Mutable WadlerFest DOT

Marianna Rapoport
University of Waterloo
mrapoport@uwaterloo.ca

Ondřej Lhoták
University of Waterloo
olhotak@uwaterloo.ca

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.

ACM Reference format:
Marianna Rapoport and Ondřej Lhoták. 2017. Mutable WadlerFest DOT. In *Proceedings of FTfJP'17, Barcelona , Spain, June 18-23, 2017,* 6 pages.
DOI: 10.1145/3103111.3104036

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 (Amin 2016; Amin and Tate 2016; Odersky 2016).

To model the interaction between Scala's core features soundly, researchers have worked for over ten years to devise formal calculi (Amin et al. 2016, 2012, 2014; Cremet et al. 2006; Moors et al. 2008; Odersky et al. 2003; Rompf and Amin 2015, 2016a,b). We refer to the 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 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 (Amin et al. 2016; 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.

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).

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]

FTfJP'17, Barcelona , Spain
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5098-3/17/06…$15.00
DOI: 10.1145/3103111.3104036

[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.

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

- We discuss the challenges that we encountered in adding mutation to WadlerFest DOT and the design choices that we made to overcome them.

The rest of the paper is organized as follows. Section 2 presents the Mutable DOT calculus. Section 3 outlines its type-safety proof. We discuss Mutable DOT's design in Section 4. Related work is discussed in Section 5.

2 Mutation in WadlerFest DOT

In this section, we present Mutable DOT, our extension of the WadlerFest DOT calculus with mutable references. An introduction to DOT can be found in our technical report (Rapoport and Lhoták 2016) and in the WadlerFest DOT paper (Amin et al. 2016).

2.1 Abstract syntax

To support mutation, we augment the WadlerFest DOT syntax with *references* that point to *mutable memory cells*, or *locations*. The full resulting abstract syntax is shown in the accompanying technical report.

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:

- $\mathsf{ref}\, x\, T$ *creates* a new reference of type T in the store and initializes it with the variable x.
- $!x$ *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* (we follow the terminology of Pierce (2002)), 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 4.1. 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 4.2.

Updating a store σ that contains a mapping $l \mapsto x$ with a new mapping $l \mapsto y$ (denoted as $\sigma[l \to y]$) overwrites x with y.

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

In the rest of this paper, for readability, we will occasionally write program examples that relax the ANF requirements. For example, we might write an application $t\, u$ denoting $\mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{let}\ y = u\ \mathsf{in}\ x\, y$, or a reference $\mathsf{ref}\, t\, T$ to denote $\mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{ref}\, x\, T$.

2.2 Reduction rules

Amin et al. (2016). define the DOT operational semantics in terms of evaluation contexts. However, to make more explicit the evaluation order of subterms in evaluation contexts, in the Coq proof, 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. Thus, 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).

Since the meaning of a Mutable DOT term depends on the store contents, we represent a program state as a triple $\sigma \mid \gamma \mid t$, denoting a term t that can point to memory contents in the store σ and that has variable bindings in the stack γ.

The new reduction rules are as follows:

- A newly created reference $\mathsf{ref}\, x\, T$ reduces to a fresh location with an updated store that maps l to x:

$$\frac{l \notin \mathsf{dom}(\sigma)}{\sigma \mid \gamma \mid \mathsf{ref}\, x\, T \longmapsto \sigma[l \mapsto x] \mid \gamma \mid l} \qquad (\textsc{Ref})$$

- 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:

$$\frac{\gamma(x) = l \qquad \sigma(l) = y}{\sigma \mid \gamma \mid !x \longmapsto \sigma \mid \gamma \mid y} \qquad (\textsc{Deref})$$

- 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:

$$\frac{\gamma(x) = l}{\sigma \mid \gamma \mid x := y \longmapsto \sigma[l \mapsto y] \mid \gamma \mid y} \qquad (\textsc{Store})$$

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 $\mathsf{ref}\, x\, T$ 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. The full stack-based reduction relation (including our Mutable DOT extensions) is shown in the technical report.

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 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.

2.3 Type rules

The Mutable DOT typing rules 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 typechecked without an explicit store typing (using only the runtime store and the type environment):

$$p = \begin{pmatrix} \mathsf{let} & \mathsf{id} & = \lambda(x\colon \top).x & \mathsf{in} \\ \mathsf{let} & r & = \mathsf{ref}\,\mathsf{id}\,(\top \to \top) & \mathsf{in} \\ \mathsf{let} & \mathsf{id}' & = \lambda(x\colon \top).(!r)\,x & \mathsf{in} \\ r := \mathsf{id}' & & & \end{pmatrix}$$

Starting with an empty store, after a few reduction steps we get

$$\varnothing \mid p \longmapsto^{*} \{l \to \mathsf{id}'\} \mid p',$$

where

$$p' = \begin{pmatrix} \mathsf{let} & \mathsf{id} & = \lambda(x\colon \top).x & \mathsf{in} \\ \mathsf{let} & r & = l & \mathsf{in} \\ \mathsf{let} & \mathsf{id}' & = \lambda(x\colon \top).(!r)\,x & \mathsf{in} \\ \mathsf{id}' & & & \end{pmatrix}$$

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 $(\top \to \top)$ 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 $\Gamma, \Sigma \vdash t\colon T$.

