & ::= & & \textbf{Partial Qualifier Evaluation} \\ & | & C & => \quad C \\ & | & P \wedge R & => \quad \mathtt{eval}(P) \sqcap \mathtt{eval}(R) \\ & | & P \vee R & => \quad \mathtt{eval}(P) \sqcup \mathtt{eval}(R) \\ & | & \sim P & => \quad \neg\, \mathtt{eval}(P) \end{array}$$

Figure 5.2: Reduction rules for System $\mathtt{F}_{<:\mathtt{B}}$, as an extension of System $\mathtt{F}_{<:\mathtt{Q}}$. Changes are highlighted in gray.

support for *asserting*, as well as *upcasting*, qualifier tags. This is to equip our base calculus with some meaningful runtime semantics. For example, the value $\lambda(x : \mathtt{Int})_\top x$ would represent the integer identity function qualified at $\top$. For brevity, we say "v tagged with P" to mean that the runtime tag attached to v is P; here, $\lambda(x : \mathtt{Int})_\top x$ is tagged with $\top$.

Semantics The evaluation rules of System $\mathtt{F}_{<:\mathtt{B}}$ in Figure 5.2 remain largely unchanged from System $\mathtt{F}_{<:\mathtt{Q}}$, except that qualifier evaluation is extended to account for negation. Note that while `eval` is partial and is only defined in terms of concrete qualifier expressions without variables, it will only be used on concrete qualifier expressions during reduction.

Subqualification System $\mathtt{F}_{<:\mathtt{B}}$ extends System $\mathtt{F}_{<:\mathtt{Q}}$'s subqualification rules to account for distributivity via (SQ-DIST) and negation via (SQ-NEG-INTRO) and (SQ-NEG-ELIM), shown

Subqualification for System $\mathtt{F}_{<:\mathtt{B}}$ $\boxed{\Gamma \vdash Q <: R}$

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

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

$$\frac{\Gamma \vdash Q <: R_1}{\Gamma \vdash Q <: R_1 \vee R_2}\ \text{(SQ-JOIN-INTRO-1)}$$

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

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

$$\Gamma \vdash P \wedge (Q \vee R) <: (P \wedge Q) \vee (P \wedge R)\ \text{(SQ-DIST)}$$

$$\Gamma \vdash \top <: Q \vee (\sim Q)\ \text{(SQ-NEG-INTRO)}$$

$$\Gamma \vdash Q \wedge (\sim Q) <: \bot\ \text{(SQ-NEG-ELIM)}$$

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

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

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

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

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

$$\frac{\Gamma \vdash P <: Q \qquad \Gamma \vdash Q <: R}{\Gamma \vdash P <: R}\ \text{(SQ-TRANS)}$$

$$\frac{b_1, b_2 \in B \qquad l_1 \sqsubseteq l_2}{\Gamma \vdash b_1 <: b_2}\ \text{(SQ-LIFT)}$$

Figure 5.3: Subqualification rules of System $\mathtt{F}_{<:\mathtt{B}}$, as an extension of System $\mathtt{F}_{<:\mathtt{Q}}$. Changes are highlighted in gray.

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

$$\frac{}{\Gamma \vdash n_P : \{P\}\ \mathtt{Int}} \quad (\textsc{int})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\textsc{var})$$

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

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

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

$$\frac{\Gamma \vdash t : \{Q\}\ S \qquad \Gamma \vdash Q <: P}{\Gamma \vdash \mathtt{assert}\ P\ t : \{Q\}\ S} \quad (\textsc{typ-assert})$$

$$\frac{\Gamma \vdash t : \{Q\}\ T_1 \to T_2 \qquad \Gamma \vdash s : T_1}{\Gamma \vdash t(s) : T_2} \quad (\textsc{app})$$

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

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

$$\frac{\Gamma \vdash s : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash s : T_2} \quad (\textsc{sub})$$

$$\frac{\Gamma \vdash t : \{Q\}\ S \qquad \Gamma \vdash Q <: P}{\Gamma \vdash \mathtt{upqual}\ P\ t : \{P\}\ S} \quad (\textsc{typ-upqual})$$

Figure 5.4: Typing rules for System $\mathtt{F}_{<:\mathtt{B}}$ (and of System $\mathtt{F}_{<:\mathtt{Q}}$).

in Figure 5.3. These additional rules extend the free lattice axioms of System $\mathtt{F}_{<:\mathtt{Q}}$ to the rules necessary to capture the structure of a free Boolean algebra; it is straightforward to show by induction that these rules define a Boolean algebra (modulo bounds on free variables), and to show that the resulting Boolean algebra embeds in any other (modulo bounds on free variables). Note that transitivity is now an explicit rule in System $\mathtt{F}_{<:\mathtt{B}}$ as the new Boolean algebra rules do not easily admit integrated transitivity; in particular, the rule for distributivity does not play well with integrated transitivity, as [66] observed in a setting where transitivity was integrated with distributivity for *subtyping*.

It is also worth noting that only one-sided distributivity is required; the other distributivity rule can be proven from the one given (as we do in our mechanization). This reflects the additional rules in Theorem 14 compared to Theorem 1.

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

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

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

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

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

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

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

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

Figure 5.5: Subtyping rules of System $\mathtt{F}_{<:\mathtt{B}}$ (and of System $\mathtt{F}_{<:\mathtt{Q}}$).

Subtyping and Typing Finally System $\mathtt{F}_{<:\mathtt{B}}$ inherits all of System $\mathtt{F}_{<:\mathtt{Q}}$'s subtyping and typing rules; all of its changes are in the structure and semantics of its type qualifiers, not in the structure of its types or terms. For completeness we reproduce these rules in Figures 5.4 and 5.5.

5.3 Metatheory

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

Theorem 15 (Preservation)**.** *Suppose* $\Gamma \vdash s : T$*, and* $s \longrightarrow t$*. Then* $\Gamma \vdash t : T$ *as well.*

Theorem 16 (Progress)**.** *Suppose* $\varnothing \vdash s : T$*. Then either* s *is a value, or* $s \longrightarrow t$ *for some term* t*.*

Proof. Can be found in the mechanization accompanying this thesis. □

5.4 Application: Polymorphic Effect Exclusion

To demonstrate System $\mathtt{F}_{<:\mathtt{B}}$, we apply it to model an effect safety system with support for *effect exclusion*, using it as a basis of an effect calculus System $\mathtt{F}_{<:\mathtt{BE}}$. System $\mathtt{F}_{<:\mathtt{BE}}$ itself

is based on System $\mathtt{F}_{\mathtt{<:QA}}$, introduced earlier in this thesis to support a function colouring system, but we extend it on top of System $\mathtt{F}_{\mathtt{<:B}}$ to support effect exclusion.

