Simple Reference Immutability for System $\mathtt{F}_{<:}$

Edward Lee and Ondřej Lhoták

University of Waterloo

October 26, 2023

- Pure, functional, programs are great, as they are easy to reason about.

- Pure, functional, programs are great, as they are easy to reason about.
- But side effects are useful!

Let's build a compiler

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

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

Mutation is often handy!

Some mutation is bad!

```
object codegen {
  def go() = {
    analysis.table = ??? /* oops */
  }
}
```

We may want to forbid changing the symbol table outside of semantic analysis, for example.

Mutation is hard to control!

```
object analysis {
  private val table[analysis] = ???
  def symbolTable = table.toMap
    // create immutable copy of table
}
```

Mutation is hard to control!

```
object analysis {
  private val table[analysis] = ???
  def symbolTable = table.toMap
    // create immutable copy of table
}
```

This doesn't work!

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

While we created an immutable map, we created an immutable map mapping strings to mutable `Procedure` objects.

So we can still mutate values that were transitively reachable from the map.

Enter Reference Immutability

- References can be marked `readonly`.
- `readonly` references can only be read from (naturally)!

Enter Reference Immutability

- References can be marked `readonly`.
- `readonly` references can only be read from (naturally)!
- References *read* from a `readonly` reference are themselves `readonly`. This viewpoint adapts references to ensure transitive immutability.

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

// OK, takes in a regular (mutable) reference
def good(x : Pair[Int]) =
  { x.first = 5 }
```

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

// OK, takes in a regular (mutable) reference
def good(x : Pair[Int]) =
  { x.first = 5 }

// Bad, mutates through a read-only
// reference
def bad1(y : @readonly Pair[Int]) =
  { y.first = 7 }
```

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

// OK, takes in a regular (mutable) reference
def good(x : Pair[Int]) =
  { x.first = 5 }

// Bad, mutates through a read-only
// reference
def bad1(y : @readonly Pair[Int]) =
  { y.first = 7 }

// Bad, mutates through a (read-only)
// reference read from a read-only
// reference.
def bad2(y : @readonly Pair[Pair[Int]]) =
  { y.first.first = 5 }
```

Long line of work here!

- Javari (Tschantz 2005).
- IGJ (Zibin 2007).
- ReIm (Huang 2012).
- Immutability for C# (Gordon 2012).
- roDOT (Dort 2020).

Languages with reference immutability: D, Rust, ...

Formalizations have been challenging

Today

- Simple operational semantics for modelling transitively immutable references.
- Simple polymorphic type system for modelling references given an immutable type.

Untyped operational semantics

- Start with the untyped lambda calculus with records (collections of mutable references).
- Add a new syntactic form, `seal`, which will wrap records but pass through other values.
- Forbid writes to sealed records, and reads from sealed records return `sealed` values.

Untyped operational semantics

- Start with the untyped lambda calculus with records (collections of mutable references).
- Add a new syntactic form, `seal`, which will wrap records but pass through other values.
- Forbid writes to sealed records, and reads from sealed records return `sealed` values.
- `seals` *viewpoint adapt* references at runtime to ensure transitive immutability.

This is good:

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

This is bad:

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

Transitive sealing

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

Equivalence lemmas

With this we can show the only way seals affect reduction is in getting terms stuck. If two terms are equivalent modulo seals, then they will reduce to equivalent terms or one will get stuck.

Equivalence lemmas

Definition

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

Equivalence lemmas

Definition

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

Lemma

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

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

Types for sealed references

- Start with System $\mathtt{F}_{<:}$, for polymorphism and subtyping (as immutable references are a supertype of mutable references).
- How to model making references immutable though at the type level, formally? Seals make term references immutable.
- Add a type-level operator `readonly` for marking a reference type read-only as well.

Small technical details...

- `readonly String`, `readonly readonly String`, etc, are all equivalent types (under subtyping).

Small technical details...

- `readonly String`, `readonly readonly String`, etc, are all equivalent types (under subtyping).
- Actually really annoying though to discharge this equivalence.
- We show that every type has a normal form (with at most 1 top-level `readonly`) that is equivalent to it.
- Details in the paper.

Standard Theorems

Lemma

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* $\Gamma \mid \Sigma' \vdash t : T$.

Lemma

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* $\langle s, \sigma \rangle \longrightarrow \langle t, \sigma' \rangle$.

Standard Theorems

Lemma

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* $\Gamma \mid \Sigma' \vdash t : T$.

Lemma

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* $\langle s, \sigma \rangle \longrightarrow \langle t, \sigma' \rangle$.

But what does this buy you?

Using progress and preservation

- We already get that well-typed programs with sealed references to mutable records won't get stuck.
- Since writing to sealed forms gets you stuck, well typed programs won't write to sealed references, or references marked immutable at the term level!

Using progress and preservation

- We already get that well-typed programs with sealed references to mutable records won't get stuck.
- Since writing to sealed forms gets you stuck, well typed programs won't write to sealed references, or references marked immutable at the term level!
- But what of references marked immutable only at the type level? Can we say anything about them?

Equivalence lemmas

Definition

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

Equivalence lemmas

Definition

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

Lemma

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 $|t'| < |t|$.

Equivalence lemmas

Definition

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

Lemma

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

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

Using those equivalence lemmas

- References typed readonly can be transparently sealed without affecting typing!
- Progress and preservation ensure that the modified term will reduce and not get stuck.
- Equivalence lemmas ensure that the modified program and the original program reduce the same way. *(modulo extra seals in the modified program.)*

Thank you!

Any questions?