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 $\mathsf{Ref}\,T$:

$$\frac{\Sigma(l) = T}{\Gamma, \Sigma \vdash l\colon \mathsf{Ref}\,T} \qquad \text{(Loc)}$$

- A newly created reference $\mathsf{ref}\,x\,T$ 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 $\mathsf{ref}\,x\,T$:

$$\frac{\Gamma, \Sigma \vdash x\colon T}{\Gamma, \Sigma \vdash \mathsf{ref}\,x\,T\colon \mathsf{Ref}\,T} \qquad \text{(Ref-I)}$$

- Conversely, dereferencing a variable of a reference type $\mathsf{Ref}\,T$ yields the type T:

$$\frac{\Gamma, \Sigma \vdash x\colon \mathsf{Ref}\,T}{\Gamma, \Sigma \vdash !x\colon T} \qquad \text{(Ref-E)}$$

- Finally, if x is a reference of type $\mathsf{Ref}\,T$, 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:

$$\frac{\Gamma, \Sigma \vdash x\colon \mathsf{Ref}\,T \qquad \Gamma, \Sigma \vdash y\colon T}{\Gamma, \Sigma \vdash x := y\colon T} \qquad \text{(Asgn)}$$

The typing rules for Mutable DOT are shown in the technical report. The WadlerFest DOT rules are intact except that all typing derivations carry a store typing.

2.4 Subtyping rules

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

Subtyping between reference types is invariant: usually, $\mathsf{Ref}\,T <: \mathsf{Ref}\,U$ 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 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:

$$\frac{\Gamma, \Sigma \vdash T <: U \qquad \Gamma, \Sigma \vdash U <: T}{\Gamma, \Sigma \vdash \mathsf{Ref}\,T <: \mathsf{Ref}\,U} \qquad \text{(Ref-Sub)}$$

3 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). The proof is based on the syntactic technique by Wright and Felleisen (Wright and Felleisen 1994).

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

3.1 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 $\lambda(y\colon \{A\colon T..U\}).t$, the body t of the function is type checked in a type environment Γ in which $\Gamma(y) = \{A\colon T..U\}$. Then $\Gamma \vdash T <: y.A$ and $\Gamma \vdash y.A <: U$, so $\Gamma \vdash T <: U$ (using (<:-Sel), (Sel-<:), and (Trans)). In particular, if T and U are chosen as $\top$ and $\bot$, respectively, then we get $\Gamma \vdash \top <: \bot$. Since every type is a subtype of $\top$ and a supertype of $\bot$, 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. (2016) 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 $\Gamma \vdash y : \{A\colon T..U\}$, 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 $\mathsf{let}\ y = v\ \mathsf{in}\ u$. Since v is a value, it binds A with some specific type S, so its type is $\{A\colon S..S\}$ by (Typ-I).

Precise typing In order to reason about "good" bounds, the paper introduces the *precise typing* relation, denoted as $\vdash_!$. 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 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: $\Gamma \vdash_! v\colon \{A : T..T\}$. 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* $\mathsf{Ts}(\Gamma, x, v)$. 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 $\Gamma \vdash x\colon T$ are actually included in $\mathsf{Ts}(\Gamma, x, \gamma(x))$.

3.2 Adjusting Definitions to Mutable DOT

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

Precise typing needs to be defined for location values.

Definition 3.1 (Precise Value Typing). $\Gamma, \Sigma \vdash_! v\colon T$ if $\Gamma, \Sigma \vdash v\colon T$ 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 3.2 (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, $\Gamma, \Sigma \vdash_! v_i : T$.

The set of *possible types* needs to include a store typing and two additional cases for references. First, if a value is a location storing variables of type T, then the reference type Ref T should be in the set of possible types: if $\Sigma(l) = T$, then $T \in \mathsf{Ts}(\Gamma, \Sigma, x, l)$. Second, we need to account for reference subtyping. If the set of possible types includes a reference type Ref T, and U is both a sub- and supertype of T, then Ref U is also in the set of possible types. The full updated definition of possible types is shown in the accompanying technical report (Rapoport and Lhoták 2016).

