Simple Reference Immutability for System $F_{<:}$

EDWARD LEE, University of Waterloo, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada

Reference immutability is a type based technique for taming mutation that has long been studied in the context of object-oriented languages, like Java. Recently, though, languages like Scala have blurred the lines between functional programming languages and object oriented programming languages. We explore how reference immutability interacts with features commonly found in these hybrid languages, in particular with higher-order functions – polymorphism – and subtyping. We construct a calculus System $F_{<:M}$ which encodes a reference immutability system as a simple extension of System $F_{<:}$ and prove that it satisfies the standard soundness and immutability safety properties.

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

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

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

252

1 INTRODUCTION

Code written in a pure, functional language is *referentially transparent* – it has no side effects and hence can be run multiple times to produce the same result. Reasoning about referentially transparent code is easier for both humans and computers. However, purely functional code can be hard to write and inefficient, so many functional languages contain impure language features.

One important side effect that is difficult to reason about is *mutation* of state. 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.

One method of taming mutation is *reference immutability* [Huang et al. 2012; Tschantz and Ernst 2005]. 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.

Mutable and immutable references can coexist for the same value, so an immutable reference does not 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.

Reference immutability has long been studied in existing object-oriented programming languages such as Java [Huang et al. 2012; Tschantz and Ernst 2005; Zibin et al. 2007] and C# [Gordon et al. 2012]. Recently work has been done to study reference immutability in the context of functional

Authors' addresses: Edward Lee, e45lee@uwaterloo.ca, School of Computer Science, University of Waterloo, 200 University Ave W., Waterloo, ON, N2L 3G1, Canada; Ondřej Lhoták, olhotak@uwaterloo.ca, School of Computer Science, University of Waterloo, 200 University Ave W., Waterloo, ON, N2L 3G1, Canada.

2475-1421/2023/10-ART252
https://doi.org/10.1145/3622828

languages with impure fragments, in particular by Dort and Lhoták [2020], as many programs in impure functional languages tend not to mutate their data structures anyways [Haller and Axelsson 2017]. However, their work builds on a complex calculus, DOT, Amin et al. [2016], and their work is itself complex as well.

A simple system that formally enforces specified patterns of immutability in a functional base calculus would help programmers and compilers better reason about immutability in such programs. However, such a system can be hard to come by. Parametric polymorphism, a key feature in any language, is especially so in functional languages, and as we see later in Section 4.1 the interaction between immutability qualifiers and polymorphism raises issues that are not so easy to solve. What should the type `@readonly X` mean if the variable `X` is instantiated with `@mutable String`?

Our contribution to this area is a *simple* and *sound* treatment of reference immutability in System $\mathsf{F}_{<:}$ [Cardelli et al. 1991]. Specifically, we formulate a simple extension System $\mathsf{F}_{<:\mathsf{M}}$ of System $\mathsf{F}_{<:}$ with the following properties:

- **Immutability safety:** When dealing with reference immutability, one important property to show is *immutability safety*: showing that when a reference is given a read-only type, then the underlying value is not modified through that reference. In System $\mathsf{F}_{<:\mathsf{M}}$ we introduce a dynamic form of immutability, a term-level `seal` construct, which makes precise the runtime guarantees that we expect from a reference that is statically designated as immutable by the type system. We do this by formalizing System λ_{M}, an untyped calculus with references and seals. Dynamic seals are transitive in that they seal any new references that are read from a field of an object through a sealed reference.
- **System $\mathsf{F}_{<:}$-style polymorphism:** System $\mathsf{F}_{<:\mathsf{M}}$ preserves the same bounded-quantification structure of System $\mathsf{F}_{<:}$. At the same time, it allows type variables to be further modified by immutability modifiers.
- **Immutable types are types:** To allow for System $\mathsf{F}_{<:}$-style polymorphism, we need to treat immutable types as types themselves. To do so, instead of type qualifiers, we introduce a type *operator* `readonly` that can be freely applied to existing types (including type variables). The `readonly` operator turns a type into an immutable version of the same type. While this complicates the definition of subtyping and proofs of canonical forms lemmas, we resolve these issues by reducing types to a normal form.

Our hope is to enable reference immutability systems in functional languages by giving simple, sound foundations in System $\mathsf{F}_{<:}$, a calculus that underpins many practical functional programming languages.

The rest of this paper is organized as follows. In Section 2 we give an overview of reference immutability. In Section 3 we introduce an un-typed core calculus, System λ_{M}, to describe `sealing` and how it relates to reference immutability safety at run time. In Section 4 we present System $\mathsf{F}_{<:\mathsf{M}}$, which enriches System λ_{M} with types, and show that it satisfies the standard soundness theorems. In Section 5 we use the soundness results from System $\mathsf{F}_{<:\mathsf{M}}$ and the dynamic safety results from System λ_{M} to show that our desired immutability safety properties hold in System $\mathsf{F}_{<:\mathsf{M}}$. We survey related and possible future work in Section 7 and we conclude in Section 8.

Our development is mechanized in the Coq artifact that accompanies this paper.

2 REFERENCE IMMUTABILITY

Reference immutability at its core is concerned with two key ideas:

- **Immutable references:** References to values can be made immutable, 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* [Dietl et al. 2007] to be a read-only reference, even if the pair contains references that are otherwise mutable.

For example, consider the following snippet of Scala-like code that deals with polymorphic 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 }
```

A reference immutability system would deem the function good to be well-typed because it mutates the pair through a mutable reference `x`. However, it would disallow `bad1` because it mutates the pair through a read-only reference `y`. Moreover, it 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`.

Now, 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 [Haller and Axelsson 2017], 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 special tricks like lazyness, this sort of structure cannot be constructed, let alone copied, without mutation.

```
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.

```
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 -> @readonly v) // update
            binding
    }
    interpret (env + (x -> @readonly v), f)
```

```
}
```

3 DYNAMIC IMMUTABILITY SAFETY

Now, 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?

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 λ_{M} calculus with a notion of *sealed* references.

System λ_{M} is adapted from the CS-machine of Felleisen and Friedman [1987] 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 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* – records – 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:

$$
\begin{array}{lllll}
s, t & ::= & & \textbf{Terms} \\
& | & \lambda x.t & \text{term abstraction} \\
& | & x & \text{term variable} \\
& | & s(t) & \text{application} \\
& | & \{f_1 : s_1, f_2 : s_2, \ldots\} & \text{records} \\
& | & s.f & \text{field read} \\
& | & s.f = t & \text{field write} \\
& | & \boxed{\texttt{seal}\ s} & \text{sealing}
\end{array}
\qquad
\begin{array}{llll}
& & l & \textbf{Location} \\
s, t & ::= & & \textbf{Runtime Terms} \\
& | & \{f_i : l_i, \ldots\} & \text{runtime record} \\
v & ::= & & \textbf{Runtime Values} \\
& | & \lambda x.t & \\
& | & \{f_i : l_i\} & \\
& | & \boxed{\texttt{seal}\ \{f_i : l_i, \ldots\}} &
\end{array}
$$

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

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

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

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

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

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

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

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

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

Fig. 1. The syntax and semantics of λ_m.

