

Mutable WadlerFest DOT

Marianna Rapoport and Ondřej Lhoták

{mrapoport, olhotak}@uwaterloo.ca
University of Waterloo

Abstract. The Dependent Object Types (DOT) calculus aims to model the essence of Scala, with a focus on abstract type members, path-dependent types, and subtyping. Other Scala features could be defined by translation to DOT.

Mutation is a fundamental feature of Scala currently missing in DOT. Mutation in DOT is needed not only to model effectful computation and mutation in Scala programs, but even to precisely specify how Scala initializes immutable variables and fields (vals).

We present an extension to DOT that adds typed mutable reference cells. We have proven the extension sound with a mechanized proof in Coq. We present the key features of our extended calculus and its soundness proof, and discuss the challenges that we encountered in our search for a sound design and the alternative solutions that we considered.

Keywords: DOT calculus, mutation, path-dependent types, Scala

1 Introduction

Abstract type members, parametric polymorphism, and mix-in composition are only a few features of Scala’s complex type system. The presence of path-dependent types has made it particularly hard to understand the interaction between the numerous language components and to come up with a precise formalization for Scala. The lack of a theoretical foundation for the language has in turn led to unsound design choices (Odersky, 2016; Amin, 2016b; Amin and Tate, 2016).

To model the interaction between Scala’s core features soundly, researchers have worked for over ten years to devise formal calculi (Odersky et al., 2003; Cremet et al., 2006; Moors et al., 2008; Amin et al., 2012, 2014; Rompf and Amin, 2015, 2016a; Amin et al., 2016; Rompf and Amin, 2016b; Amin, 2016a). We refer to the specific calculus of Amin et al. (2016) as WadlerFest DOT because several different calculi have used the name DOT. WadlerFest DOT models the key components of the Scala type system, such as type members, path-dependent types, and subtyping. The eventual intent is to formalize other constituents of the full language, such as classes and inheritance, by a translation to the core features of DOT.

However, WadlerFest DOT is still lacking some fundamental Scala features, one of which is mutation. Without mutation, it is difficult to model (mutable) variables and fields, or to reason about side effects in general.

Interestingly, mutation is even necessary to model a sound class initialization order for *immutable* fields, which are mutated once when they are initialized. At the moment, Scala’s complex initialization order can lead to programs with unintuitive behaviour of fields (Petrashko, 2016); in particular, current versions of the Scala compiler permit programs in which immutable fields are read before they have been initialized. In order for the Scala community to discuss alternative designs of the initialization order, it needs a means to specify candidate designs precisely and evaluate them formally. A sound formalization of initialization order, in turn, requires reasoning about overwriting of class members that first hold a null value from the time that they are allocated to the time that they are initialized, which is not directly possible in WadlerFest DOT.

This paper presents the Mutable DOT calculus, which is an extension to WadlerFest DOT with typed mutable references. To that end, we augment the calculus with a mutable heap and the possibility to create, update, and dereference mutable memory cells, or locations. A Scala mutable variable (`var`) can then be modelled by an immutable variable (already included in WadlerFest DOT), containing a mutable memory cell. For example, a Scala object

```
object O {
  val x = 1
  var y = 2
}
```

can be represented in mutable-DOT pseudocode as follows:¹

```
new {this: {x: Int} ∧ {y: Ref Int}} // structural type of object
  {x = 1} ∧ {y = ref 2 Int}         // definitions in object body
```

An unusual characteristic of our heap implementation is that it maps locations to variables instead of values. This design choice is induced by WadlerFest DOT’s type system, which disallows subtyping between recursive types. We show how, as a result, storing *values* on the heap would significantly limit the expressiveness of our calculus, and explain the correctness of storing *variables* on the heap.

It is not surprising that adding mutable references to a DOT calculus is *possible*. Indeed, in an update to their technical report, Rompf and Amin (2016a) report that they added them to a different DOT calculus with a big-step semantics.

WadlerFest DOT is well suited as a basis for future extension, both to specify existing higher-level Scala features by translation to a core calculus, and to formally explore new proposed extensions to Scala. It comes with a soundness proof formalized and verified in Coq. WadlerFest DOT is simpler than the other full DOT calculi, and its semantics is small-step, so the soundness proof is based on the familiar approach of progress and preservation (Wright and Felleisen, 1994).

¹ The Scala type system is nominal while WadlerFest DOT is (mostly) structural. Therefore, the Scala example assigns the object a name, while WadlerFest DOT does not.

The contributions of this paper are:

- We define an operational semantics and type system for *Mutable DOT*, an extension of the small-step WadlerFest DOT calculus with mutable references.
- We provide a mechanized type safety proof in Coq, in the form of an extension of the original WadlerFest DOT proof, which is suitable to be used for extensions of WadlerFest DOT that require mutation.²
- We discuss the challenges that we encountered in adding mutation to WadlerFest DOT, and the design choices that we made to overcome them. We conjecture that this reported experience will be helpful for adding mutation to other DOT calculi, including the OOPSLA DOT, which is to appear (Rompf and Amin, 2016b).

The rest of the paper is organized as follows. Section 2 is a brief introduction to the original WadlerFest DOT calculus. Section 3 presents the mutable DOT calculus Mutable DOT, and Section 4 outlines its type-safety proof. We discuss Mutable DOT’s design in Section 5 and present an example of using mutable references in Mutable DOT in Section 6. Related work is discussed in Section 7.

2 The WadlerFest DOT Calculus

We introduce the features of the WadlerFest DOT calculus through an example. Suppose we want to keep track of fish that live in aquariums. In Scala, we could write:

```
object AquariumModule {
  trait Aquarium {
    type Fish
    val fish : List[Fish]
  }
  def addFish(a: Aquarium, f: a.Fish) =
    new Aquarium {
      type Fish = a.Fish
      fish = a.fish + f
    }

  val piranhas = new Aquarium {
    type Fish = Piranha
    fish = List.empty[Piranha]
  }
  val goldfish = new Aquarium {
    type Fish = Goldfish
  }
```

² The mechanized proof can be found in our fork of the WadlerFest DOT proof repository (see <https://github.com/amaurremi/dot-calculus>).

```

    fish = List.empty[Goldfish]
  }
}

```

This program lets us add a fish `gf` to the `goldfish` aquarium:

```

val gf: Goldfish = ...
addFish(goldFish, gf)

```

but it will result in a type error when trying to add `gf` to the `piranha` aquarium:

```

addFish(piranhas, gf)
// type error: expected: piranhas.Fish, actual: goldFish.Fish

```

The reason the goldfish is protected from the piranhas is that the type `Fish` is *path dependent*, i.e. specific to the run-time `Aquarium` object that the fish belongs to. This allows the `addFish` method to guarantee at compile time that an aquarium `a` accepts only fish of type `a.Fish`.

The syntax, reduction semantics, and type system of the WadlerFest DOT calculus can be seen in Figures 1, 2, 3, and 4. The shaded parts are our mutation-related changes and can be ignored for now.