Modelling Effects and Assigning Qualifiers Effects will be drawn from a fixed base (but possibly infinite) Boolean algebra B; effectful terms like abstractions will be qualified with the effect b drawn from B they are allowed to perform. To model the act of performing an effect in the calculus, we introduce a $\mathtt{do}\, b$ expression, which performs the effect labelled by b. In the operational semantics of System $\mathtt{F}_{\mathtt{<:BE}}$, $\mathtt{do}\, b$ simply returns a value immediately after checking that the effect b is allowed by the current evaluation context; in a real implementation an effectful operation would also be performed.

Syntax Figure 5.6 presents the additional syntax of System $\mathtt{F}_{\mathtt{<:BE}}$ relative to System $\mathtt{F}_{\mathtt{<:B}}$ with support for performing effectful operations and excluding effects from terms. Qualifier annotations, as noted above, are reused on abstraction terms to denote the set of possible effects an abstraction may perform.

Evaluation To model effect safety, Figure 5.7 describes the operational semantics of System $\mathtt{F}_{\mathtt{<:BE}}$ using Felleisen and Friedman [30] -style CK semantics; namely, with a machine modelling an evaluation context/stack (**K**) surrounding an expression under evaluation. To model effects and effect safety, we extend the standard frames of CK with special *barrier* frames installed on the stack denoting the effect of the function that was called, as we did earlier with System $\mathtt{F}_{\mathtt{<:QA}}$, and *fences* excluding effects from evaluation contexts.

$\mathtt{fence}\, b$ frames are used to mark the extent of an evaluation context where the effects labelled by b cannot be performed. Placing a barrier frame $\mathtt{barrier}\, b$ on top of a $\mathtt{fence}\, d$ frame fails unless $b \sqsubseteq \neg d$. Finally, to ensure effect safety, effects can only be performed by $\mathtt{do}\, b$ if b is compatible – contained within – all `barrier` frames on the stack and disjoint from all `fence` frames on the stack.

Typing To guarantee soundness, Figure 5.8 endows the typing rules of System $\mathtt{F}_{\mathtt{<:BE}}$ with modified rules for keeping track of the effect context abstractions need. We extend the typing rules with an effect context R to keep track of the effects a function is allowed to perform. This effect context R is simply a qualifier expression, and is introduced by the rules for typing abstractions by lifting the qualifier tagged on those abstractions – see rules (A-ABS), (A-T-ABS), and (A-Q-ABS).

t, c	$::=$		**Terms**	f	$::=$		**Evaluation Frames**
	$\ldots$		as before, except:		$\mid$	$\texttt{barrier}\, C$	barrier
	$\mid$	$\texttt{without}\, C\ t$	exclusion fence		$\mid$	$\texttt{fence}\, C$	fence
	$\mid$	$\texttt{do}\ b$	effectful operation		$\mid$	$\texttt{arg}\, t$	argument
κ	$::=$		**Evaluation Context**		$\mid$	$\texttt{app}\, v$	application
	$\mid$	$[]$			$\mid$	$\texttt{targ}\, T$	type application
	$\mid$	$f :: \kappa$			$\mid$	$\texttt{qarg}\, Q$	qualifier application

Figure 5.6: The syntax of System $\texttt{F}_{\texttt{<:QA}}$, based off of System $\texttt{F}_{\texttt{<:QA}}$. Non-standard context frames highlighted in grey.

To ensure effect safety, both the elimination forms for abstraction application and effect application check the effect context to ensure that the effects of the operation or function invocation are compatible with the effects allowed by the current context.

Metatheory With all this, we can state and prove progress and preservation for System $\texttt{F}_{\texttt{<:BE}}$.

Theorem 17 (Progress of System $\texttt{F}_{\texttt{<:BE}}$)**.** *Suppose $\langle s, \kappa \rangle$ is a well-typed machine configuration. Then either c is a value and k is the empty continuation, or there is a machine state $\langle t, \kappa' \rangle$ that it steps to.*

Theorem 18 (Preservation of System $\texttt{F}_{\texttt{<:BE}}$)**.** *Suppose $\langle s, \kappa \rangle$ is a well-typed machine configuration. If it steps to another configuration $\langle t, \kappa' \rangle$, then that configuration is also well-typed.*

Note that progress and preservation guarantee meaningful safety properties about System $\texttt{F}_{\texttt{<:BE}}$; effectful operations can only be invoked in contexts where they are allowed.

5.5 Mechanization

The proofs of the standard soundness theorems of our calculi System $\texttt{F}_{\texttt{<:B}}$ and System $\texttt{F}_{\texttt{<:BE}}$ have been fully mechanized in Rocq, with inspiration drawn from the mechanization of System $\texttt{F}_{\texttt{<:}}$ by Aydemir et al. [3].

The proof artifacts can be found at https://github.com/e45lee/phd-thesis-proofs.

Evaluation for System $\mathtt{F}_{\mathtt{<:QA}}$ $\boxed{\langle s, \kappa\rangle \longrightarrow \langle t, \kappa'\rangle}$

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

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

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

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

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

$$\langle v, \mathtt{fence}\, C :: \kappa\rangle \longrightarrow \langle v, \kappa\rangle \text{ (JUMP-FENCE)}$$

$$\langle \mathtt{without}\, C\ t, \kappa\rangle \longrightarrow \langle t, \mathtt{fence}\, C :: \kappa\rangle \text{ (REDUCE-FENCE)}$$

$$\frac{\begin{array}{c} C \sqsubseteq C_i \text{ for all } \mathtt{barrier}\ C_i \text{ frames on } \kappa \\ \mathtt{eval}\, P = C \\ C \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\ C_i \text{ frames on } \kappa \end{array}}{\langle v, \mathtt{app}\, \lambda(x)_P.t :: \kappa\rangle \longrightarrow \langle t[x \mapsto v], \mathtt{barrier}\, C :: \kappa\rangle} \text{ (REDUCE-APP)}$$

$$\frac{\begin{array}{c} C \sqsubseteq C_i \text{ for all } \mathtt{barrier}\ C_i \text{ frames on } \kappa \\ \mathtt{eval}\, P = C \\ C \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\ C_i \text{ frames on } \kappa \end{array}}{\begin{array}{c} \langle \Lambda(X <: S)_P.t, \mathtt{targ}\, S' :: \kappa\rangle \longrightarrow \\ \langle t[X \mapsto S'], \mathtt{barrier}\, C :: \kappa\rangle \end{array}} \text{ (REDUCE-TAPP)}$$

$$\frac{\begin{array}{c} C \sqsubseteq C_i \text{ for all } \mathtt{barrier}\ C_i \text{ frames on } \kappa \\ \mathtt{eval}\, P = C \\ C \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\ C_i \text{ frames on } \kappa \end{array}}{\begin{array}{c} \langle \Lambda(Y <: Q)_P.t, \mathtt{qarg}\, Q' :: \kappa\rangle \longrightarrow \\ \langle t[Y \mapsto Q'], \mathtt{barrier}\, C :: \kappa\rangle \end{array}} \text{ (REDUCE-QAPP)}$$

$$\frac{\begin{array}{c} b \sqsubseteq C_i \text{ for all } \mathtt{barrier}\ C_i \text{ frames on } \kappa \\ b \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\ C_i \text{ frames on } \kappa \end{array}}{\langle \mathtt{do}\, b, \kappa\rangle \longrightarrow \langle 1, \kappa\rangle} \text{ (REDUCE-DO)}$$

Figure 5.7: Operational Semantics (CK-style) for System $\mathtt{F}_{\mathtt{<:BE}}$ (based off System $\mathtt{F}_{\mathtt{<:QA}}$). Modifications highlighted in *grey*.