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

So a term like:

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

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

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

For example:

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

Sealed references and *dynamic viewpoint adaptation* allow for a succinct guarantee of *dynamic transitive immutability safety* – that 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 3.1. Let s and t be two terms. We say $s \le t$ if t can be obtained from s by repeatedly replacing sub-terms s' of s with sealed subterms $\texttt{seal}\ s'$.

This implies a similar definition for stores:

Definition 3.2. Let σ and σ' be two stores. We say $\sigma \le \sigma'$ if and only if they have the same locations and for every location $l \in \sigma$, we have $\sigma(l) \le \sigma'(l)$.

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 3.3. Let s be a term. Then $|s|$ is the number of seals in s.

Lemma 3.4. *Let v be a value, σ_v be a store, t be a term such that $v \le t$, and σ_t be a store such that $\sigma_v \le \sigma_t$.*
If $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$ then $v \le t'$, $\sigma_v \le \sigma'_t$, and $|t'| < |t|$.

The next lemma is an analogue of Lemma 3.4 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 $s \le t'$.

Lemma 3.5. *Let s, t be terms such that $s \le t$ and let σ_s, σ_t be stores such that $\sigma_s \le \sigma_t$. 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 $s \le t'$, $\sigma_s \le \sigma'_t$, and $|t'| < |t|$, or*
(2) *$s' \le t'$ and $\sigma'_s \le \sigma'_t$.*

Together, Lemmas 3.4 and 3.5 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 3.6. *Let s, t be terms such that $s \le t$ and let σ_s, σ_t be stores such that $\sigma_s \le \sigma_t$. If $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$ then:*
(1) *Either $s \le t'$, $\sigma_s \le \sigma'_t$, and $|t'| < |t|$, or*
(2) *There exists s' and σ'_s such that $\langle s, \sigma_s \rangle \longrightarrow \langle s', \sigma'_s \rangle$, $s' \le t'$ and $\sigma'_s \le \sigma'_t$.*

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

Lemma 3.7. *If s is a term and v is a value such that $s \le v$, then s is also a value.*

Hence:

Lemma 3.8. *Suppose s and t are terms such that $s \le t$. If $\langle t, \sigma_t \rangle \longrightarrow^* \langle v_t, \sigma'_t \rangle$ for some value v_t, then for any $\sigma_s \le \sigma_t$ we have $\langle s, \sigma_s \rangle \longrightarrow^* \langle v_s, \sigma'_s \rangle$ such that $v'_s \le v'_s$ and $\sigma'_s \le \sigma'_t$.*

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 3.9. *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 `seals` 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 TYPING AND STATIC SAFETY

System λ_{M} provides a *dynamic guarantee* that a given program will never modify its sealed references, but it does not provide any static guarantees about the dynamic behavior of a given program. To do that, we need a type system for System λ_{M} that will reject programs like `access(seal Pair(3,5)).first = 10`, which we know will crash.

To ensure that well-typed programs do not get stuck, a type system for System λ_{M} needs a static analogue of `sealing` – a way to turn an existing type into a *read-only type*. Read-only types denote references that are *immutable* and that (transitively) *adapt* any other references read through them to be *immutable* as well.

Issues arise, however, when we introduce 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 annotations (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`?

4.1 Polymorphism

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. This raises important questions – for example, is this operation even well founded? This depends on what X ranges over.

X ranges over an unqualified type: If type variables range over types that have not been qualified by @readonly, then this operation is clearly well founded – it is simply qualifying the unqualified type that X will eventually be substituted by with the @readonly qualifier. This approach has been used by ReIm for Java and for an immutability system for C# – [Gordon et al. 2012; Huang et al. 2012].

However, this raises the problem of polymorphism over immutability qualifiers as well – for example, a Pair should be able to store both immutable and mutable object references. The only natural solution is to then introduce a *mutablity* qualifier binder to allow for polymorphism over immutability qualifiers, as thus:

```
case class Pair[M, X](var first: M X, var second: M X)
def inplace_map[M, X](pair: Pair[M, X], f: @readonly X => M X): Unit = {
    pair.first = f(pair.first);
    pair.second = f(pair.second);
}
```

Mutability qualifier binders have been used previously, most notably by [Gordon et al. 2012]. For one, updating the binding structure of a language is not an easy task – ReIm notably *omits* this sort of parametric mutability polymorphism [Huang et al. 2012]. However, this sort of solution has its downsides; in particular, existing higher-order functions need to be updated with immutability annotations or variables, as type variables no longer stand for a full type. For example, an existing definition of List map which appears as thus originally:

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

needs to be updated to read as the following instead:

```
def map[M, X](l: List[M X], f: M X => M X): List[M X]
```

Instead, we would like to have X range over fully qualified types as well, but as we will see that poses some issues as well.

X ranges over fully-qualified types: If type variables can range over types that have already been qualified by @readonly, then we can avoid introducing mutability binders in the definitions for Pair, inplace_map, and map above. A Pair can be polymorphic over its contents X without caring about the underlying mutability of X. However, this raises the question – how do we interpret repeated applications of the @readonly qualifier? For example, what if we applied inplace_map on a Pair[@readonly Pair[Int]]? Then inplace_map would expect a function f with type @readonly (@readonly Pair[Int]) => @readonly Pair[Int]. While our intuition would tell us that @readonly (@readonly Pair[Int]) is really just a @readonly Pair[Int], discharging this equivalence in a proof is not so easy.

One response is to explicitly prevent type variables from being further qualified. Calculi which take this approach include [Tschantz and Ernst 2005; Zibin et al. 2007]. However, this restriction prevents this version of inplace_map from being expressed. How can we address this?