As a first attempt to define `Aquarium` in WadlerFest DOT, we can make it an *intersection* of two types:

```

{ Aquarium = {Fish: ⊥..⊤} ∧ {fish: List} }

```

The first type, `{Fish: ⊥..⊤}`, declares a *type member* `Fish` with lower bound `⊥` (`Nothing`) and upper bound `⊤` (`Any`). The second type, `{fish: List}`, is a *field declaration* of type `List` that represents the list of `fish` in the aquarium. The type `List` is assumed to be defined in a library and contains a type member `A` for list elements.

A problem with the current `Aquarium` implementation is that it does not say that the type of elements in the `fish` list should be `Fish`. More specifically, the list elements should have the `Fish` type of the `Aquarium runtime object` to which the list belongs. To let the `Aquarium` type refer to its own runtime object `a`, we make `Aquarium` a *recursive type*:

```

{ Aquarium = μ(a: {Fish: ⊥..⊤} ∧
                  {fish: List ∧ {A: a.Fish .. a.Fish}}) }

```

Here, to express that the type `Fish` should belong to the object `a`, we use the *type selection* `a.Fish`. The type `a.Fish` is then used as a refinement of `List`'s element type `A`. In this way, the list can contain only the fish that are allowed in the aquarium `a`.

We can now define `addFish` as a function that takes an aquarium `a` and a fish `f` of type `a.Fish`, and creates a new aquarium `a2`:

```

{ addFish = λ(a: aq.Aquarium).λ(f: a.Fish).
  ν(a2: aq.Aquarium ∧ { Fish: a.Fish .. a.Fish }) {
    Fish = a.Fish
    fish = ... }}

```

The construct $\nu(x: T)d$ creates a new object of type T with a self-variable x and definitions d . In this case, the definitions are used to initialize the `Fish` type and fish list of the new aquarium. The `Fish` type is assigned `a.Fish`. The new fish list needs to append the fish `f` to the old `a.fish` list.

To be able to add an element to a list, we need access to an `append` method, which we will get from `List`. Suppose that the `List` type is defined in a `collections` library. It can be defined as a recursive type $\mu(\text{list}: \dots)$ that declares an element type A and an `append` function. `append` takes a parameter `a` of the element type `list.A` and returns a `List` of elements that are subtypes of `a.A`:

```
let collections =  $\nu(\text{col}:$ 
  { List :  $\mu(\text{list} : (\{A: \perp..T\} \wedge$ 
    {append:  $\forall(a: \text{list}.A)(\text{col}.List \wedge \{A: \perp..a.A\})))$  } ...
in ...
```

With an `append` method on `Lists`, we can fully implement the `addFish` method. The field `a2.fish` should be defined as `a.fish.append(f)`. However, since WadlerFest DOT uses administrative normal form (ANF), before performing any operations on terms, we have to bind the terms to variables:

```
fish = let oldFish = a.fish in
  let append = oldFish.append in
  append f
```

For better readability, we introduce the following abbreviations (similar ones are used in the WadlerFest DOT paper):

$$\begin{aligned}
\{A\} &\equiv \{A: \perp..T\} & t\ u &\equiv \text{let } x = t \text{ in} \\
\{A: T\} &\equiv \{A: T..T\} & &\text{let } y = u \text{ in } xy \\
\{A <: T\} &\equiv \{A: \perp..T\} & t.L &\equiv \text{let } x = t \text{ in } x.L \\
\{D_1; D_2\} &\equiv \{D_1\} \wedge \{D_2\} & \nu(x)d: T &\equiv \nu(x: T)d
\end{aligned}$$

where D_1, D_2 are declarations or definitions of either fields or types, and L is a label of a type or field.

With those abbreviations, the full aquarium program example looks as follows:

```
let collections =  $\nu(\text{col}) \{ \dots \}$ :
  { List :  $\mu(\text{list} : (A; \text{append}: \forall(a: \text{list}.A)(\text{col}.List; A <: a.A)))$  }
in  $\nu(\text{aq}) \{$ 
  Aquarium =  $\mu(a: \{Fish; fish: \{ \text{collections}.List; A: a.Fish \} \})$ ;
  addFish =  $\lambda(a: \text{aq}.Aquarium).\lambda(f: a.Fish).$ 
     $\nu(a2) \{$ 
      Fish = a.Fish
      fish = a.fish.append f
    } : {aq.Aquarium; Fish: a.Fish}
} : {Aquarium:  $\mu(a: \{Fish; fish: \{ \text{collections}.List; A: a.Fish \} \})$ };
```

```
addFish:  $\forall(a: \text{aq.Aquarium})\forall(f: a.\text{Fish})$ 
          $\{\text{aq.Aquarium}; \text{Fish}: a.\text{Fish}\}$ 
```

3 Mutation in WadlerFest DOT

In this section, we present Mutable DOT, our extension of the WadlerFest DOT calculus with mutable references. The changes in the syntax are inspired by Pierce’s extension of the simply-typed lambda calculus with references (Pierce, 2002, Chapter 13).

Throughout this paper, we highlight the changes necessary to convert WadlerFest DOT into Mutable DOT in grey.

3.1 Abstract syntax

To support mutation, we augment the WadlerFest DOT syntax with *references* that point to *mutable memory cells*, or *locations*, as shown in Figure 1. Locations are a new kind of value that is added to the syntax, and are denoted as l . The syntax comes with three new terms to support the following reference operations:

- $\text{ref } x T$ *creates* a new reference of type T in the store and initializes it with the variable x . Section 5.3 explains why reference expressions need to contain a declared type T , unlike the references in Pierce’s book.
- $!x$ *reads* the contents of a reference x .
- $x := y$ *updates* the contents of a reference x with the variable y .

The operations that create, read, and update references operate on variables, not arbitrary terms, in order to preserve ANF.

Newly-created references become *locations*, or memory addresses, denoted as l . Locations are stored in the *store*, denoted as σ , which serves as a heap.

The store is a *map* from locations to *variables*. This differs from the common definition of a store, which maps locations to values. We discuss the motivation for this design choice in Section 5.1. In order to preserve the commonly expected intuitive behaviour of a store, we must be sure that while a variable is in the store, it does not go out of scope or change its value. We show this in Section 5.2.

Updating a store σ that contains a mapping $l \mapsto x$ with a new mapping $l \mapsto y$ overwrites x with y :

$$(\sigma[l \mapsto x])(l') = \begin{cases} x & \text{if } l = l' \\ \sigma(l') & \text{otherwise.} \end{cases}$$

Locations are typed with the reference type $\text{Ref } T$. The underlying type T indicates that the location stores variables of type T .