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

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

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

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

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

$$\frac{\Gamma \vdash R <: \sim b \qquad \Gamma \mid R \vdash s : T}{\Gamma \mid R \vdash \mathtt{without}\, b\ s : T} \text{ (A-WITHOUT)}$$

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

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

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

$$\Gamma \mid b \vdash \mathtt{do}\, b : \{\bot\}\ \mathtt{int} \quad \text{(A-DO)}$$

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

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

Figure 5.8: Typing rules for System $\mathtt{F}_{<:\mathtt{BE}}$ (based off of System $\mathtt{F}_{<:\mathtt{QA}}$)). Changes from System $\mathtt{F}_{<:\mathtt{B}}$ are highlighted in *grey*.

Chapter 6

Related Work

6.1 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. [31] 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 [70]. 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 [49]. Additionally, the *Clean* programming language modelled *uniqueness* as a type qualifier with support for polymorphism by *constrained uniqueness schemes* by 1993 [6]. The work on Clean predates Foster et al. [31] 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. [10] 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. [90] building off of Bao et al. [5] recently developed a qualifier system for tracking reachability using variable sets as well. Like Boruch-Gruszecki et al. [10], 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, using an extension of our boolean algebra qualifier system developed in Chapter 5. They handle freshness as an ad-hoc qualifier extension ⧫/`fresh` to their system with special subqualification rules; we conjecture that a combination of polymorphism and negation could accomplish the same task. Concretely, for example, instead of `malloc` returning a fresh qualifier in their system, `malloc` could return a new qualifier, existentially quantified, disjoint from an existing set of qualifiers passed in to it.

```
def malloc[T](): {fresh} T = ???
def malloc[T][E](): exists L <: (not E), {L} T = ???
```

They are currently working on extending their system to work over free join-semilattice terms though.[1] It remains future work to extend our qualifier system for support for existentials.

Boolean Formulas as Qualifiers Madsen and van de Pol [62] recently investigated modelling *nullability* as a type qualifier. Types in their system comprise a scheme of type

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

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`.[2] 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 [62] as an ML-polymorphism style extension of Foster et al. [31] that solves the problem of encoding qualifier constraints: one can just encode them using Boolean formulas.

Notions of Subqualification Notions of qualifier polymorphism and subqualification have been worked on in many recent papers, even if only implicitly developed or discussed in the context of other works above. Unlike other works, however, Haack and Poll [40], explicitly construct a calculus which features a restricted variant of qualifier polymorphism and subqualification, and explicitly call out both. We believe that they are the first to call out the concepts of qualifier polymorphism and subqualification; they do so in an immutability system for Java. Their system features a restricted variant of subqualification; unlike ours, bounds on a polymorphic qualifier variable are restricted to come from a fixed set `{Any, Writeable, Qual}`. However, they do model a notion of freshness using qualifiers `Fresh(n)` that grant access to write newly allocated objects, using mechanisms similar to how Wei et al. [90] handle freshly allocated objects as well. It would be interesting future work to model this in the setting of fully-polymorphic type qualifiers.

Reference Immutability for C# [38] Of existing qualifier systems, the polymorphism structure of Gordon et al. [38] is closest to System $\mathtt{F}_{<:\mathtt{Q}}$. Polymorphism is possible over *both* mutability qualifiers and simple types in Gordon's system, but must be done separately, as in System $\mathtt{F}_{<:\mathtt{Q}}$. 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:

[2] 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.

```
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 [28] roDOT extends the calculus of Dependent Object Types [2] 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 $\mathtt{F}_{<:\mathtt{Q}}$'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 $\mathtt{F}_{<:\mathtt{Q}}$. 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. [71], Xhebraj et al. [93], Zhao [95]. 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.

Overloading For binary qualifiers (such as `@readonly`/`@mutable`), one can implement polymorphism by duplicating function and method implementations for both possible instantiations. This was pointed out by Huang et al. [43] in their work for mutability qualifiers. For example, consider the following top-level parametric function, `access`, which is parametric on the mutability variable `M`:

```
def access[M](z: [M] Pair[Pair[Int]]): M Pair[Int] = { z.first }
```

This function can be rewritten instead as two functions with the same name `access`, one taking in a regular, mutable pair, and one taking in a readonly pair:

```
def access(z: Pair[Pair[Int]]): Pair[Int] = { z.first }
def access(@readonly z: Pair[Pair[Int]]): @readonly Pair[Int]
    = { z.first }
```

The Checker Framework [23] uses this strategy of overloading to implement polymorphism across other qualifier kinds as well: `@PolyNull`, `@PolyRegex`, etc. A qualifier in a function `@PolyNull|Regex|Read` is treated implicitly as a single qualifier variable given to that function and desugared accordingly. So we could instead write:

```
def access(z: @PolyRead Pair[Pair[Int]]): @PolyRead Pair[Int] =
    { z.first }
```

6.2 Languages with Type Qualifier Systems

Rust The Rust community is currently investigating approaches [92] 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> { ... }
```

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

OCaml The OCaml community [85, 58] 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* [20] 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.

Java Java, as of JSR 308 [16], has integrated support for type qualifiers via type annotations. While Java only specifies a few builtin annotations (including an effect/flow system for tracking calls to `@Deprecated` methods), tools such as the The Checker Framework [72, 23] build on top Java's type annotations to support type qualifiers in general. The Checker Framework generally allows for qualifying type variables with qualifiers, but in their system, there is no qualifier 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, so a `@mutable X` instantiated with `X |-> @const List` becomes a `@mutable List`. The Checker Framework has been used to implement many qualifier systems, including some more exotic ones tracking *units* [94] and information flow [29]. The Checker Framework has also been used to model effect systems as well [37].