Our approach, which we explain below, is to treat @readonly as a type operator that works over all types. Following the intuition that sealing removes setters from references, @readonly should be a type operator which removes setters from types. While this does cause complications, we

s, t	$::=$		**Terms**
	$\mid$	$\lambda x.t$	term abstraction
	$\mid$	$\Lambda(X <: S).t$	type abstraction
	$\mid$	x	term variable
	$\mid$	$s(t)$	application
	$\mid$	$s[T]$	type application
	$\mid$	$\{f_1 : s_1, f_2 : s_2, \ldots\}$	records
	$\mid$	$s.x$	field read
	$\mid$	$s.x = t$	field write
	$\mid$	$\texttt{seal}\ s$	sealing

S, T	$::=$		**Types**
	$\mid$	X	type variable
	$\mid$	$S \to T$	function type
	$\mid$	$\forall(X<:S).T$	for-all type
	$\mid$	$S \wedge T$	intersection type
	$\mid$	$\{f : T\}$	record type
	$\mid$	$\texttt{readonly}\ T$	readonly type
Γ	$::=$		**Environment**
	$\mid$	$\cdot$	empty
	$\mid$	$\Gamma,\ x : T$	term binding
	$\mid$	$\Gamma,\ X <: T$	type binding

		l	**Location**
s, t	$::=$		**Runtime Terms**
	$\mid$	$\{x_1 : l_1, x_2 : l_2, \ldots\}$	runtime record
v	$::=$		**Runtime Values**
	$\mid$	$\lambda x.t$	
	$\mid$	$\Lambda(X <: S).t$	
	$\mid$	$\{f_1 : l_1, f_2 : l_2, \ldots\}$	
	$\mid$	$\texttt{seal}\ \{f_1, l_1, f_2 : l_2, \ldots\}$	

σ	$::=$		**Store**
	$\mid$	$\cdot$	empty
	$\mid$	$\sigma,\ l : v$	cell l with value v
Σ	$::=$		**Store Environment**
	$\mid$	$\cdot$	empty
	$\mid$	$\sigma,\ l : T$	cell binding

Fig. 2. The syntax of System $\mathsf{F}_{<:\mathsf{M}}$.

show below how types like `@readonly @readonly Pair[Int]` can be dealt with, using *subtyping* and *type normalization.*

4.2 System $\mathsf{F}_{<:\mathsf{M}}$

To address these issues, we introduce System $\mathsf{F}_{<:\mathsf{M}}$, which adds a type system in the style of System $\mathsf{F}_{<:}$ to System λ_{M}. The syntax of System $\mathsf{F}_{<:\mathsf{M}}$ is given in Figure 2; changes from System $\mathsf{F}_{<:}$ are noted in grey.

System $\mathsf{F}_{<:\mathsf{M}}$ is a straightforward extension of System $\mathsf{F}_{<:}$ with collections of mutable references – namely, records – and with two new extensions: *read-only* types and *sealed* references. To be close to existing functional languages with subtyping and records, records in System $\mathsf{F}_{<:\mathsf{M}}$ are modelled as intersections of single-element record types, to support record subsumption, as in [Amin et al. 2016] and [Reynolds 1997]. See Figures 4 and 5 for full subtyping and typing rules respectively.

Read-only types: The `readonly` type operator transforms an existing type to a read-only version of itself. Unlike the read-only mutability qualifier in Javari and ReIm, which is paired with a type to form a pair of a qualifier and a type, a read-only type in System $\mathsf{F}_{<:\mathsf{M}}$ is itself a type. The `readonly` operator can be seen as the static counterpart of sealing or of deleting setter methods from an object-oriented class type.

Normal Forms

S, T	$::=$	**Types in normal form**	
	$\mid$	$\bigwedge_i (R_i)$	Intersection of components
R	$::=$	**Normal form type components**	
	$\mid$	$\top$	Top type
	$\mid$	$S \to T$	Normal function type
	$\mid$	$\forall(X <: S).T$	Normal for-all type
	$\mid$	$\{f : S\}$	Normal record type
	$\mid$	X	Type variable
	$\mid$	$\texttt{readonly}\ \{f : S\}$	Read-only normal record type
	$\mid$	$\texttt{readonly}\ X$	Read-only type variable

Fig. 3. Normal forms for System $\mathsf{F}_{<:\mathsf{M}}$.

Subtyping

$$\boxed{\Gamma \vdash S <: T}$$

$$\Gamma \vdash T <: T \quad \text{(REFL)}$$

$$\frac{\Gamma \vdash R <: S \qquad \Gamma \vdash S <: T}{\Gamma \vdash R <: T} \quad \text{(TRANS)}$$

$$\frac{X <: T \in \Gamma}{\Gamma \vdash X <: T} \quad \text{(TVAR)}$$

$$\Gamma \vdash U <: \top \quad \text{(TOP)}$$

$$\frac{\Gamma \vdash T_1 <: S_1 \qquad \Gamma \vdash S_2 <: T_2}{\Gamma \vdash S_1 \to S_2 <: T_1 \to T_2} \quad \text{(ARROW)}$$

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

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: S}{\Gamma \vdash \{x : S\} <: \{x : T\}} \quad \text{(RECORD)}$$

$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash \texttt{readonly}\ \{x : S\} <: \texttt{readonly}\ \{x : T\}} \quad \text{(READONLY-RECORD)}$$

$$\Gamma \vdash S \wedge T <: S \quad \text{(INTER-LEFT)}$$

$$\Gamma \vdash S \wedge T <: T \quad \text{(INTER-RIGHT)}$$

$$\frac{\Gamma \vdash S <: T_1 \qquad \Gamma \vdash S <: T_2}{\Gamma \vdash S <: T_1 \wedge T_2} \quad \text{(INTER)}$$

$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash \texttt{readonly}\ S <: \texttt{readonly}\ T} \quad \text{(READONLY)}$$

$$\frac{\Gamma \vdash S <: T}{\Gamma \vdash S <: \texttt{readonly}\ T} \quad \text{(MUTABLE)}$$

$$\frac{\Gamma \vdash nf(S) <: nf(T)}{\Gamma \vdash S <: T} \quad \text{(DENORMALIZE)}$$

Fig. 4. Subtyping rules of System $\mathsf{F}_{<:\mathsf{M}}$.