$\sigma ::=$ Store \emptyset empty store $\sigma[l \mapsto x]$ extended or updated store		$\text{Ref } T$ reference type	
x, y, z Variable a, b, c Term member A, B, C Type member $S, T, U ::=$ Type \top top type \perp bottom type $\{a: T\}$ field declaration $\{A: S..T\}$ type declaration $x.A$ type projection $S \wedge T$ intersection $\mu(x: T)$ recursive type $\forall(x: S)T$ dependent function		$v ::=$ Value $\nu(x: T)d$ object $\lambda(x: T).t$ lambda l location	
		$s, t, u ::=$ Term x variable v value $x.a$ selection xy application $\text{let } x = t \text{ in } u$ let binding $\text{ref } x T$ reference $!x$ dereferencing $x := y$ assignment	
		$d ::=$ Definition $\{a = t\}$ field definition $\{A = T\}$ type definition $d \wedge d'$ aggregate definition	

Fig. 1: Abstract syntax of Mutable DOT

To write concise Mutable DOT programs, we extend the abbreviations from Section 2 with the following rules:

$$\begin{aligned}
\text{ref } t \, T &\equiv \text{let } x = t \text{ in ref } x \, T \\
t := u &\equiv \text{let } x = t \text{ in let } y = u \text{ in } x := y \\
!t &\equiv \text{let } x = t \text{ in } !x \\
t; u &\equiv \text{let } x = t \text{ in } u
\end{aligned}$$

3.2 Reduction rules

Since the meaning of an Mutable DOT term depends on the store contents, we represent a program state as a tuple $\sigma \mid t$, denoting a term t that can point to memory contents in the store σ .

The new reduction semantics is shown in Figure 2:

- A newly created reference $\text{ref } x \, T$ reduces to a fresh location with an updated store that maps l to x (REF).
- Dereferencing a variable using $!x$ is possible if x is bound to a location l by a **let** expression. If so, $!x$ reduces to $\sigma(l)$, the variable stored at location l (DEREF).
- Similarly, if x is bound to l by a **let**, then the assignment operation $x := y$ updates the store at location l with the variable y (STORE).

Programs written in the Mutable DOT calculus generally do not contain explicit location values in the original program text. Locations are included as values in the Mutable DOT syntax only because terms such as $\text{ref } x \, T$ will evaluate to fresh locations during reduction.

The remaining rules are the WadlerFest DOT evaluation rules, with the only change that they pass along a store.

3.3 Type rules

The Mutable DOT typing rules, depicted in Figure 3, depend on a *store typing* Σ in addition to a type environment Γ . A store typing maps locations to the types of the variables that they store.

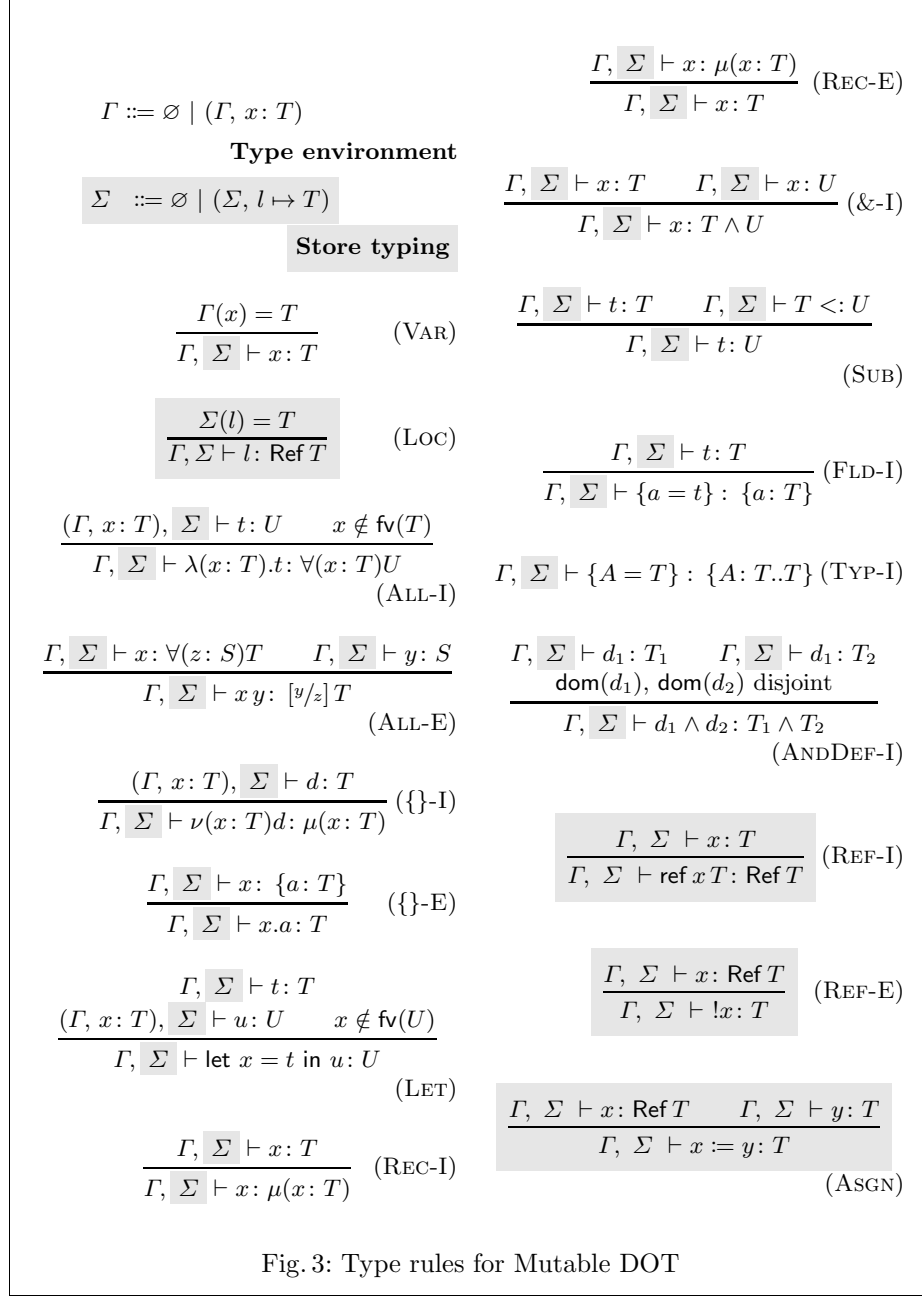
The store typing spares us the need to re-typecheck locations and allows to typecheck cyclic references (Pierce, 2002).

As an example, the following Mutable DOT program cannot be easily type-checked without an explicit store typing (using only the runtime store and the type environment):

$$p = \left(\begin{array}{l} \text{let } \text{id} = \lambda(x : \top).x \text{ in} \\ \text{let } r = \text{ref id } (\top \rightarrow \top) \text{ in} \\ \text{let } \text{id}' = \lambda(x : \top).(!r) \, x \text{ in} \\ r := \text{id}' \end{array} \right)$$