6.3 Implementing Type Qualifiers

Boolean Algebras without Subqualification Dually to work on subtyping with type qualifiers, Boolean algebras and formulas have also been used to enrich types to capture additional invariants about a program. Unlike the work on type qualifiers, which has a long history, the work done on Boolean algebras has been more recent. For example, only recently have Boolean algebras been used to model (non)-nullable references [62] and effects [60], in comparison to a long line of work done on nullability in the context of type qualifiers. As they note, one of the limitations of their work is that subqualification on Boolean qualifiers was an open problem [62, Conjecture 4.18]. We answer their conjecture affirmatively, and we hope that our work makes Boolean algebra based type qualifiers more accessible to other language designers going forward.

Boolean Unification Algorithms Inference of Boolean effects requires Boolean unification algorithms for solving the subqualification constraints that are generated as part of the type checking process. Boolean formula unification algorithms date back as far as Boolean values themselves, as seen in Boole [7]'s original work, and [61]'s turn of the century book. More modern work includes that of Rudeanu [79] and Buttner and Simonis [17]. Boudet et al. [11] investigate unification on general Boolean algebras and rings. Finally, of particular interest to implementers will be Baader [4]'s study of the computational complexity of Boolean unification algorithms.

Algebraic Subtyping Closely related to lattices and Boolean algebras is recent work on *algebraic subtyping* first by Dolan [27] and more recently by Parreaux [73] and Parreaux and Chau [74]. In contrast to our approach, where we lightly augment the existing subtyping lattice of System $\mathtt{F}_{<:}$ with support for boolean-algebra valued type qualifiers, the literature on algebraic subtyping is focused on integrating the algebraic theory of lattices into the subtyping structure directly. While their approach is more expressive – for example, with records, they are able to reason about intersections, unions, and negations of records, while we only propose using qualifiers to indicate what fields are present – the changes needed to support a full boolean algebra based subtyping lattice are more heavyweight than what we propose here.

6.4 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 [65], whereas traditionally it would be viewed as an effect annotation. In contrast to type qualifiers, both *effect polymorphism* [59] and the *lattice structure of effects* [80] 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 [56]. As for Rytz et al. [80], 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 [39, 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 of Leijen and Tate [57] can be viewed as a variant of the lattice-based subqualification system of Foster et al. [31] with HM(X)-style polymorphism. More interestingly, the novel Indirect-Callε rule of [35], the reachability rule of Wei et al. [90], and the subcapturing rule of [10] all model subqualification in a free join-semilattice (of effects). We have presented an effect system in Chapter 5 using the algebraic structure of Boolean algebras which demonstrates how an effect system with subeffecting can be neatly implemented using Boolean-algebra based qualifiers. It remains interesting future work to connect qualifiers with effects where *the order that effects are performed in* matters; for example, in locking, where the order of operations do matter [36].

6.5 Algorithmic Subtyping

System $\texttt{F}_{\texttt{<:Q}}$'s subtyping rules are syntax-directed and admit algorithmic rules, but it is not so easy to see if extended System $\texttt{F}_{\texttt{<:Q}}$ or System $\texttt{F}_{\texttt{<:B}}$ admits algorithmic subtyping rules. The difficulty is that extended System $\texttt{F}_{\texttt{<:Q}}$ and System $\texttt{F}_{\texttt{<:B}}$ need two new non-syntax directed rules (SQ-EVAL-ELIM) and (SQ-EVAL-INTRO) to handle transitivity through base lattice or algebra elements. It remains an open question whether extended System $\texttt{F}_{\texttt{<:Q}}$ or System $\texttt{F}_{\texttt{<:B}}$ admits algorithmic subtyping rules. We conjecture that algorithmic subtyping rules could exist for a particular instantiation of extended System $\texttt{F}_{\texttt{<:Q}}$ or System $\texttt{F}_{\texttt{<:B}}$ 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.

6.6 Flow Sensitivity

Foster et al. [32] extended the original work of Foster et al. [31] with support for *flow sensitivity* on type qualifiers, and they have been implemented in the Checker Framework [72]. Even though flow sensitivity can be sometimes avoided, for example, with *pattern matching* as Madsen and van de Pol [62] show with `nullable` as a qualifier, flow sensitivity is a natural addition to many qualifier systems. One often checks if a variable `x` is not `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. [32] use to extend their system to support flow sensitivity can also be used to add flow sensitivity to

System $\mathtt{F}_{<:\mathtt{Q}}$. 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 [36].

6.7 Viewpoint Adaptation and Immutability Systems

While we have already discussed related reference immutability systems earlier in Chapter 2, there is still related work that we have yet to discuss below on *viewpoint adaptation* and on implementations of reference immutability systems.

6.7.1 Viewpoint Adaptation

Viewpoint adaptation has been used in reference immutability systems to denote the type-level adaptation which is enforced to guarantee transitive immutability safety. When a field $r.f$ is read from some record r, the mutability of the resulting reference needs to be adapted from *both* the mutability of r and from the type of f in the record itself. While this notion of adaptation was known as early as Javari [89], the term "viewpoint adaptation" was first coined by Dietl et al. [24]. They realized that this notion of adaptation could be generalized to arbitrary qualifiers – whether or not the type of a field read $r.f$ should be qualified by some qualifier $@q$ should depend on whether or not f's type is qualified and whether or not r's type is qualified as well.

6.7.2 Generic Universe Types

Viewpoint adaptation has also been adapted to other use cases. Dietl et al. [24] and Huang et al. [42] use viewpoint adaptation and type qualifiers to implement an *ownership* system for Java references in order to tame *aliasing* in Java programs. Unlike viewpoint adaptation for mutability, viewpoint adaptation for ownership follows a different algebraic structure that is not lattice-like. Here, qualifiers `rep` represent an object that is owned by the current object `this`, and `peer` represent an object that is owned by the owner for the current object `this`.

Moreover, viewpoint adaptation in this scenario is defined in a way that is not amenable to commutative lattice operations: `rep |> peer = rep` but `peer |> rep = any`. It would be interesting future work to study free algebraic systems which can express these operations.

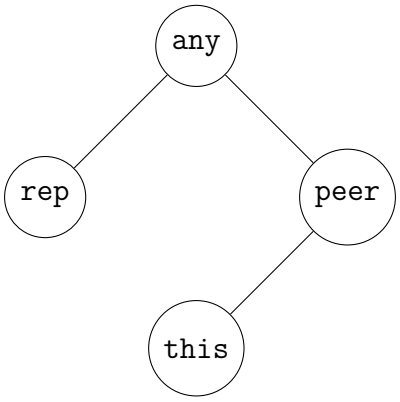

Figure 6.1: Order for Ownership

6.7.3 Languages with Immutability Systems

Finally, some languages have been explicitly designed with immutability in mind.