3.3 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: $\forall x.\Gamma \vdash \gamma(x)\colon \Gamma(x)$. 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 3.3 (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.

3.4 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 3.4 (Canonical Forms). *If* $\Gamma, \Sigma \sim \gamma$ *and* $\Gamma, \Sigma \vdash \sigma$ *, then*

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

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

Lemma 3.5 (Substitution). *If* $(\Gamma, x\colon S), \Sigma \vdash t\colon T$ *and* $\Gamma, [y/x]\Sigma \vdash y\colon [y/x]S$ *then* $\Gamma, [y/x]\Sigma \vdash [y/x]t\colon [y/x]T$.

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 3.6. *Let* $\Gamma, \Sigma \vdash t\colon T$, $\Gamma, \Sigma \sim \gamma$, *and* $\Gamma, \Sigma \vdash \sigma$. *Then either (i)* t *is a value, or (ii) there exist a stack* γ', *store* σ' *and a term* t' *such that* $\sigma \mid \gamma \mid t \longmapsto \sigma' \mid \gamma' \mid t'$ *and for any such* γ', σ', t' *there exist environments* Γ' *and* Σ' *such that* $(\Gamma, \Gamma'), (\Sigma, \Sigma') \vdash t'\colon T$, $(\Gamma, \Gamma'), (\Sigma, \Sigma') \sim \gamma$, *and* $(\Gamma, \Gamma'), (\Sigma, \Sigma') \vdash \sigma$.

Progress and preservation follow directly from Proposition 3.6.

4 Discussion

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

4.1 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 $\{a\colon \top\}$ 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

[3] We omit the definition of precise typing for variables because our proof modifications hardly affect it. Please refer to Amin et al. (2016) for the full definition.

$$\begin{array}{llll}
\emptyset & \mid\ f \mapsto \lambda(x\colon \top).\mathsf{ref}\,x\,T,\ y \mapsto v & \mid\ \mathsf{let}\ r = f\,y\ \mathsf{in}\ !r & \longmapsto \\
\emptyset & \mid\ f \mapsto \lambda(x\colon \top).\mathsf{ref}\,x\,T,\ y \mapsto v & \mid\ \mathsf{let}\ r = [y/x]\ \mathsf{ref}\,x\,T\ \mathsf{in}\ !r & \longmapsto \\
\emptyset & \mid\ f \mapsto \lambda(x\colon \top).\mathsf{ref}\,x\,T,\ y \mapsto v & \mid\ \mathsf{let}\ r = \mathsf{ref}\,y\,T\ \mathsf{in}\ !r & \longmapsto \\
l \mapsto y & \mid\ f \mapsto \lambda(x\colon \top).\mathsf{ref}\,x\,T,\ y \mapsto v & \mid\ \mathsf{let}\ r = l\ \mathsf{in}\ !r & \longmapsto \\
l \mapsto y & \mid\ f \mapsto \lambda(x\colon \top).\mathsf{ref}\,x\,T,\ y \mapsto v,\ r \mapsto l & \mid\ !r & \longmapsto \\
l \mapsto y & \mid\ f \mapsto \lambda(x\colon \top).\mathsf{ref}\,x\,T,\ y \mapsto v,\ r \mapsto l & \mid\ y &
\end{array}$$

Figure 1. Reduction sequence for example program

that opens a recursive type $\mu(x\colon \{a\colon \top\})$ into $\{a\colon \top\}$ applies only to variables, not to values. In particular, in the term

$$\mathsf{let}\ x = \nu(y\colon \{a\colon \top\})\{a = t\}\ \mathsf{in}\ \mathsf{ref}\,x\ \{a\colon \top\}$$

x has type $\{a\colon \top\}$ but $\nu(y\colon \{a\colon \top\})\{a = t\}$ 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 $\{a\colon \top\}$ would not make sense, because no values have that type.

We could try to restrict reference types to always store recursive (or function) types. However, this would severely restrict the polymorphism of memory cells, because WadlerFest DOT does not support subtyping between recursive types (subtyping between recursive structural types is not supported by Scala either). In particular, it would be impossible to define a memory cell containing objects with a field a of type $\top$ and possibly additional fields.

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

4.2 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.

$$\begin{array}{llll}
\mathsf{let}\ f & = & \lambda(x\colon \top).\mathsf{ref}\,x\,T & \mathsf{in} \\
\mathsf{let}\ y & = & v & \mathsf{in} \\
\mathsf{let}\ r & = & f\,y & \mathsf{in} \\
!r & & &
\end{array}$$

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 1. Notice that before the body $\mathsf{ref}\,x\,T$ of the function is reduced, the parameter x is first substituted with the argument y, which does not go out of scope.