Typing and Runtime Typing $\boxed{\Gamma \mid \Sigma \vdash t : T \text{ and } \Gamma \mid \Sigma \vdash \sigma}$

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

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \mid \Sigma \vdash \lambda x.t : S \to T} \quad (\text{ABS})$$

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

$$\frac{\Gamma \mid \Sigma \vdash t : S \to T \qquad \Gamma \mid \Sigma \vdash s : S}{\Gamma \mid \Sigma \vdash t(s) : T} \quad (\text{APP})$$

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

$$\frac{\Gamma \mid \Sigma \vdash t_i : T_i}{\Gamma \mid \Sigma \vdash \{x_i : t_i \ldots\} : \bigwedge_i \{x_i : T_i\}} \quad (\text{RECORD-INTRO})$$

$$\frac{\Gamma \mid \Sigma \vdash t : \{x : T\}}{\Gamma \mid \Sigma \vdash t.x : T} \quad (\text{RECORD-ELIM})$$

$$\frac{\Gamma \mid \Sigma \vdash s : \{x : T\} \qquad \Gamma \vdash t : T}{\Gamma \mid \Sigma \vdash s.x = t : T} \quad (\text{RECORD-UPDATE})$$

$$\frac{\Gamma \mid \Sigma \vdash s : S \qquad \Gamma \vdash S <: T}{\Gamma \mid \Sigma \vdash s : T} \quad (\text{SUB})$$

$$\frac{\Gamma \mid \Sigma \vdash s : S}{\Gamma \mid \Sigma \vdash \texttt{seal}\ s : \texttt{readonly}\ S} \quad (\text{SEAL})$$

$$\frac{\Gamma \mid \Sigma \vdash s : \texttt{readonly}\ \{x : S\}}{\Gamma \mid \Sigma \vdash s.x : \texttt{readonly}\ S} \quad (\text{READONLY-RECORD-ELIM})$$

$$\frac{l_i : T_i \in \Sigma}{\Gamma \mid \Sigma \vdash \{x_i : l_i\} : \bigwedge_i \{x_i : T_i\}} \quad (\text{RUNTIME-RECORD})$$

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

Fig. 5. Typing rules for System $\mathsf{F}_{<:\mathsf{M}}$

Any type T is naturally a subtype of its readonly counterpart `readonly T`, which motivates the choice of System $\mathsf{F}_{<:}$ as a base calculus. This subtyping relationship is reflected in the subtyping rule (MUTABLE). The (SEAL) typing rule gives a read-only type to sealed references.

Static viewpoint adaptation: The (READONLY-RECORD-ELIM) rule is a static counterpart of the (SEALED-FIELD) reduction rule. Given a reference s to a record with read-only type, it gives a read-only type to the result of a read $s.x$ of a field x from that reference. If S is the type of field x in the record type given to s, the rule viewpoint-adapts the type, giving $s.x$ the type `readonly` S.

4.2.1 Normal Forms for Types. In System $\mathsf{F}_{<:\mathsf{M}}$, `readonly` is a type operator that can be applied to any type, which enables us to express types such as `readonly` X, where X is some type variable of unknown mutability. However, if X is itself instantiated with some readonly type `readonly` T, the type `readonly` X becomes `readonly readonly` T, with two occurrences of the type operator. Intuitively, such a type should have the same meaning as `readonly` T.

Additionally, certain types should be equivalent under subtyping. For example, for both backwards compatibility and simplicity, arrow $S \to T$ and for-all types $\forall(X <: S).T$ should be equivalent under

Normalization $\boxed{nf(T) \text{ and } merge(T)}$

$$
\begin{array}{lllll}
nf(T) & ::= & & & \textbf{Normalization} \\
& \mid \top & \texttt{=>} & \top \\
& \mid X & \texttt{=>} & X \\
& \mid S \to T & \texttt{=>} & nf(S) \to nf(T) \\
& \mid \forall(X <: S).T & \texttt{=>} & \forall(X <: nf(S)).nf(T) \\
& \mid S \wedge T & \texttt{=>} & nf(S) \wedge nf(T) \\
& \mid \{f : T\} & \texttt{=>} & \{f : nf(T)\} \\
& \mid \texttt{readonly}\ T & \texttt{=>} & merge(T)
\end{array}
$$

$$
\begin{array}{llll}
merge(T) & ::= & & \textbf{Merging} \\
& \mid X & \texttt{=>} & \texttt{readonly}\ X \\
& \mid \{f : T\} & \texttt{=>} & \texttt{readonly}\ \{f : T\} \\
& \mid S \wedge T & \texttt{=>} & merge(S) \wedge merge(T) \\
& \mid \_ & \texttt{=>} & T
\end{array}
$$

Fig. 6. Normalizing Types for System $F_{<:\mathtt{M}}$.

subtyping to their read-only forms `readonly` $(S \to T)$ and `readonly` $(\forall(X <: S).T)$, respectively, as well.

Having multiple representations for the same type, even infinitely many, complicates reasoning about the meanings of types and proofs of soundness. Therefore, we define a canonical representation for types as follows:

Definition 4.1. A type T is in normal form if:

(1) T is the top type $\top$.
(2) T is a function type $S_1 \to S_2$, where S_1 and S_2 are in normal form.
(3) T is an abstraction type $\forall(X <: S_1).S_2$, where S_1 and S_2 are in normal form.
(4) T is an intersection type $S_1 \wedge S_2$, where S_1 and S_2 are in normal form.
(5) T is a record type $\{x : S\}$, where S is in normal form.
(6) T is a read-only record type `readonly` $\{x : S\}$, where S is in normal form.
(7) Type variables X and read-only type variables `readonly` X are in normal form.

A type in normal form is simple – it is an intersection of function, abstraction, and record types, each possibly modified by a single `readonly` operator. For example, $\{x : X\} \wedge$ `readonly` $\{y : Y\}$ is in normal form. The type `readonly` $(\{x : X\} \wedge \{y : Y\})$ is not. A grammar for types in normal form can be found in Figure 3.

This allows us to reason about both the shape of the underlying value being typed, and whether or not it has been modified by a `readonly` operator. Naturally we need a theorem which states that every type has a normal form and a function nf to compute that normal form. Such a function nf is shown in Figure 6. Normalization both computes a normal form and is idempotent – a type in normal form normalizes to itself.

Lemma 4.2. *For any type T, $nf(T)$ is in normal form. Moreover, if T is in normal form, $nf(T) = T$.*

Moreover, types are equivalent to their normalized forms under the subtyping relationship.

Lemma 4.3. *$\Gamma \vdash nf(T) <: T$ and $\Gamma \vdash T <: nf(T)$.*

Proof. For one direction, note that $nf(nf(T)) = nf(T)$, and hence $nf(nf(T)) <: nf(T)$. Applying (denormalize) allows us to show that $nf(T) <: T$, as desired. The other case follows by a symmetric argument. □

Not only does this allow us to simplify types to a normal form, this also allows us to state and prove canonical form lemmas and inversion lemmas, necessary for preservation and progress: Theorems 4.9 and 4.11. Below we give examples for record types. Similar lemmas exist and are mechanized for function types and type-abstraction types as well.