C++: `const`-qualified methods and values provide limited viewpoint adaptation. Reading a field from a `const`-qualified object returns a `const`-qualified field, and C++ supports function and method dispatching based on the `const`ness of its arguments [87]. Mutability polymorphism is not explicitly supported but can be done with a combination of templates and overloading.

```
struct BoxedInt {
    int v{0};
};
template<typename T> struct HasF<T> {
    T f;
    T& getF() { return f; }
    const T& getF() const { return f; }
}

const HasF<BoxedInt> x;
x.getF() // Calls const qualified getF()
const BoxedInt& OK = x.f; // OK, as x.f is of type const BoxedInt.
BoxedInt& Bad = x.f; // Bad, discards const-qualifier.
```

In this example, a C++ compiler would disallow `Bad` because the type of `x.f` has been adapted to a l-value of `const BoxedInt`. However, viewpoint adaptation does not lift to reference or pointer types in C++. For example, if instead we had a pointer-to-`T` in `HasF`, we could modify that `T` instance through `x`.

```
template<typename T> struct HasF<T> {
    T* f;
}

BoxedInt b{5};
const HasF<BoxedInt> x{&b};
// OK, as x stores a constant pointer to a mutable BoxedInt
BoxedInt* NotGreat = x.f;
NotGreat->v = 10; // Modifies b!
```

C++'s limited viewpoint adaptation gives `x.f` the type `BoxedInt * const`, which is a constant pointer to a mutable `BoxedInt`, not the type `BoxedInt const * const`, which would be a constant pointer to a *constant* `BoxedInt`. This allows the underlying field to be mutated.

D: In contrast to C++, where `const` becomes useless for pointer and reference fields, D supports full reference immutability and viewpoint adaptation with a *transitive* `const` extended to work for pointer and reference types [15]. Again, mutability polymorphism is not directly supported but can be encoded with D's compile-time meta-programming system.

Rust: In Rust, references are either *mutable* or *read-only*, and only one *mutable* reference can exist for any given value. Read-only references are transitive, like they are in System $F_{<:M}$, roDOT, and other reference immutability systems, and unlike C++. Here, in this example, we cannot write to `s3.f` as `s3` is a read-only reference to `s2`, even though `s2.f` has type `&mut String`.

```
struct HasF<T> {
    f: T
}

fn main() {
```

```
    let mut s1 = String::from("hello");
    let s2 = HasF { f: &mut s1 };
    s2.f.push_str("OK");
    let s3 = &s2;
    s3.f.push_str("BAD");
}
```

Unlike other languages, though, the mutability of a reference is an intrinsic property of the reference type itself. Instead of having a type operator `readonly` that, given a reference type `T`, creates a read-only version of that reference type, Rust instead defines `&` and `&mut`, type operators that, given a type `T`, produce the type of a read-only reference to a `T` and the type of a mutable reference to a `T`, respectively. Here, in the following example, `s1` is a `String`, `s2` is a mutable reference to a `s1` – `&mut String`, and `s3` is a read-only reference to `s2` – `& (&mut String)`, where all three of `s1`, `s2`, and `s3` are stored at distinct locations in memory.

```
let s1 = String::from("hello");
let mut s2 = &s1;
let s3 = &s2;
```

As such, in Rust, one cannot create a read-only version of an existing reference type. This makes higher-order functions over references that are polymorphic over mutability, like `inplace_map` from above, inexpressible in Rust. However, if we instead had a `Pair` that owned its elements, we could write the following version of `inplace_map`:

```
struct Pair<T> {
    fst: T,
    snd: T
}

fn inplace_map<T>(p: &mut Pair<T>, f: fn (&T) -> T) {
    p.fst = f(&p.fst);
    p.snd = f(&p.snd);
}
```

Note, though, that in this setting, the elements `p.fst` and `p.snd` are embedded in the pair `p` and owned by it.

6.8 Contracts

Our approach to sealing references (in System $\lambda_{\mathtt{M}}$), and to attaching concrete tags to qualified values (in System $\mathtt{F}_{\mathtt{<:Q}}$, System $\mathtt{F}_{\mathtt{<:B}}$, and derived calculi), is similar to and was inspired by practical programming experience with Racket contracts [86].

`Seal`ing, in particular, can be viewed as attaching a *chaperone* contract which raises an exception whenever the underlying chaperoned value is written to, and attaches a similar chaperone to every value read out of the value. For example, a dynamic reference immutability scheme for Racket vectors could be implemented with the following chaperone contract:

```
(define (chaperone-read vec idx v)
  (seal v))
(define (chaperone-write vec idx v)
  (error 'seal "Tried to write through an immutable reference."))

(define (seal v)
  (cond
    [(vector? v) (chaperone-vector vec chaperone-read chaperone-write)]
    [else v]))
```

Strickland et al. [86] prove that chaperones can be safely erased without changing the behaviour of the underlying program when it reduces to a value. Our results on dynamic safety, Lemmas 5, 6, and 12 can be viewed as an analogue of Strickland et al. [86, Theorem 1] specialized to reference immutability. In this setting, our static immutability safety results show that a well-typed program will never raise an error by writing to a chaperoned vector.

6.9 Refinement Types

Related to type qualifiers is work on *refinement types* [33], another mechanism for specifying subtypes of a given base type. Unlike type qualifiers, which are used to qualify existing types with additional constraints, *refinement types* approach this problem from a different lens and allow end users to specify a set of *refinements* of an existing base type. For example, an instatntiaion of a refinement type can be given as the following refinement of a bitstring to bitstrings representing numbers in least significant digit first standard form (no trailing zeros) can be represented as:

```
datatype bitstr =
    e | z of bitstr | o of bitstr
rectype std = e | stdpos
    and stdpos = o(e) | z(stdpos) | o(stdpos)
```

Here, a `bitstr` is in `std` (standard) form if it is `e`mpty or if it is in `stdpos` (standard position), consisting of `1` followed by an empty bitstring, or a `1` or a `0` followed by a standard position bitstring. This naturally forms a subtyping lattice of refined types: $\bot$ `<: stdpos` and $\bot$ `<: e`, which are both covered by `std` and itself covered by `bitstr`.

There are more mechanisms, of course, for refining types. One mechanism closely connected to Boolean algebras and lattice formulas is that of Lanzinger et al. [50]. There they refine each type with a *property*, a Boolean first-order predicate containing information about that type. For example, an integer 0 might be given the type `int` refined with the predicate `subject == 0`; here, `subject` represents the value `0` that must satisfy the refinement. Now, Boolean first-order predicates in one variable (`subject`) themselves form a Boolean algebra under the usual propositional connectives, but with one catch: the one variable must be drawn from comparable base types when connecting two predicates. Connecting a predicate on `int` with a predicate on `String` would be ill-formed. It would be interesting future work to connect our work on Boolean qualifiers with Lanzinger et al. [50]'s work on refinements. More recently Lehmann et al. [55] describe *generic refinements* which allow even more expressiveness in the formulas used to refine types. Again, it would be interesting to connect their work back to our presentation of free algebra (lattice and Boolean) based type qualifiers.