$e ::= [] \mid \text{let } x = [] \text{ in } t \mid \text{let } x = v \text{ in } e$	Evaluation context
$\frac{\sigma \mid t \mapsto \sigma' \mid t'}{\sigma \mid e[t] \mapsto \sigma' \mid e[t']}$	(TERM)
$\frac{v = \lambda(z: T).t}{\sigma \mid \text{let } x = v \text{ in } e[x y] \mapsto \sigma \mid \text{let } x = v \text{ in } e[[y/z] t]}$	(APPLY)
$\frac{v = \nu(x: T) \dots \{a = t\} \dots}{\sigma \mid \text{let } x = v \text{ in } e[x.a] \mapsto \sigma \mid \text{let } x = v \text{ in } e[t]}$	(PROJECT)
$\sigma \mid \text{let } x = y \text{ in } t \mapsto \sigma \mid [y/x] t$	(LET-VAR)
$\sigma \mid \text{let } x = \text{let } y = s \text{ in } t \text{ in } u \mapsto \sigma \mid \text{let } y = s \text{ in let } x = t \text{ in } u$	(LET-LET)
$\frac{l \notin \text{dom}(\sigma)}{\sigma \mid \text{ref } x T \mapsto \sigma[l \mapsto x] \mid l}$	(REF)
$\sigma \mid \text{let } x = l \text{ in } e[x := y] \mapsto \sigma[l \mapsto y] \mid \text{let } x = l \text{ in } e[y]$	(STORE)
$\frac{\sigma(l) = y}{\sigma \mid \text{let } x = l \text{ in } e[!x] \mapsto \sigma \mid \text{let } x = l \text{ in } e[y]}$	(DEREF)

Fig. 2: Reduction rules for Mutable DOT



Starting with an empty store, after two reduction steps we get

$$\emptyset \mid p \longmapsto^* \{l \rightarrow \text{id}'\} \mid p',$$

where

$$p' = \left(\begin{array}{l} \text{let } \text{id} = \lambda(x : \top).x \quad \text{in} \\ \text{let } r = l \quad \text{in} \\ \text{let } \text{id}' = \lambda(x : \top).(!r) x \text{ in} \\ \text{id}' \end{array} \right)$$

We would see by looking into the store that to typecheck the location l , we needed to typecheck id' . id' depends on r , which in turn refers to the location l , creating a cyclic dependency.

We therefore augment our typing rules with a store typing, allowing us to typecheck each location once and for all, at the time of a reference creation. In the example, we would know that l is mapped to $(\top \rightarrow \top)$ from the `let`-binding of r and remember this typing in Σ . To express that a term t has type T under the type environment Γ and store typing Σ , we write $\Gamma, \Sigma \vdash t : T$.

The typing rules for Mutable DOT are shown in Figure 3. The WadlerFest DOT rules are intact except that all typing derivations carry a store typing. The new rules related to mutable references are as follows:

- We typecheck locations by looking them up in the store typing. If, according to Σ , a location l stores a variable of type T , then l has type $\text{Ref } T$ (LOC).
- A newly created reference `ref x T` can be initialized with the variable x if x has type T . In particular, if x 's precise type U is a subtype of T , then x has type T by SUB, so we can still create a `ref x T` (REF-I).
- Conversely, dereferencing a variable of a reference type $\text{Ref } T$ yields the type T (REF-E).
- Finally, if x is a reference of type $\text{Ref } T$, we are allowed to store a variable y into it if y has type T . To avoid the need to add a `Unit` type to the type system, we define an assignment $x := y$ to reduce to y , so the type of the assignment is T (ASGN).

3.4 Subtyping rules

The subtyping rules of Mutable DOT include an added store typing, and a subtyping rule for references. The rules are shown in Figure 4.

Subtyping between reference types is invariant: usually, $\text{Ref } T <: \text{Ref } U$ if and only if $T = U$. Invariance is required because reference types need to be (i) covariant for reading, or dereferencing, and (ii) contravariant for writing, or assignment.

However, in WadlerFest DOT, co- and contra-variance between types does not imply type equality: the calculus contains examples of types that are not equal, yet are equivalent with respect to subtyping. For example, for any types

$\Gamma, \Sigma \vdash T <: \top$ (TOP)	$\frac{\Gamma, \Sigma \vdash x: \{A: S..T\}}{\Gamma, \Sigma \vdash x.A <: T}$ (SEL-<:)
$\Gamma, \Sigma \vdash \perp <: T$ (BOT)	$\frac{\Gamma, \Sigma \vdash S <: T \quad \Gamma, \Sigma \vdash S <: U}{\Gamma, \Sigma \vdash S <: T \wedge U}$ (<:-AND)
$\Gamma, \Sigma \vdash T <: T$ (REFL)	
$\frac{\Gamma, \Sigma \vdash S <: T \quad \Gamma, \Sigma \vdash T <: U}{\Gamma, \Sigma \vdash S <: U}$ (TRANS)	$\frac{\Gamma, \Sigma \vdash T <: U}{\Gamma, \Sigma \vdash \{a: T\} <: \{a: U\}}$ (FLD-<:-FLD)
$\Gamma, \Sigma \vdash T \wedge U <: T$ (AND ₁ -<:)	$\frac{\Gamma, \Sigma \vdash S_2 <: S_1 \quad \Gamma, \Sigma \vdash T_1 <: T_2}{\Gamma, \Sigma \vdash \{A: S_1..T_1\} <: \{A: S_2..T_2\}}$ (TYP-<:-TYP)
$\Gamma, \Sigma \vdash T \wedge U <: U$ (AND ₂ -<:)	
$\frac{\Gamma, \Sigma \vdash S <: T \quad \Gamma, \Sigma \vdash S <: U}{\Gamma, \Sigma \vdash S <: T \wedge U}$ (<:-AND)	$\frac{\Gamma, \Sigma \vdash S_2 <: S_1 \quad (\Gamma, x: S_2), \Sigma \vdash T_1 <: T_2}{\Gamma, \Sigma \vdash \forall(x: S_1)T_1 <: \forall(x: S_2)T_2}$ (ALL-<:-ALL)
$\frac{\Gamma, \Sigma \vdash x: \{A: S..T\}}{\Gamma, \Sigma \vdash S <: x.A}$ (<:-SEL)	$\frac{\Gamma, \Sigma \vdash T <: U \quad \Gamma, \Sigma \vdash U <: T}{\Gamma, \Sigma \vdash \text{Ref } T <: \text{Ref } U}$ (REF-SUB)

Fig. 4: Subtyping rules for Mutable DOT

T and U , $T \wedge U <: U \wedge T <: T \wedge U$. Yet, $T \wedge U \neq U \wedge T$. Therefore, subtyping between reference types requires both covariance and contravariance:

$$\frac{\Gamma, \Sigma \vdash T <: U \quad \Gamma, \Sigma \vdash U <: T}{\Gamma, \Sigma \vdash \text{Ref } T <: \text{Ref } U} \quad (\text{REF-SUB})$$

4 Type Safety

In this section, we outline the soundness proof of Mutable DOT as an extension of the WadlerFest DOT soundness proof (Amin et al., 2016).

To formulate the progress theorem, Amin et al. introduce the notion of an *answer*:

$$n ::= x \mid v \mid \text{let } x = v \text{ in } n.$$

The type safety of Mutable DOT is expressed as an extension to the WadlerFest DOT progress and preservation theorems. The theorems include a store σ and a store typing Σ , where Σ , unlike Γ , can be non-empty.

Theorem 1 (Progress). *If $\emptyset, \Sigma \vdash t : T$, then either t is an answer, or for any store σ such that $\emptyset, \Sigma \vdash \sigma$, there is a term t' and a store σ' such that $\sigma \mid t \mapsto \sigma' \mid t'$.*

Theorem 2 (Preservation). *If*

- $\emptyset, \Sigma \vdash t : T$
- $\emptyset, \Sigma \vdash \sigma$
- $\sigma \mid t \mapsto \sigma' \mid t'$,

then for some $\Sigma' \supseteq \Sigma$,

- $\emptyset, \Sigma' \vdash t' : T$
- $\emptyset, \Sigma' \vdash \sigma'$.