Lemma 4.4 (Inversion of Record Subtyping). *If S is a subtype of $\{f : T'\}$, and S is in normal form, then at least one of its components is a type variable X or a record type $\{f : S'\}$, where $\Gamma \vdash T' <: S' <: T'$.*

Lemma 4.5 (Canonical Forms for Records). *If v is a value and $\varnothing \mid \Sigma \vdash v : \{f : T\}$, then v is a record and f is a field of v that maps to some location l.*

Lemma 4.6 (Inversion of Read-Only Record Subtyping). *If S is a subtype of $\texttt{readonly}\ \{f : T'\}$, and S is in normal form, then at least one of its components is a type variable X, read-only type variable $\texttt{readonly}\ X$, a record type $\{f : S'\}$ where $\Gamma \vdash T' <: S' <: T'$, or a read-only record type $\texttt{readonly}\ \{f : S'\}$ where $\Gamma \vdash T' <: S' <: T'$.*

Lemma 4.7 (Canonical Forms for Read-Only Records). *If v is a value and $\varnothing \mid \Sigma \vdash v : \texttt{readonly}\ \{f : T\}$, then v is a record or a sealed record and f is a field of v that maps to some location l.*

Note that normalization is necessary to state the inversion lemmas for read-only records, as $\texttt{readonly}\ \{f : T'\}$, $\texttt{readonly readonly}\ \{f : T'\}$, etc, give an infinite series of syntactically inequivalent but semantically equivalent types describing the same object – a read-only record where field f has type T'.

4.2.2 *Operational Safety.* Operationally, we give small-step reduction semantics coupled with a store to System $\mathsf{F}_{<:\mathsf{M}}$ in Figure 7.

Again, these rules are a straightforward extension of System $\mathsf{F}_{<:}$ with mutable boxes and records, with additional rules for reducing sealed records. To prove progress and preservation theorems, we additionally need to ensure that the store σ itself is well typed in the context of some store typing environment Σ – see rule (store).

The crux of preservation for System $\mathsf{F}_{<:\mathsf{M}}$ is to show that `sealed` records are never given a non-read-only type, so that the typing rule for reading from a mutable record – (record-elim) – cannot be applied to `sealed` record values.

Lemma 4.8. *Suppose $\Gamma \mid \Sigma \vdash \texttt{seal}\ r : T$ for some record r. If T is in normal form, then the components of T are:*

- *The top type $\top$, or*
- *a read-only record type $\texttt{readonly}\ \{f : T'\}$.*

From this key result we can show that preservation holds for System $\mathsf{F}_{<:\mathsf{M}}$.

Theorem 4.9 (Preservation of System $\mathsf{F}_{<:\mathsf{M}}$). *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 $\Gamma \mid \Sigma' \vdash t : T$.*

Conversely, values given a non-read-only record type must be an unsealed collection of references.

Evaluation $\boxed{\langle s, \sigma \rangle \longrightarrow \langle t, \sigma' \rangle}$

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

$$\frac{l_i \notin \sigma}{\langle \{x_i : v_i\}, \sigma \rangle \longrightarrow \langle \{x_i : l_i\}, (\sigma, l_1 : v_1, l_2 : v_2, \ldots) \rangle} \text{ (RECORD-STORE)}$$

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

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

$$\langle (\Lambda(X <: S).t)[T], \sigma \rangle \longrightarrow \langle t[X \mapsto T], \sigma \rangle \text{ (BETA-T)}$$

$$\frac{l : v \in \sigma}{\langle\ (\mathtt{seal}\ \{\ldots x : l \ldots\})\ .x, \sigma \rangle \longrightarrow \langle\ \mathtt{seal}\ v\ , \sigma \rangle} \text{ (SEALED-FIELD)}$$

$$\langle\ \mathtt{seal}\ (\lambda x.t)\ , \sigma \rangle \longrightarrow \langle \lambda x.t, \sigma \rangle \text{ (SEAL-ELIM-ABS)}$$

$$\langle\ \mathtt{seal}\ (\Lambda(X <: S).t)\ , \sigma \rangle \longrightarrow \langle \Lambda(X <: S).t, \sigma \rangle \text{ (SEAL-ELIM-TABS)}$$

$$\langle\ \mathtt{seal}\ \mathtt{seal}\ v\ , \sigma \rangle \longrightarrow \langle \mathtt{seal}\ v, \sigma \rangle \text{ (SEAL-ELIM-MULTIPLE)}$$

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

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

Fig. 7. Reduction rules for System $\mathsf{F}_{<:\mathsf{M}}$

Lemma 4.10. *Suppose $\varnothing \mid \Sigma \vdash v : \{f : T\}$ for runtime value v. Then v is an unsealed runtime record where field f maps to some location l.*

This lemma is needed to prove progress.

Theorem 4.11 (Progress for System $\mathsf{F}_{<:\mathsf{M}}$). *Suppose $\varnothing \mid \Sigma \vdash \sigma$ and $\varnothing, \Sigma \vdash s : T$. Then either s is a value or there is some t and σ' such that $\langle s, \sigma \rangle \longrightarrow \langle t, \sigma' \rangle$.*

5 STATIC IMMUTABILITY SAFETY

Armed with Progress and Preservation, we can state immutability safety for full System $\mathsf{F}_{<:\mathsf{M}}$. System λ_{M} allows us to show that sealed records are never used to mutate their underlying referenced values. System $\mathsf{F}_{<:\mathsf{M}}$ shows that well-typed programs using seals do not get stuck. To prove immutability safety for System $\mathsf{F}_{<:\mathsf{M}}$, one problem still remains – System $\mathsf{F}_{<:\mathsf{M}}$ allows records that are not sealed to be given a read-only type. We still need to show that records with such a type are not used to mutate their values. In other words, we need to show that records with a read-only type *could be* sealed, and that the resulting program would execute in the same way.

We will do this by showing that, given an original, well-typed System $\mathsf{F}_{<:\mathsf{M}}$ program s, we can add `seal`s to its read-only subterms to obtain a new, well-typed System $\mathsf{F}_{<:\mathsf{M}}$ program t, and furthermore that t behaves the same way as s, up to having additional `seal`s in the resulting state.

The first step is to show that `seal`ing does not disturb the typing judgment for terms.

Lemma 5.1. *Suppose $\Gamma \mid \Sigma \vdash t : \mathtt{readonly}\ T$. Then $\Gamma \mid \Sigma \vdash \mathtt{seal}\ t : \mathtt{readonly}\ T$.*

Proof. By (seal), $\Gamma \mid \Sigma \vdash \texttt{seal}\ t : \texttt{readonly readonly}\ T$. Then since $\texttt{readonly readonly}\ T <: \texttt{readonly}\ T$, by (sub), $\Gamma \mid \Sigma \vdash \texttt{seal}\ t : \texttt{readonly}\ T$, as desired. □

From this, given a term s and a typing derivation for s, $D = \Gamma \mid \Sigma \vdash s : T$, we can seal those subterms of s that are given a read-only type in D.