Chapter 7

Future Work

In this chapter, we perform a quick survey of potential future directions for our research.

7.1 Re-casting existing work

As we observed, earlier in related work, many existing qualifier systems, such as System $\mathtt{CC}_{<:\Box}$ and CallE use the same underlying ideas for subqualification as we do in our thesis but present these ideas in an ad-hoc fashion. It would be interesting future work to re-examine their work with our ideas to see if their calculi (and proofs) could be simplified.

7.2 New applications

It would be interesting future work to apply our ideas around subqualification and qualifier polymorphism to other programming language features that can use the expressiveness of type qualifiers grounded on Boolean algebras. For example, to extend Madsen and van de Pol [62]'s work on Boolean formulas for qualifying nullability to support subqualification. In addition, it would be interesting future work to investigate applying Boolean algebras based qualifiers to topical problems in programming languages, for example: staging/compile time evaluation for efficient programs, type-safe and fluent APIs to aid software library developers, and safe polymorphic laziness to allow developers to safely mix strict and lazy code without running into the pitfalls that plague lazy evaluation.

7.3 Qualifier inference

Many existing qualifier systems are verbose. They often require developers to annotate typing information throughout their code; Java's *checked exception* system is perhaps the canonical example. Anecdotally, this burden has caused some developers to forgo these systems.

Systems with *inference* – i.e. algorithms – which can automatically infer every type and effect in the program offer the best of both worlds: we can get all the benefits of static systems without burdening the developer. Unfortunately, automatic inference algorithms are often shunned by compiler implementers and software developers as their behaviour can be hard to understand. Rust infamously avoids general inference for its reference lifetimes and instead requires software developers to provide annotations when its simple, incomplete heuristics fail to add the appropriate (lifetime) annotations.

Moreover, even when inference algorithms are present in compilers, their correctness is often left formally unspecified and unverified. Many systems are done as heuristics, such as [25, 48, 82], which are by nature not guaranteed to be complete, especially in the face of parametric polymorphism. Others aim to be complete but are left formally unverified. While heuristics may be the best way practically infer types in this context [46], as fully complete algorithms may require non-polynomial time, this area remains underinvestigated. This undesirable gap in the literature has led, in practice, to an extraordinary amount of bugs in real-world compilers, for example, 195 open and almost 1000 historical bugs for the Scala typer.[1] A type system is useless if we cannot reason about its practical implementation in a compiler, as it is what is actually used by developers, in contrast with the declarative systems presented in calculi papers.

Now, there are already existing algorithms for inference for Boolean-algebra based type qualifiers [63]. They however lack proofs of correctness. It would be interesting future work to develop a proof of correctness for this existing Boolean-algebra based type-and-effect inference algorithm and to publish the proof as a reusable artifact in Rocq (or Lean) to benefit other researchers, compiler writers, and language designers.

Some additional useful lines of investigation would be in investigating new techniques for *Boolean unification* for qualifier inference. Today, type and effect inference is the main source of slowdown in compilers; efficient algorithms here will benefit users. Potential ideas include different algebraic representations for efficient unification and a new inference process that treats flexible variables differently from slack variables.

[1] https://github.com/scala/scala3/issues?q=is:issue+label:area:typer

Chapter 8

Conclusion

In this thesis, 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 $\mathtt{F}_{<:}$ can be extended using this structure by constructing such an extension System $\mathtt{F}_{<:\mathtt{Q}}$, 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 then further extended the design recipe provided by System $\mathtt{F}_{<:\mathtt{Q}}$ to a more expressive algebra for qualifiers – boolean algebras – and have have presented a calculus System $\mathtt{F}_{<:\mathtt{B}}$ that extends System $\mathtt{F}_{<:}$ with type qualifiers over Boolean algebras with support for qualifier polymorphism and subqualification. We have shown how System $\mathtt{F}_{<:\mathtt{B}}$ can be used as a design recipe for a type and effect system, System $\mathtt{F}_{<:\mathtt{QA}}$, with effect polymorphism, subeffecting, and polymorphic effect exclusion. We hope that this work advances our understanding of the structure of polymorphism over type qualifiers, and that future systems can reuse the ideas presented in our work instead of reinventing ad-hoc solutions.

References

[1] Scott Aaronson. 2002. Polynomial Hierarchy Collapses: Thousands Feared Tractable. https://scottaaronson.com/writings/phcollapse.pdf 58

[2] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday (Lecture Notes in Computer Science, Vol. 9600)*, Sam Lindley, Conor McBride, Philip W. Trinder, and Donald Sannella (Eds.). Springer, 249–272. https://doi.org/10.1007/978-3-319-30936-1_14 15, 59, 63, 94

[3] Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 3–15. https://doi.org/10.1145/1328438.1328443 44, 65, 88

[4] Franz Baader. 1998. On the complexity of Boolean unification. *Inform. Process. Lett.* 67, 4 (1998). https://doi.org/10.1016/s0020-0190(98)00106-9 97

[5] Yuyan Bao, Guannan Wei, Oliver Bracevac, 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 (2021), 1–32. https://doi.org/10.1145/3485516 92

[6] 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 91

[7] George Boole. 1847. *The mathematical analysis of logic.* 97

[8] Aleksander Boruch-Gruszecki, Jonathan Immanuel Brachthäuser, Edward Lee, Ondrej Lhoták, and Martin Odersky. 2021. Tracking Captured Variables in Types. *CoRR* abs/2105.11896 (2021). arXiv:2105.11896 https://arxiv.org/abs/2105.11896 59, 60, 68

[9] Aleksander Boruch-Gruszecki, Adrien Ghosn, Mathias Payer, and Clément Pit-Claudel. 2024. Gradient: Gradual Compartmentalization via Object Capabilities Tracked in Types. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 1135–1161. https://doi.org/10.1145/3689751 xi, 5, 6, 21

[10] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondrej Lhoták, and Jonathan Immanuel Brachthäuser. 2023. Capturing Types. *ACM Trans. Program. Lang. Syst.* 45, 4 (2023), 21:1–21:52. https://doi.org/10.1145/3618003 8, 18, 19, 20, 59, 60, 63, 64, 68, 71, 92, 98

[11] Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauss. 1989. Unification in Boolean Rings and Abelian Groups. *Journal of Symbolic Computation* 8, 5 (1989). https://doi.org/10.1016/s0747-7171(89)80054-9 97

[12] 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 28

[13] Gilad Bracha. [n. d.]. Pluggable type systems, October 2004. In *OOPSLA Workshop on Revival of Dynamic Languages*. 2

[14] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–30. https://doi.org/10.1145/3527320 59, 60, 63, 68