Below we describe how to extend the WadlerFest DOT proof to prove Mutable DOT soundness. Our paper comes with a mechanized Coq proof, which is also an extension of the WadlerFest DOT proof. The Coq proof can be found in our fork of the WadlerFest DOT proof repository:

<https://github.com/amaurremi/dot-calculus>

4.1 Main ideas of the WadlerFest DOT soundness proof

We start by introducing the key ideas of the WadlerFest DOT proof. We will later show how to adapt them to prove Mutable DOT type safety.

Bad bounds One of the challenges of proving DOT sound is the problem of “bad bounds” (Amin et al., 2012). For every pair of arbitrary types T and U , there exists an environment Γ such that $\Gamma \vdash T <: U$. Specifically, when type checking the function $\lambda(y: \{A: T..U\}).t$, the body t of the function is type checked in a type environment Γ in which $\Gamma(y) = \{A: T..U\}$. Then $\Gamma \vdash T <: y.A$ and $\Gamma \vdash y.A <: U$, so $\Gamma \vdash T <: U$ (using ($<:-\text{SEL}$), ($\text{SEL}<:$), and (TRANS)). In particular, if T and U are chosen as \top and \perp , respectively, then we get $\Gamma \vdash \top <: \perp$. Since every type is a subtype of \top and a supertype of \perp , this means that *all* types become equivalent with respect to subtyping in this environment. Thus, if arbitrary type environments were possible, the type system would collapse, all types would be subtypes of each other, and types would give us no information about terms.

To avoid bad bounds, Amin et al. observe that such a type environment cannot occur for an evaluation context during a concrete execution of the program. Specifically, if t' is a subterm of some term t , then the type checking rules for $\emptyset \vdash t : T$ require the subterm t' to be type checked in some specific environment Γ (i.e. $\Gamma \vdash t' : T'$). If there is some variable y such that $\Gamma \vdash y : \{A: T..U\}$, then y must be bound somewhere in t outside of t' . If t' is in an evaluation context of t (i.e. $t = e[t']$), then the syntactic definition of an evaluation context ensures that y can only be bound to a *value* by a binding of the form $\text{let } y = v \text{ in } u$. Since v is a value, it binds A with some specific type S , so its type is $\{A: S..S\}$ by (TYP-I).

Precise typing In order to reason about “good” bounds, the paper introduces the *precise typing* relation, denoted as \vdash_1 . A precise typing derivation is allowed to use only a subset of WadlerFest DOT’s type rules, so as to eliminate the rules that can lead to non-equal lower and upper type bounds.

The typing derivation of a value is said to be precise if its root is either ($\{\}$ -I) (typing an object) or (ALL-E) (typing an abstraction).³ Since the only other rule that could complete a value’s typing derivation is subsumption (SUB), precise typing computes a value’s most specific type.

Stack-based reduction rules To make more explicit the evaluation order of subterms in evaluation contexts, Amin et al. define an equivalent reduction semantics without evaluation contexts that uses a variable environment as syntactic sugar for a series of let bindings whose expressions have already been evaluated to values. In the WadlerFest DOT paper, the variable environment is called a *store*. We call it a *stack*, and reserve the term *store* for the mutable heap. The stack-based reduction relation (including our Mutable DOT extensions) is shown in Figure 5. As soon as a let-bound variable x evaluates to a value v , the binding $x \mapsto v$ is moved onto the stack γ using the Rule (LET-VALUE).

Although the stack and store appear similar, they have important differences. A stack needs to support only the lookup and append operations, since we never

³ We omit the definition of precise typing for variables because our proof modifications hardly affect it. Please refer to Amin et al.’s paper for the full definition.

$\gamma ::= \emptyset \mid (\gamma, x \mapsto v)$	Stack
$\sigma ::= \emptyset \mid \sigma[l \mapsto x]$	Store
$\frac{\gamma(x) = \nu(x : T) \dots \{a = t\} \dots}{\sigma \mid \gamma \mid x.a \mapsto \sigma \mid \gamma \mid t}$	(PROJECT)
$\frac{\gamma(x) = \lambda(z : T).t}{\sigma \mid \gamma \mid xy \mapsto \sigma \mid \gamma \mid [y/z]t}$	(APPLY)
$\sigma \mid \gamma \mid \text{let } x = y \text{ in } t \mapsto \sigma \mid \gamma \mid [y/x]t$	(LET-VAR)
$\sigma \mid \gamma \mid \text{let } x = v \text{ in } t \mapsto \sigma \mid (\gamma, x \mapsto v) \mid t$	(LET-VALUE)
$\frac{\sigma \mid \gamma \mid t \mapsto \sigma' \mid \gamma' \mid t'}{\sigma \mid \gamma \mid \text{let } x = t \text{ in } u \mapsto \sigma' \mid \gamma' \mid \text{let } x = t' \text{ in } u}$	(CTX)
$\frac{l \notin \text{dom}(\sigma)}{\sigma \mid \gamma \mid \text{ref } xT \mapsto \sigma[l \mapsto x] \mid \gamma \mid l}$	(REF)
$\frac{\gamma(x) = l}{\sigma \mid \gamma \mid x := y \mapsto \sigma[l \mapsto x] \mid \gamma \mid y}$	(STORE)
$\frac{\gamma(x) = l \quad \sigma(l) = y}{\sigma \mid \gamma \mid !x \mapsto \sigma \mid \gamma \mid y}$	(DEREF)

Fig. 5: Reduction rules for Mutable DOT in which the evaluation context is replaced with a stack for let bindings. The underlying DOT reduction rules are taken from the Coq proof that accompanies the paper of Amin et al. (2016).

perform updates on the stack. A stack also needs to have a notion of order since values can refer to variables defined earlier in the stack. A store on the other hand needs to support appending *and* overwriting locations with different variables. The store does not need to be ordered because variables cannot refer to locations. For those reasons, in the Coq formalization of the soundness proof, the stack is represented as a list, and the store as a map data structure.

The stack is an optional element of the calculus, while the store is necessary. A stack is just syntactic sugar for let-bindings: t and $\gamma \mid t'$ can be alternative, but equivalent ways of writing the same term. However, there is no way to write a term $\sigma \mid t$ as just a t . Consequently, we can write $\sigma \mid t$ and $\sigma \mid \gamma \mid t'$ as equivalent programs.

Stack correspondence The precise type of a value v cannot have bad bounds because to every type member A that v defines, it assigns a concrete type T , so the upper and lower bounds in the precise type of v must both be T : $\Gamma \vdash_! v : \{A : T..T\}$. A type environment Γ is said to *correspond* to a stack γ (written $\Gamma \sim \gamma$) if it assigns to every variable x the precise type of the corresponding value $\gamma(x)$. In such a type environment, variables cannot have type members with bad bounds.