Lemma 5.2. *Let C be a term context with n holes, and let $s = C[s_1, s_2, s_3, \ldots, s_n]$ be a term. Suppose D is a typing derivation showing that $\Gamma \mid \Sigma \vdash s : T$. Suppose also that D gives each subterm s_i of s a type* $\texttt{readonly}\ T_i$*. Then $s' = s[\texttt{seal}\ s_1, \texttt{seal}\ s_2, \ldots, \texttt{seal}\ s_n]$ has the following properties:*

(1) *$s \leq s'$, and*
(2) *There exists a typing derivation D' showing that $\Gamma \mid \Sigma \vdash s' : T$ as well.*

Proof. (1) is by definition. As for (2), to construct D', walk through the typing derivation D showing that $\Gamma \mid \Sigma \vdash s : T$. When we reach the point in the typing derivation that shows that s_i is given the type $\texttt{readonly}\ T_i$, note that $\texttt{seal}\ s_i$ can also be given the type $\texttt{readonly}\ T_i$ by the derivation given by Lemma 5.1. Replace the sub-derivation in D with the derivation given by Lemma 5.1 to give a derivation in D' for $\texttt{seal}\ s_i$, as desired. □

This motivates the following definition.

Definition 5.3. Let s be a term and let $D = \Gamma \mid \Sigma \vdash s : T$ be a typing derivation for s. Define $\texttt{crest}(s, D)$ to be the term constructed from s by replacing all subterms s_i of s given a read-only type in D by $\texttt{seal}\ s_i$.

A crested term essentially seals any sub-term of the original term that is given a read-only type in a particular typing derivation. By definition, for any term s and typing derivation D for s, we have $s \leq \texttt{crest}(s, D)$. Moreover, a crested term can be given the same type as its original term as well.

Lemma 5.4. *Let s be a term and let $D = \Gamma \mid \Sigma \vdash s : T$ be a typing derivation for s. Then $s \leq \texttt{crest}(s, D)$, and there exists a typing derivation showing that $\Gamma \mid \Sigma \vdash \texttt{crest}(s, D) : T$ as well.*

Now by progress – Theorem 4.11 – we have that for any well typed term s with typing derivation $D = \varnothing \mid \Sigma \vdash s : T$, its protected – crested – version $\texttt{crest}(s, D)$ will also step. By preservation – Theorem 4.9 – we have that $\texttt{crest}(s, D)$ either eventually steps to a value or runs forever, but never gets stuck. It remains to relate the reduction steps of $\texttt{crest}(s, D)$ to those of s, and specifically to show that if one reduces to some specific value and store, then the other also reduces to an equivalent pair of value and store.

We will do so by using the dynamic immutability safety properties proven in Section 3. System $\mathsf{F}_{<:\mathsf{M}}$ satisfies the same sealing-equivalence properties as System λ_{M} – seals do not affect reduction, except perhaps by introducing other seals. The following are analogues of Lemmas 3.4, 3.5, and 3.6 for System $\mathsf{F}_{<:\mathsf{M}}$.

Lemma 5.5. *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 $|t'| < |t|$.

Lemma 5.6. *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 $\sigma'_s \leq \sigma'_t$.*

Lemma 5.7. *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* $\sigma'_s \leq \sigma'_t$.

Stepping back, we can see using Lemma 5.6 that one step of s to a term s' corresponds to *finitely many* steps of $\texttt{crest}(s, D)$; every step that $\texttt{crest}(s, D)$ takes either removes a `seal` or corresponds to a reduction step that s originally took. Hence $\texttt{crest}(s, D)$ eventually steps to a term t' such that $s' \leq t'$, preserving the desired equivalence of reduction between s and $\texttt{crest}(s, D)$. The following is a generalization of the previous statement to two arbitrarily chosen well-typed terms s and t satisfying $s \leq t$.

Lemma 5.8. *Suppose* $\varnothing, \Sigma \vdash \sigma_s$ *and* $\varnothing, \Sigma \vdash s : T$. *Suppose* $\langle s, \sigma_s \rangle \longrightarrow \langle s', \sigma'_s \rangle$. *For* $\sigma_s \leq \sigma_t$, *and* $s \leq t$, *such that* $\Gamma, \Sigma \vdash \sigma_s$ *and* $\Gamma, \Sigma \vdash t : T$, *we have that* $\langle t, \sigma_t \rangle \longrightarrow^* \langle t', \sigma'_t \rangle$ *where* $s' \leq t'$ *and* $\sigma'_s \leq \sigma'_t$.

Proof. From Theorem 4.11 we have that there exists a t' and σ'_t such that $\langle t, \sigma_t \rangle \longrightarrow \langle t', \sigma'_t \rangle$. By Lemma 5.6 we have that either $s \leq t'$, $\sigma_s \leq \sigma'_t$, and $|t'| < |t|$, or that $s' \leq t'$ and $\sigma'_s \leq \sigma'_t$. If $s' \leq t'$ and $\sigma'_s \leq \sigma'_t$ we are done. Otherwise, observe that since $|t'| < |t|$, a seal was removed. This can only occur a finite number of times, as t and t' have at most a finite number of seals, so we can simply loop until we obtain a t' and σ'_t such that $s' \leq t'$ and $\sigma'_s \leq \sigma'_t$. Note that Preservation – Theorem 4.9 allows us to do so as each intermediate step t' can be given the same type $\Gamma \mid \Sigma \vdash t' : T$. □

Finally, when s eventually reduces to a value v, we can use Lemma 5.5 to show that $\texttt{crest}(s, D)$ reduces to a similar value v' as well. Again, the following is a generalization of the previous statement to two arbitrarily chosen well-typed terms s and t satisfying $s \leq t$.

Lemma 5.9. *Suppose* $\varnothing, \Sigma \vdash \sigma_s$ *and* $\varnothing, \Sigma \vdash s : T$ *such that* s *eventually reduces to a value* v_s *– namely, that* $\langle s, \sigma_e \rangle \longrightarrow^* \langle v_s, \sigma'_s \rangle$ *for some* σ'_s.

Then for any t *such that* $s \leq t$ *and* $\varnothing, \Sigma \vdash t : T$, *we have that* t *eventually reduces to some value* v_t, *– namely* $\langle t, \sigma_e \rangle \longrightarrow^* \langle v_t, \sigma'_t \rangle$, *such that* $v_s \leq v_t$ *and* $\sigma'_s \leq \sigma'_t$.