[15] Walter Bright, Andrei Alexandrescu, and Michael Parker. 2020. Origins of the D programming language. *Proc. ACM Program. Lang.* 4, HOPL (2020), 73:1–73:38. https://doi.org/10.1145/3386323 3, 101

[16] Alex Buckley, Michael Ernst, and Expert Group. 2014. *JSR 308: Annotations on Java Types*. Java Specification Request 308. Java Community Process. https://jcp.org/en/jsr/detail?id=308 Final Release — 04 Mar 2014. 96

[17] Wolfram Buttner and Helmut Simonis. 1987. Embedding boolean expressions into logic programming. *Journal of Symbolic Computation* 4, 2 (1987). https://doi.org/10.1016/s0747-7171(87)80065-2 97

[18] 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

[19] Dave Clarke, James Noble, and Tobias Wrigstad (Eds.). 2013. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification.* Lecture Notes in Computer Science, Vol. 7850. Springer. https://doi.org/10.1007/978-3-642-36946-9

[20] 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, AGERE! 2015, Pittsburgh, PA, USA, October 26, 2015*, Elisa Gonzalez Boix, Philipp Haller, Alessandro Ricci, and Carlos A. Varela (Eds.). ACM, 1–12. https://doi.org/10.1145/2824815.2824816 96

[21] Aaron Craig, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2018. Capabilities: Effects for Free. In *Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11232)*, Jing Sun and Meng Sun (Eds.). Springer, 231–247. https://doi.org/10.1007/978-3-030-02450-5_14 19

[22] Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155. https://doi.org/10.1145/365230.365252 18, 57

[23] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd W. Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 681–690. https://doi.org/10.1145/1985793.1985889 95, 96

[24] Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4609)*, Erik Ernst (Ed.). Springer, 28–53. https://doi.org/10.1007/978-3-540-73589-2_3 7, 13, 24, 99

[25] Werner Dietl, Michael D. Ernst, and Peter Müller. 2011. Tunable Static Inference for Generic Universe Types. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings (Lecture Notes in Computer Science, Vol. 6813)*, Mira Mezini (Ed.). Springer, 333–357. https://doi.org/10.1007/978-3-642-22655-7_16 106

[26] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 287–300. https://doi.org/10.1145/2429069.2429104

[27] Stephen Dolan. 2016. *Algebraic subtyping*. Ph. D. Dissertation. Cambridge University. 3, 32, 97

[28] Vlastimil Dort and Ondrej Lhoták. 2020. Reference Mutability for DOT. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:28. https://doi.org/10.4230/LIPICS.ECOOP.2020.18 10, 15, 94

[29] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward XueJun Wu. 2014. Collaborative Verification of Information Flow for a High-Assurance App Store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, Gail-Joon Ahn, Moti Yung, and Ninghui Li (Eds.). ACM, 1092–1104. https://doi.org/10.1145/2660267.2660343 96

[30] Matthias Felleisen and Daniel P. Friedman. 1987. A Calculus for Assignments in Higher-Order Languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 314–325. https://doi.org/10.1145/41625.41654 49, 74, 87

[31] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, May 1-4, 1999*, Barbara G. Ryder and Benjamin G. Zorn (Eds.). ACM, 192–203. https://doi.org/

10.1145/301618.301665 1, 2, 4, 6, 7, 9, 13, 20, 23, 24, 26, 27, 29, 30, 32, 33, 37, 41, 91, 93, 98

[32] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 1–12. https://doi.org/10.1145/512529.512531 98

[33] Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. https://doi.org/10.1145/113445.113468 103

[34] Nikolaos Galatos. 2023. Decidability of Lattice Equations. https://doi.org/10.1007/s11225-023-10063-4 31

[35] 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 21, 22, 98

[36] Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 4:1–4:79. https://doi.org/10.1145/3450272 98, 99

[37] 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 96

[38] 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 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 21–40. https://doi.org/10.1145/2384616.2384619 5, 7, 10, 15, 16, 17, 93

[39] 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. 97

[40] Christian Haack and Erik Poll. 2009. Type-Based Object Immutability with Flexible Initialization. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5653)*, Sophia Drossopoulou (Ed.). Springer, 520–545. https://doi.org/10.1007/978-3-642-03013-0_24 29, 93

[41] Philipp Haller and Ludvig Axelsson. 2017. Quantifying and Explaining Immutability in Scala. In *Proceedings Tenth Workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2017, Uppsala, Sweden, 29th April 2017 (EPTCS, Vol. 246)*, Vasco T. Vasconcelos and Philipp Haller (Eds.). 21–27. https://doi.org/10.4204/EPTCS.246.5 10

[42] Wei Huang, Werner Dietl, Ana L. Milanova, and Michael D. Ernst. 2012. Inference and Checking of Object Ownership. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 181–206. https://doi.org/10.1007/978-3-642-31057-7_9 99

[43] Wei Huang, Ana L. Milanova, Werner Dietl, and Michael D. Ernst. 2012. Reim & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 879–896. https://doi.org/10.1145/2384616.2384680 9, 10, 17, 45, 94

[44] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. https://doi.org/10.1145/503502.503505

[45] Peter Jipsen. 2001. A Gentzen system and decidability for residuated lattices. *Preprint* (2001). https://www1.chapman.edu/~jipsen/reslat/gentzenrl.pdf 31

[46] Nahid Juma, Werner Dietl, and Mahesh Tripunitara. 2020. A computational complexity analysis of tunable type inference for Generic Universe Types. *Theor. Comput. Sci.* 814 (2020), 189–209. https://doi.org/10.1016/J.TCS.2020.01.035 106

[47] Paul A. Karger and A. J. Herbert. 1984. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 29 - May 2, 1984*. IEEE Computer Society, 2–12. https://doi.org/10.1109/SP.1984.10001 41

[48] Martin Kellogg, Daniel Daskiewicz, Loi Ngo Duc Nguyen, Muyeed Ahmed, and Michael D. Ernst. 2023. Pluggable Type Inference for Free. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*. IEEE, 1542–1554. https://doi.org/10.1109/ASE56229.2023.00186 106

[49] Brian W. Kernighan and Dennis Ritchie. 1988. *The C Programming Language, Second Edition*. Prentice-Hall. https://en.wikipedia.org/wiki/The_C_Programming_Language 91

[50] Florian Lanzinger, Alexander Weigl, Mattias Ulbrich, and Werner Dietl. 2021. Scalability and precision by combining expressive type systems and deductive verification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–29. https://doi.org/10.1145/3485520 104

[51] Edward Lee and Ondrej Lhoták. 2023. Simple Reference Immutability for System F. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 857–881. https://doi.org/10.1145/3622828 iv, 45