Possible types To prove the Canonical Forms Lemmas, the WadlerFest DOT paper introduces the set of *possible types* $\text{Ts}(\Gamma, x, v)$. Informally, this set is defined to contain the types that one would expect x to have if it is bound to v , in the absence of bad bounds in Γ . The paper then proves that if $\Gamma \sim \gamma$, then all of the types T such that $\Gamma \vdash x : T$ are actually included in $\text{Ts}(\Gamma, x, \gamma)(x)$.

4.2 Adjusting Definitions to Mutable DOT

To extend the WadlerFest DOT proof to an Mutable DOT proof, we need to adjust the definitions from above.

Precise typing needs to be defined for location values.

Definition 1 (Precise Value Typing). $\Gamma, \Sigma \vdash_! v : T$ if $\Gamma, \Sigma \vdash v : T$ and the typing derivation of t ends in ($\{\}$ -I), (ALL-E), or (LOC).

Since the typing relation depends on a store typing, the *stack correspondence* relation needs to include Σ .

Definition 2 (Stack Correspondence). A stack $\gamma = \overline{x_i \mapsto v_i}$ corresponds to a type environment $\Gamma = \overline{x_i : T_i}$ and store typing Σ , written $\Gamma, \Sigma \sim \gamma$, if for each i , $\Gamma, \Sigma \vdash_! v_i : T_i$.

The set of *possible types* needs to include a store typing and two additional cases for references. First, if a value is a reference to variables of type T , then the reference type $\text{Ref } T$ should be in the set of possible types: if $\Sigma(l) = T$, then $T \in \text{Ts}(\Gamma, \Sigma, x, l)$. Second, we need to account for reference subtyping. If the

set of possible types includes a reference type $\text{Ref } T$, and U is both a sub- and supertype of T , then $\text{Ref } U$ is also in the set of possible types.

The updated definition of possible types is as follows.

Definition 3 (Possible Types). *The possible types $\text{Ts}(\Gamma, \Sigma, x, v)$ of a variable x bound in an environment Γ and corresponding to a value v is the smallest set \mathcal{S} such that*

1. *If $v = \nu(x: T)d$ then $T \in \mathcal{S}$.*
2. *If $v = \nu(x: T)d$ and $\{a = t\} \in d$ and $\Gamma, \Sigma \vdash t: T'$ then $\{a: T'\} \in \mathcal{S}$.*
3. *If $v = \nu(x: T)d$ and $\{A = T'\} \in d$ and $\Gamma, \Sigma \vdash S <: T'$, $\Gamma, \Sigma \vdash T' <: U$ then $\{A: S..U\} \in \mathcal{S}$.*
4. *If $v = \lambda(x: S).t$ and $(\Gamma, x: S), \Sigma \vdash t: T$ and $\Gamma, \Sigma \vdash S' <: S$ and $(\Gamma, x: S'), \Sigma \vdash T <: T'$ then $\forall(x: S')T' \in \mathcal{S}$.*
5. *If $v = l$ and $\Sigma(l) = T$ then $\text{Ref } T \in \mathcal{S}$.*
6. *If $\text{Ref } T \in \mathcal{S}$, $\Gamma, \Sigma \vdash T <: U$, and $\Gamma, \Sigma \vdash U <: T$, then $\text{Ref } U \in \mathcal{S}$.*
7. *If $S_1 \in \mathcal{S}$ and $S_2 \in \mathcal{S}$ then $S_1 \wedge S_2 \in \mathcal{S}$.*
8. *If $S \in \mathcal{S}$ and $\Gamma, \Sigma \vdash_! y: \{A: S..S\}$ then $y.A \in \mathcal{S}$.*
9. *If $T \in \mathcal{S}$ then $\mu(x: T) \in \mathcal{S}$.*

4.3 Stores and well-typedness

It is standard in proofs of progress and preservation to require that an environment be well-formed with respect to a typing: $\forall x. \Gamma \vdash \gamma(x): \Gamma(x)$. For stacks and stack typings, this condition follows from the definition of $\Gamma \sim \gamma$. We need to also define well-formedness for stores and store typings:

Definition 4 (Well-Typed Store). *A store $\sigma = \{l_i \mapsto x_i\}$ is well-typed with respect to an environment Γ and store typing $\Sigma = \overline{l_i \mapsto T_i}$, written $\Gamma, \Sigma \vdash \sigma$, if for each i , $\Gamma, \Sigma \vdash x_i: T_i$.*

The stronger corresponding stacks condition is not required for stores. For stacks, it is needed to ensure absence of bad bounds, because a type can depend on a stack variable (e.g. $x.A$ depends on x). No similar strengthening of well-typed stores is needed because types cannot depend on store locations.

4.4 Proof

In this section, we present the central lemmas required to prove the Mutable DOT soundness theorems.

The Canonical Forms Lemma requires a well-typed store, and a statement that values corresponding to reference types must be locations.

Lemma 1 (Canonical Forms). *If $\Gamma, \Sigma \sim \gamma$ and $\Gamma, \Sigma \vdash \sigma$, then*

1. If $\Gamma, \Sigma \vdash x: \forall(x: T)U$ then $\gamma(x) = \lambda(x: T').t$ for some T' and t such that $\Gamma, \Sigma \vdash T <: T'$ and $(\Gamma, x: T), \Sigma \vdash t: U$.
2. If $\Gamma, \Sigma \vdash x: \{a: T\}$ then $\gamma(x) = \nu(x: S)d$ for some S, d, t such that $\Gamma, \Sigma \vdash d: S, \{a = t\} \in d, \Gamma, \Sigma \vdash t: T$.
3. If $\Gamma, \Sigma \vdash x: \text{Ref } T$ then $\gamma(x) = l$ and $\sigma(l) = y$ for some l, y such that $\Gamma, \Sigma \vdash l: \text{Ref } T$ and $\Gamma, \Sigma \vdash y: T$.

The Substitution Lemma requires substitution inside of the store typing, since the types in Σ can refer to the substituted variable.

Lemma 2 (Substitution). *If $(\Gamma, x: S), \Sigma \vdash t: T$ and $\Gamma, [y/x]\Sigma \vdash y: [y/x]S$ then $\Gamma, [y/x]\Sigma \vdash [y/x]t: [y/x]T$.*

The following proposition is the main soundness result of the Mutable DOT proof. It is also an extension of the original proposition of the WadlerFest DOT soundness proof.

Proposition 1. *Let*

- $\Gamma, \Sigma \vdash t: T$,
- $\Gamma, \Sigma \sim \gamma$, and
- $\Gamma, \Sigma \vdash \sigma$.

Then either

- *t is an answer, or*
- *there exist a stack γ' , store σ' and a term t' such that $\sigma \mid \gamma \mid t \mapsto \sigma' \mid \gamma' \mid t'$ and for any such γ', σ', t' there exist environments Γ' and Σ' such that*
 - $(\Gamma, \Gamma'), (\Sigma, \Sigma') \vdash t': T$,
 - $(\Gamma, \Gamma'), (\Sigma, \Sigma') \sim \gamma$, and
 - $(\Gamma, \Gamma'), (\Sigma, \Sigma') \vdash \sigma$.