Proof. For each step in the multi-step reduction from $\langle s, \sigma_e \rangle \longrightarrow^* \langle v_s, \sigma'_s \rangle$ we can apply Lemma 5.8 to show that $\langle t, \sigma_t \rangle$ eventually reduces to $\langle t', \sigma'_t \rangle$ where $v_s \leq t'$ and $\sigma'_s \leq \sigma'_t$. Now by Theorem 4.11 and Lemma 5.5 we have that either t' is a value, in which case we are done, or that $\langle t', \sigma'_t \rangle$ steps to $\langle t'', \sigma'_s \rangle$ where $v_s \leq t''$. Again, we can only take a finite number of steps of this fashion as the rule which reduces $t' \longrightarrow t''$ can only be one that removed a seal, so eventually we obtain a value v_s such that $\langle t, \sigma_s \rangle \longrightarrow^* \langle v_t, \sigma'_t \rangle$ with $v_s \leq v_t$, and $\sigma'_s \leq \sigma'_t$, as desired. Again, note that Preservation – Theorem 4.9 allows us to do so as each intermediate step t' can be given the same type $\Gamma \mid \Sigma \vdash t' : T$. □

Now from Lemma 5.9 we obtain our desired immutability safety results as a consequence – namely, given a well-typed term s that reduces to a value v_s, any references in s with a `readonly` type are never actually mutated, since they can be transparently sealed (which does not change the typing) to no ill effect. Formally, our main result is:

Theorem 5.10. *Suppose* s *is a term,* $D = \varnothing \mid \Sigma \vdash s : T$ *is a typing derivation for* s, *and let* σ_s *be some initial store such that* $\varnothing \mid \Sigma \vdash \sigma_s$. *Then:*

- $\texttt{crest}(s, D)$ *can be given the same type as* s *–* $\varnothing \mid \Sigma \vdash \textit{crest}(s, D) : T$.

Moreover, if $\langle s, \sigma_s \rangle \longrightarrow^* \langle v_s, \sigma'_s \rangle$, *for some value* v_s, *then:*

- $\texttt{crest}(s, D)$ *will reduce to a value* v_t *–* $\langle \textit{crest}(s, D), \sigma_e \rangle \longrightarrow^* \langle v_t, \sigma'_t \rangle$, *such that*
- v_t *and* σ'_t *are equivalent to* v_s *and* σ'_s, *modulo additional seals – namely, that* $v_s \leq v_t$ *and* $\sigma'_s \leq \sigma'_t$.

Finally, it is useful to show that the converse result is also true; seals can be safely *removed* without affecting reduction. First note that `seal`s themselves can be transparently removed without affecting the types assigned to the term.

LEMMA 5.11. *Suppose* $\Gamma \mid \Sigma \vdash \texttt{seal}\ s : T$*. Then* $\Gamma \mid \Sigma \vdash s : T$*.*

Moreover, the following analogue of Lemma 3.8 holds in System $\mathsf{F}_{<:\mathsf{M}}$.

LEMMA 5.12. *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* $v_s \leq v_t$ *and* $\sigma_s' \leq \sigma_t'$*.*

While Lemma 5.12 is enough to show that when $s \leq t$, if t reduces to a value then so does s, we need Lemma 5.13 to reason about the types of s and v_s.

LEMMA 5.13. *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* $\langle s, \sigma_s \rangle \longrightarrow^* \langle v_s, \sigma_s' \rangle$ *for some value* v_s *such that* $v_s' \leq v_s'$ *and* $\sigma_s' \leq \sigma_t'$*. Moreover,* $\Gamma \mid \Sigma \vdash s : T$ *and* $\Gamma \mid (\Sigma', \Sigma) \vdash v_s : T$ *for some* Σ' *as well.*

PROOF. By Lemma 5.11 we can show that $\Gamma \mid \Sigma \vdash s : T$. By Lemma 5.12 we have that v reduces to some value v_s. By preservation – Theorem 4.9 we have that v_s has type T, as desired. □

6 MECHANIZATION

Our mechanization of System $\mathsf{F}_{<:\mathsf{M}}$ is based on the mechanization of System $\mathsf{F}_{<:}$ by Aydemir et al. [2008]. Our mechanization is a faithful model of System $\mathsf{F}_{<:\mathsf{M}}$ as described in this paper except for one case. To facilitate mechanization, reduction in our mechanization is done via explicit congruence rules in each reduction rule instead of an implicit rule for reducing inside an evaluation context, similar to how Aydemir et al. [2008] originally mechanize System $\mathsf{F}_{<:}$ as well.

Proofs for all lemmas except for Theorem 5.10 and Lemmas 3.9, 5.2, and 5.4 have been mechanized using Coq 8.15 in the attached artifact. Theorem 5.10 and Lemmas 5.2, 5.4, and 5.13 have not been mechanized as they require computation on typing derivations which is hard to encode in Coq as computation on `Prop` cannot be reflected into `Set`. Lemma 3.9 has been omitted from our mechanization as it is hard to formally state, let alone prove, in a setting where reduction is done by congruence, though it almost follows intuitively from how the reduction rules are set up.

7 RELATED AND FUTURE WORK

7.1 Limitations – Parametric Mutability Polymorphism

Unlike other systems, System $\mathsf{F}_{<:\mathsf{M}}$ does not support directly encoding mutability polymorphism, neither through a restricted `@polyread` modifier as seen in Huang et al. [2012], nor through explicit mutability variables as seen in Gordon et al. [2012].

This is a true limitation of System $\mathsf{F}_{<:\mathsf{M}}$, however, we note that it is possible to desugar parametric mutability polymorphism from a surface language into a core calculus like System $\mathsf{F}_{<:\mathsf{M}}$. As Huang et al. [2012] point out in their work, parametric mutability polymorphism can be desugared via *overloading*, noting that overloading itself can be dealt with in a surface language before desugaring into a base calculus, as seen before with Featherweight Java [Igarashi et al. 2001].

For example, consider the following top-level parametric function, `access`, which is parametric on 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 }
```

Nested and first-class functions are a little trickier but one can view a polymorphic, first-class function value as a read-only record packaging up both overloads.

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

It would be interesting future work to see how one could add parametric mutability polymorphism to System $F_{<:M}$.

7.2 Future Work – Algorithmic Subtyping

The subtyping rules of System $F_{<:M}$ are fairly involved and it is difficult to see if an algorithmic subtyping system could be devised. We would conjecture that one could do so, using techniques from Muehlboeck and Tate [2018]'s integrated subtyping work, but nonetheless algorithmic subtyping for System $F_{<:M}$ remains an interesting and open problem.

7.3 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 [Tschantz and Ernst 2005], the term "viewpoint adaptation" was first coined by Dietl et al. [2007]. 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 – and used it to implement an *ownership* system for Java references in order to tame *aliasing* in Java programs.

7.4 Reference Immutability

Reference immutability has long been studied in the context of existing object-oriented languages such as Java and C#, and more recently has been studied in impure, functional languages like Scala.

roDOT *[Dort and Lhoták 2020]:* roDOT extends the calculus of Dependent Object Types [Amin et al. 2016] with support for reference immutability. In their system, immutability constraints are expressed through a type member field $x.M$ of each object, where x is mutable if and only if $M \leq \bot$, and x is read-only if and only if $M \geq \top$. Polymorphism in roDOT is out of all reference immutability systems closest to how polymorphism is done in System $F_{<:M}$. Type variables quantify over full types, and type variables can be further restricted to be read-only as in System $F_{<:M}$. Constructing a read-only version of a type, like how we use `readonly` in System $F_{<:M}$, is done in roDOT by taking an intersection with a bound on the type member M. For example, `inplace_map` from before could be expressed in roDOT using an intersection type to modify immutability on the type variable `X`:

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

Dort et. al. also prove that roDOT respects immutability safety, but with different techniques than how we show immutability safety in System $F_{<:M}$. Instead of giving operational semantics with special forms that guard references from being mutated, and relying on progress and preservation to imply static safety, they take a different approach and show instead that values on the heap that change during reduction must be reachable by some statically-typed mutable reference in the initial program. roDOT is a stronger system than System $F_{<:M}$, as in particular mutabilities can be combined. For 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 is not directly expressible in System $F_{<:M}$ and can only be expressed using overloading:

```
def getF[T](x: @readonly {f : T}): @readonly T = x.f
def getF[T](x: {f : T}) : T = x.f
```

However, in contrast, roDOT is significantly more complicated than System $F_{<:M}$.

Immutability for C# *[Gordon et al. 2012]:* Of all the object calculi with reference immutability, the calculus of Gordon et al. [2012] is closest to that of roDOT in terms of flexibility. Polymorphism is possible over *both* mutabilities and types in Gordon's system, but must be done separately; type variables instead quantify over base types that have not been qualified with some immutability annotation, whether that be read-only or mutable. 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 the mutability operator `~>`. For example, `getF` could be written as the following in Gordon's system:

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

Unlike roDOT however, 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
```

Gordon et. al. also demonstrate the soundness and immutability safety of their system but through an embedding into a program logic [Dinsdale-Young et al. 2013].

Javari *[Tschantz and Ernst 2005]:* 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 System $F_{<:M}$, 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; }
}
```

However, this example is inexpressible in the core calculus Lightweight Javari, as `@readonly T` is ill-formed. 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 2005] claim that the soundness of Lightweight Javari reduces to showing the soundness of Lightweight Java, but no formal proof is given.

ReIm*: [Huang et al. 2012]:* 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 and System $F_{<:M}$, ReIm restricts how qualifiers interact with generics. ReIm's polymorphism model is similar to that of Gordon et al. [2012] – type variables range over unqualified types. However, ReIm has no mechanism for mutability polymorphism, and therefore `getF` cannot be written in ReIm at all. Unlike other related work, neither soundness nor immutability safety is proven to hold for ReIm.

Immutability Generic Java: *[Zibin et al. 2007]:* 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. Type variables by the nature of how immutability is expressed in IGJ 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.
```

7.5 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 [Stroustrup 2007]. 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`:

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

BoxedInt b{5};
const HasF<BoxedInt> x{&b};
BoxedInt* NotGreat = x.f; // OK, as x stores a constant pointer to a mutable
    BoxedInt
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 [Bright et al. 2020]. 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 $\mathsf{F}_{<:\mathsf{M}}$, 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.

7.6 Type Qualifiers and Polymorphism

Foster et al. [1999] formalize a system for enriching types with qualifiers with support for polymorphism over both ground, unqualified types and qualifiers themselves. In this setting, readonly can be viewed as a type qualifier, similar to how C++'s const can be viewed as a qualifier in [Foster et al. 1999]. The resulting calculus is similar to the calculus of [Gordon et al. 2012] restricted only to reference immutability qualifiers.

7.7 Contracts

Our approach to sealing references is similar to and was inspired by practical programming experience with Racket contracts – [Strickland et al. 2012]. Sealing, 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. 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 3.4, 3.5, and 3.6 can be viewed as an analogue of [Strickland et al. 2012, 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.

8 CONCLUSION

We contributed a *simple* and *sound* treatment of reference immutability in System $\mathsf{F}_{<:}$. We show how a simple idea, *sealing* references, can provide dynamic immutability safety guarantees in an untyped context – System λ_{M} – and how soundness and System $\mathsf{F}_{<:}$-style polymorphism can be recovered in a typed calculus System $\mathsf{F}_{<:\mathsf{M}}$ which builds on both System λ_{M} and System $\mathsf{F}_{<:}$. Our hope is to enable reference immutability systems in functional languages via this work, by giving simple soundness foundations in a calculus (System $\mathsf{F}_{<:}$) which underpins many impure functional languages today.

DATA-AVAILABILITY STATEMENT

The artifact that supports this paper is available on Software Heritage [Lee and Lhoták 2023a] and on the ACM Digital Library [Lee and Lhoták 2023b].

ACKNOWLEDGMENTS

We thank Yaoyu Zhao for his interesting discussions on reference immutability, and we thank Alexis Hunt and Hermann (Jianlin) Li for their useful feedback on early drafts of this work. We also thank our anonymous reviewers for their helpful comments which have improved the final draft of this work.

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada and by an Ontario Graduate Scholarship.

REFERENCES

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

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

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

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

Werner Dietl, Sophia Drossopoulou, and Peter Müller. 2007. Generic Universe Types. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 28–53.

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

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

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

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

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

Philipp Haller and Ludvig Axelsson. 2017. Quantifying and Explaining Immutability in Scala. *Electronic Proceedings in Theoretical Computer Science* 246 (apr 2017), 21–27. https://doi.org/10.4204/eptcs.246.5

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

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 (may 2001), 396–450. https://doi.org/10.1145/503502.503505

Edward Lee and Ondřej Lhoták. 2023a. *Artifact for the OOPSLA 2023 paper 'Simple Reference Immutability for System F-sub'*. https://archive.softwareheritage.org/swh:1:rev:709bb054f0eb4a891c9c54065a255dff0a360800;origin=https://github.com/e45lee/simple-fsub-mutability-proofs;visit=swh:1:snp:04d537961ccf54bc85606d60b180c2e8083b14ba

Edward Lee and Ondřej Lhoták. 2023b. *Artifact for the OOPSLA 2023 paper 'Simple Reference Immutability for System F-sub'*. https://doi.org/10.1145/3580414

Fabian Muehlboeck and Ross Tate. 2018. Empowering Union and Intersection Types with Integrated Subtyping. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 112 (oct 2018), 29 pages. https://doi.org/10.1145/3276482

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

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 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 943–962. https://doi.org/10.1145/2384616.2384685

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

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

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

Received 2023-04-14; accepted 2023-08-27