[52] Edward Lee, Kavin Satheeskumar, and Ondrej Lhoták. 2023. Dependency-Free Capture Tracking. In *Proceedings of the 25th ACM International Workshop on Formal Techniques for Java-like Programs, FTfJP 2023, Seattle, WA, USA, 18 July 2023*, Aaron Tomb (Ed.). ACM, 39–43. https://doi.org/10.1145/3605156.3606454 iv, 45

[53] Edward Lee, Jonathan Lindegaard Starup, Ondřej Lhoták, and Magnus Madsen. 2025. Qualified Types with Boolean Algebras. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 318 (Oct. 2025), 27 pages. https://doi.org/10.1145/3763096 iv

[54] Edward Lee, Yaoyu Zhao, Ondrej Lhoták, James You, Kavin Satheeskumar, and Jonathan Immanuel Brachthäuser. 2024. Qualifying System $F_{<:}$: Some Terms and Conditions May Apply. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 583–612. https://doi.org/10.1145/3649832 iv

[55] Nico Lehmann, Cole Kurashige, Nikhil Akiti, Niroop Krishnakumar, and Ranjit Jhala. 2025. Generic Refinement Types. *Proc. ACM Program. Lang.* 9, POPL (2025), 1446–1474. https://doi.org/10.1145/3704885 104

[56] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8 97

[57] 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/ 98

[58] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidizing OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 8, ICFP (2024), 485–514. https://doi.org/10.1145/3674642 96

[59] John M. Lucassen and David K. Gifford. 1988. Polymorphic Effect Systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, Jeanne Ferrante and Peter Mager (Eds.). ACM Press, 47–57. https://doi.org/10.1145/73560.73564 97

[60] 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 (2023), 448–475. https://doi.org/10.1145/360784678, 96

[61] Leopold Löwenheim. 1908. *Über das Auflösungsproblem im logischen Klassenkalkul.* 97

[62] Magnus Madsen and Jaco van de Pol. 2021. Relational nullable types with Boolean unification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–28. https://doi.org/10.1145/3485487 92, 93, 96, 98, 105

[63] Magnus Madsen, Jaco van de Pol, and Troels Henriksen. 2023. Fast and Efficient Boolean Unification for Hindley-Milner-Style Type and Effect Systems. *Proc. ACM*

Program. Lang. 7, OOPSLA2 (2023), 516–543. https://doi.org/10.1145/3622816 106

[64] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd D. Millstein, Chris Andreae, and James Noble. 2010. JavaCOP: Declarative pluggable types for Java. *ACM Trans. Program. Lang. Syst.* 32, 2 (2010), 4:1–4:37. https://doi.org/10.1145/1667048.1667049 3

[65] 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 28, 97

[66] Fabian Muehlboeck and Ross Tate. 2018. Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 112:1–112:29. https://doi.org/10.1145/3276482 85

[67] 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 7, 31, 37, 80

[68] Bob Nystrom. 2015. What Color is Your Function? https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/ 47, 72

[69] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. 2018. Simplicitly: foundations and applications of implicit function types. *Proc. ACM Program. Lang.* 2, POPL (2018), 42:1–42:29. https://doi.org/10.1145/3158130 19

[70] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *Theory Pract. Object Syst.* 5, 1 (1999), 35–55. 91

[71] Leo Osvald, Grégory M. 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, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 234–251. https://doi.org/10.1145/2983990.2984009 94

[72] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *Proceedings of the*

ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, Barbara G. Ryder and Andreas Zeller (Eds.). ACM, 201–212. https://doi.org/10.1145/1390630.1390656 3, 5, 46, 96, 98

[73] Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP (2020), 124:1–124:28. https://doi.org/10.1145/3409006 97

[74] Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: principal type inference in a Boolean algebra of structural types. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 449–478. https://doi.org/10.1145/3563304 97

[75] Tomas Petricek, Dominic A. 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, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 123–135. https://doi.org/10.1145/2628136.2628160 1

[76] 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 48

[77] Marianna Rapoport and Ondrej Lhoták. 2019. A path to DOT: formalizing fully path-dependent types. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 145:1–145:29. https://doi.org/10.1145/3360571 59, 63

[78] John C. Reynolds. 1997. *Design of the Programming Language Forsythe*. Birkhäuser Boston, Boston, MA, 173–233. https://doi.org/10.1007/978-1-4612-4118-8_9

[79] Sergiu Rudeanu. 1974. *Boolean functions and equations*. 97

[80] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight Polymorphic Effects. In *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7313)*, James Noble (Ed.). Springer, 258–282. https://doi.org/10.1007/978-3-642-31057-7_13 97

[81] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: approximate data types for safe and general low-power computation. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 164–174. https://doi.org/10.1145/1993498.1993518 3

[82] Narges Shadab, Pritam M. Gharat, Shrey Tiwari, Michael D. Ernst, Martin Kellogg, Shuvendu K. Lahiri, Akash Lal, and Manu Sridharan. 2023. Inference of Resource Management Specifications. *Proc. ACM Program. Lang.* 7, OOPSLA2 (2023), 1705–1728. https://doi.org/10.1145/3622858 106

[83] 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. 31

[84] Max Slater. 2023. Oxidizing OCaml: Locality. https://blog.janestreet.com/oxidizing-ocaml-locality/

[85] Max Slater. 2023. Oxidizing OCaml: Rust-Style Ownership. https://blog.janestreet.com/oxidizing-ocaml-ownership/ 3, 96

[86] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. 2012. Chaperones and impersonators: run-time support for reasonable interposition. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 943–962. https://doi.org/10.1145/2384616.2384685 103

[87] Bjarne Stroustrup. 2007. *The C++ programming language - special edition (3. ed.)*. Addison-Wesley. 3, 100

[88] EISOP team. 2024. EISOP Checker Framework. https://github.com/eisop/checker-framework 5

[89] 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 3, 6, 8, 9, 10, 16, 17, 99

[90] Guannan Wei, Oliver Bracevac, 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 (2024), 393–424. https://doi.org/10.1145/3632856 5, 6, 46, 92, 93, 98

[91] Philip M. Whitman. 1941. Free Lattices. *Annals of Mathematics* 42, 1 (1941), 325–330. http://www.jstor.org/stable/1969001 7, 8, 30, 31, 91

[92] 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 3, 95

[93] Anxhelo Xhebraj, Oliver Bracevac, 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, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29. https://doi.org/10.4230/LIPICS.ECOOP.2022.15 94

[94] Tongtong Xiang, Jeff Y. Luo, and Werner Dietl. 2020. Precise inference of expressive units of measurement types. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 142:1–142:28. https://doi.org/10.1145/3428210 96

[95] Yaoyu Zhao. 2023. *Adding Reference Immutability to Scala.* Master's thesis. http://hdl.handle.net/10012/19601 94

[96] 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 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 75–84. https://doi.org/10.1145/1287624.1287637 10, 17, 45, 48