Progress and preservation (Theorems 1 and 2) follow directly from Proposition 1, if we assume Γ to be empty.

5 Discussion

In this section, we explain the design choices of Mutable DOT in more detail and discuss possible alternative implementations.

5.1 Motivation for a store of variables

One unusual aspect of the design of Mutable DOT is that the store contains variables rather than values. We experimented with alternative designs that contained values, and observed the following undesirable interactions with the existing design of WadlerFest DOT.

A key desirable property is that the store should be well-typed with respect to a store typing: $\forall l. \Gamma, \Sigma \vdash \sigma(l) : \Sigma(l)$.

Many of the WadlerFest DOT type assignment rules apply only to variables, and not to values. For example, the type $\{a : \top\}$ is not inhabited by any value, but a variable can have this type. This is because an object value has a recursive type, and the (REC-E) rule that opens a recursive type $\mu(x : \{a : \top\})$ into $\{a : \top\}$ applies only to variables, not to values. In particular, in the term

$$\text{let } x = \nu(y : \{a : \top\})\{a = t\} \text{ in } \text{ref } x \{a : \top\}$$

x has type $\{a : \top\}$ but $\nu(y : \{a : \top\})\{a = t\}$ does not, even though the let binding suggests that the variable and the value should be equal. If memory cells were to contain values, a cell of type $\{a : \top\}$ would not make sense, because no values have that type. However, since WadlerFest DOT prohibits subtyping between recursive types, this would severely restrict the polymorphism of memory cells. In particular, it would be impossible to define a memory cell containing objects with a field a of type \top and possibly additional fields. Extending the subtyping rules to apply to values as well as variables would disrupt the delicate WadlerFest DOT soundness proofs.

The above example let term demonstrates another problem: type preservation. The type system should admit the term $\text{ref } x \{a : \top\}$ because x has type $\{a : \top\}$. This term should reduce to a fresh location l of type $\text{Ref } \{a : \top\}$. But a store that maps l to $\nu(y : \{a : \top\})\{a = t\}$ would not be well typed, because the value does not have type $\{a : \top\}$.

5.2 Correctness of a store of variables

Putting variables instead of values in the store raises a concern: when we write a variable into the store, we expect that when we read it back, it will still be in scope, and it will still be bound to the same value. For example, in the following program fragment, the variable x gets saved in the store inside the function f .

```

let  $f = \lambda(x : \top). \text{ref } x \ T$  in
let  $y = v$  in
let  $r = f \ y$  in
!r
    
```

Will x go out of scope by the time we read it from the store?

The reduction sequence for this program is shown in Figure 6. Notice that before the body $\text{ref } x \ T$ of the function is reduced, the parameter x is first substituted with the argument y , which does not go out of scope.

\emptyset	$ f \mapsto \lambda(x: \top).\text{ref } x T, y \mapsto v$	$ \text{let } r = f y \text{ in } !r$	\mapsto
\emptyset	$ f \mapsto \lambda(x: \top).\text{ref } x T, y \mapsto v$	$ \text{let } r = [y/x] \text{ref } x T \text{ in } !r$	\mapsto
\emptyset	$ f \mapsto \lambda(x: \top).\text{ref } x T, y \mapsto v$	$ \text{let } r = \text{ref } y T \text{ in } !r$	\mapsto
$l \mapsto y$	$ f \mapsto \lambda(x: \top).\text{ref } x T, y \mapsto v$	$ \text{let } r = l \text{ in } !r$	\mapsto
$l \mapsto y$	$ f \mapsto \lambda(x: \top).\text{ref } x T, y \mapsto v, r \mapsto l$	$!r$	\mapsto
$l \mapsto y$	$ f \mapsto \lambda(x: \top).\text{ref } x T, y \mapsto v, r \mapsto l$	$ y$	

Fig. 6: Reduction sequence for example program

More generally, from the stack-based reduction semantics in Figure 5, it is immediately obvious that when a variable x is saved in the store using $\text{ref } x T$ or $y := x$, the only variables that are in scope are those on the stack. There are no function parameters in scope that could go out of scope when the function finishes.

Moreover, once a variable is on the stack, it never goes out of scope, and the value that it is bound to never changes. This is because the only reduction rule that modifies the stack is (LET-VALUE), and it only adds a new variable binding, but does not affect any existing bindings.

5.3 Creating references

The Mutable DOT reference creation term $\text{ref } x T$ requires both a type T and an initial variable x . The variable is needed so that a reference cell is always initialized, to avoid the need to add a `null` value to DOT. If desired, it is possible to model uninitialized memory cells in Mutable DOT by explicitly creating a sentinel null value.

Some other calculi with mutable references (e.g. Types and Programming Languages (Pierce, 2002)) do not require the type T to be given explicitly, but just adopt the precise type of x as the type for the new cell. Such a design does not fit well with subtyping in DOT. In particular, it would prevent the creation of a cell with some general type T initialized with a variable x of a more specific subtype of T .

More seriously, such a design (together with subtyping) would break type preservation. Suppose that $\Gamma, \Sigma \vdash y: S$ and $\Gamma, \Sigma \vdash S <: T$. Then we could arrive at the following reduction sequence:

\emptyset	$ f \mapsto \lambda(x: T).\text{ref } x, y \mapsto v$	$ f y$	\mapsto
\emptyset	$ f \mapsto \lambda(x: T).\text{ref } x, y \mapsto v$	$ [y/x] \text{ref } x$	\mapsto
\emptyset	$ f \mapsto \lambda(x: T).\text{ref } x, y \mapsto v$	$ \text{ref } y$	

The term at the beginning of the reduction sequence has type $\text{Ref } T$, while the term at the end, $\text{ref } y$, has type $\text{Ref } S$. Preservation would require $\text{Ref } S$ to be

a subtype of $\text{Ref } T$, but this is not the case in general since the only condition that this example imposes on S and T is that $\Gamma, \Sigma \vdash S <: T$.

6 Example

Recall the aquarium example from Section 2. Suppose we wanted to make the aquarium mutable: instead of returning a new `Aquarium`, the `addFish` method should update the aquarium’s list of fish by appending the new fish object to it. A possible implementation in Mutable DOT looks as follows:

```

let collections =  $\nu(\text{col}) \{ \dots \}$ :
  { List :  $\mu(\text{list} : (A;$ 
    append :  $\forall(a : \text{list} . A) \{ \text{col} . \text{List} ; A <: a.A \} \}$ 
in  $\nu(\text{aq}) \{$ 
  Aquarium =  $\mu(a : \{ \text{Fish};$ 
    fish : Ref { collections . List ; A : a.Fish } };
  addFish =  $\lambda(a : \text{aq} . \text{Aquarium}) . \lambda(f : a . \text{Fish}) .$ 
    let old = !(a.fish) in
    a.fish := old.append f;
    f
  } : { Aquarium :  $\mu(a : \{ \text{Fish};$ 
    fish : Ref { collections . List ; A : a.Fish } };
    addFish :  $\forall(a : \text{aq} . \text{Aquarium}) \forall(f : a . \text{Fish}) a . \text{Fish} \}$ 

```

The `fish` member of the new `Aquarium` version is now a *reference* to a list of fish, and the `addFish` changes the list to include the new fish and returns it.