More generally, from the stack-based reduction semantics, it is immediately obvious that when a variable x is saved in the store using $\mathsf{ref}\,x\,T$ 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.

Another natural question is whether a store of variables limits the expressiveness of the calculus. Since a program contains only a finite number of variables, one might think that the size of the store is restricted by that number. However, during execution, the reduction rule for function application performs capture-avoiding substitution using alpha renaming, which introduces fresh variables as necessary. Thus, the use of variables in the store does not impose any restrictions on the number of objects that can be created.

4.3 Creating references

The Mutable DOT reference creation term $\mathsf{ref}\,x\,T$ 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 $\Gamma, \Sigma \vdash y\colon S$ and $\Gamma, \Sigma \vdash S <: T$. Then we could arrive at the following reduction sequence:

$$\begin{array}{llll}
\emptyset & \mid\ f \mapsto \lambda(x\colon T).\mathsf{ref}\,x,\ y \mapsto v & \mid\ f\,y & \longmapsto \\
\emptyset & \mid\ f \mapsto \lambda(x\colon T).\mathsf{ref}\,x,\ y \mapsto v & \mid\ [y/x]\ \mathsf{ref}\,x & \longmapsto \\
\emptyset & \mid\ f \mapsto \lambda(x\colon T).\mathsf{ref}\,x,\ y \mapsto v & \mid\ \mathsf{ref}\,y &
\end{array}$$

The term at the beginning of the reduction sequence has type $\mathsf{Ref}\,T$, while the term at the end, $\mathsf{ref}\,y$, has type $\mathsf{Ref}\,S$. Preservation would require $\mathsf{Ref}\,S$ to be a subtype of $\mathsf{Ref}\,T$, but this is not the case in general since the only condition that this example imposes on S and T is that $\Gamma, \Sigma \vdash S <: T$.

5 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.

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 challenges 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.

Rompf and Amin (2016b) introduce a Coq-verified version of DOT extended with additional features. Most notably, it adds support for subtyping between recursive types. Allowing subtyping between recursive types leads to a significant increase in the proof's complexity, and it is why Lemmas 6 to 11 in the paper are required. Because Scala has nominal rather than structural typing, subtyping between recursive structural types is not needed to model it. It is sufficient to support subtyping between abstract type members, which is modelled by WadlerFest DOT.

Amin and Rompf (2017) presents mechanized soundness proofs using definitional interpreters for big-step DOT-like calculi ranging from System F to System $D_{<:>}$, and compares System $D_{<:>}$ with DOT. The paper and an earlier technical report (Rompf and Amin 2016a) discuss how to add mutable references to this class of calculi and come with a Coq formalization of System $F_{<:}$ with mutable references.

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 expresses 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.

6 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.

References

Nada Amin. 2016. Soundness issue with path-dependent type on null path. https://issues.scala-lang.org/browse/SI-9633. (2016).

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 249–272.

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent Object Types. In *FOOL 2012*.

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL 2017*. 666–679.

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of path-dependent types. In *OOPSLA 2014*. 233–249.

Nada Amin and Ross Tate. 2016. Java and Scala's type systems are unsound: the existential crisis of null pointers. In *OOPSLA 2016*. 838–848.

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A Core Calculus for Scala Type Checking. In *MFCS 2006*. 1–23.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450.

Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Robert Cameron. 2012. Encoding Featherweight Java with assignment and immutability using the Coq proof assistant. In *FTfJP 2012*. 11–19.

Adriaan Moors, Frank Piessens, and Martin Odersky. 2008. Safe type-level abstraction in Scala. In *FOOL 2008*.

Martin Odersky. 2016. Scaling DOT to Scala — Soundness. http://www.scala-lang.org/blog/2016/02/17/scaling-dot-soundness.html. (2016).

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. A Nominal Theory of Objects with Dependent Types. In *ECOOP 2003*. 201–224.

Dimitry Petrashko. 2016. Making Sense of Initialization Order in Scala. https://d-d.me/talks/scalar2016/. (2016).

Benjamin C. Pierce. 2002. *Types and Programming Languages*.

Marianna Rapoport and Ondřej Lhoták. 2016. Mutable WadlerFest DOT. (2016). http://arxiv.org/abs/1611.07610

Tiark Rompf and Nada Amin. 2015. From F to DOT: Type Soundness Proofs with Definitional Interpreters. (2015). http://arxiv.org/abs/1510.05216v1

Tiark Rompf and Nada Amin. 2016a. From F to DOT: Type Soundness Proofs with Definitional Interpreters. (2016). http://arxiv.org/abs/1510.05216v2

Tiark Rompf and Nada Amin. 2016b. Type soundness for dependent object types (DOT). In *OOPSLA 2016*. 624–641.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.