7 Related Work

The semantics of mutable references presented in this paper is similar to Pierce’s extension of the simply-typed lambda calculus with typed mutable references (Pierce, 2002, Chapter 13). However, the resemblance is mostly syntactic: the language presented in the book does not include subtyping or other object-oriented features.

An extensive study of object-oriented calculi, including ones that support mutation, can be found in “A Theory of Objects” (Abadi and Cardelli, 1996). The book surveys imperative calculi with a range of advanced object-oriented features, including subtyping and inheritance, self types, and typed mutable objects (using protected storage cells). Mackay et al. (2012) developed a version of Featherweight Java (Igarashi et al., 2001) with mutable and immutable objects and formalized it in Coq. However, neither of the analyzed type systems involved path-dependent types.

The νObj calculus (Odersky et al., 2003) introduced types as members of objects, and thus path-dependent types. However, type members had only upper bounds, but not lower bounds, as they do in Scala. On the other hand,

the νObj calculus was richer than DOT, including features such as first-class classes, which are not present even in the full Scala language. Featherweight Scala (Cremet et al., 2006) was a simpler calculus intended to correspond more closely to Scala, and with decidable type-checking. However, its type system has not been proven sound. A related calculus, Scalina (Moors et al., 2008), intended to explore the design of higher-kinded types in Scala, was also not proven sound.

Amin et al. (2012) first used the name DOT for a calculus intended to be simple, and to capture only essential features, namely path-dependent types, type refinement, intersection, and union. This paper discussed the difficulties with proving such a calculus sound. The most notable challenge were counterexamples to type preservation in a small-step semantics. In general, a term can reduce to another term with a narrower type. In this DOT calculus, this narrowing could disrupt existing subtyping relationships between type members in that type.

Amin et al. (2014) examined simpler calculi with subsets of the features of DOT to determine which features cause type preservation to fail. They identified the problem of bad bounds, noted that they cannot occur in runtime objects that are actually instantiated, and conjectured that distinguishing types realizable at runtime could lead to a successful soundness proof for a DOT calculus with all of its features. Rompf and Amin (2015) confirmed this conjecture by providing the first soundness proof of a big-step semantics for a DOT calculus with type refinement and a type lattice with union and intersection. The use of a big-step semantics makes it possible to get around the problem of small steps temporarily violating type preservation, at the cost of a more complex soundness proof. An update to the technical report Rompf and Amin (2016a) reports that the authors were also able to add mutable references to this big-step version of the calculus.

WadlerFest DOT (Amin et al., 2016) defines a very specific evaluation order for the subexpressions of a DOT calculus that satisfies type preservation at each reduction step, and expressed it in a small-step semantics. The semantics uses administrative normal form (ANF) to make the necessary evaluation order explicit and clear, and to distinguish realizable types of objects instantiated at run time from arbitrary types. In particular, in the context in which a term is reduced, every ANF variable maps to a value, an actual run-time object, rather than an arbitrary term; thus, the ANF variables play the role of labels of run-time values in the semantics and its proof. The paper is accompanied by a Coq formalization of the full type soundness proof in the familiar style of progress and preservation Wright and Felleisen (1994), and is thus well suited as a basis for extensions to the calculus. It is this WadlerFest DOT calculus that we have extended with mutable references, to serve as a basis for further extensions that involve mutation.

One limitation of the WadlerFest DOT calculus is the lack of subtyping between recursive types. The calculus of Rompf and Amin (2016b), which is to appear, will remove this limitation, while maintaining a small step semantics. We hope that the experience that we have reported here on the WadlerFest DOT calculus will also be helpful for adding mutation to this new DOT calculus.

8 Conclusion

WadlerFest DOT formalizes the essence of Scala, but it lacks mutation, which is an important feature of object-oriented languages. In this paper, we show how WadlerFest DOT can be extended to handle mutation in a type-safe way.

As shown in the paper, adding a mutable store to the semantics of WadlerFest DOT is not straightforward. The lack of subtyping between recursive types leads to situations where variables and values, even though they are bound together, have incompatible types. As a result, if WadlerFest DOT were extended with a conventional store containing values, it would be impossible for a cell of a given type T to store values of different subtypes of T , thus significantly restricting the kinds of mutable code that could be expressed.

The key idea of this paper is to enable support for mutation in WadlerFest DOT by using a store that contains variables instead of values. We have shown that by using a store of variables, it is possible to extend WadlerFest DOT with mutable references in a type-safe way. This leads to a formalization of a language with path-dependent types and mutation, and also brings WadlerFest DOT one step closer to encoding the full Scala language.

Acknowledgement: This research was supported by the Natural Sciences and Engineering Research Council of Canada.

Bibliography

- Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science, Springer (1996)
- Amin, N.: Dependent Object Types. Ph.D. thesis (2016a)
- Amin, N.: Soundness issue with path-dependent type on null path. <https://issues.scala-lang.org/browse/SI-9633> (2016b)
- Amin, N., Grüter, S., Odersky, M., Rompf, T., Stucki, S.: The Essence of Dependent Object Types. Springer International Publishing, Cham (2016)
- Amin, N., Moors, A., Odersky, M.: Dependent Object Types. In: International Workshop on Foundations of Object-Oriented Languages (FOOL 2012) (2012)
- Amin, N., Rompf, T., Odersky, M.: Foundations of path-dependent types. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014. pp. 233–249 (2014)
- Amin, N., Tate, R.: Java and scala’s type systems are unsound: The existential crisis of null pointers. In: to appear in OOPSLA 2016 (2016)
- Cremet, V., Garillot, F., Lenglet, S., Odersky, M.: A core calculus for Scala type checking. In: Mathematical Foundations of Computer Science, 31st International Symposium, Slovakia (2006)
- Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23(3), 396–450 (2001)
- Mackay, J., Mehnert, H., Potanin, A., Groves, L., Cameron, N.R.: Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. In: Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (2012)
- Moors, A., Piessens, F., Odersky, M.: Safe type-level abstraction in scala. In: International Workshop on Foundations of Object-Oriented Languages (FOOL 2008) (2008)
- Odersky, M.: Scaling DOT to Scala — Soundness. <http://www.scala-lang.org/blog/2016/02/17/scaling-dot-soundness.html> (2016)
- Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: Proc. ECOOP’03. Springer LNCS (2003)
- Petrashko, D.: Making sense of initialization order in scala. <https://d-d.me/talks/scalar2016/#/> (2016)
- Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
- Rompf, T., Amin, N.: From F to DOT: type soundness proofs with definitional interpreters. *CoRR* abs/1510.05216v1 (2015), <http://arxiv.org/abs/1510.05216v1>

- Rompf, T., Amin, N.: From F to DOT: type soundness proofs with definitional interpreters. CoRR abs/1510.05216v2 (2016a), <http://arxiv.org/abs/1510.05216v2>
- Rompf, T., Amin, N.: Type soundness for dependent object types (DOT). In: to appear in OOPSLA 2016 (2016b)
- Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115(1), 38–94 (1994)