ιDOT: A DOT Calculus with Object Initialization

David R. Cheriton School of Computer Science Technical Report CS-2020-06

IFAZ KABIR, University of Alberta, Canada
YUFENG LI, University of Waterloo, Canada
ONDŘEJ LHOTÁK, University of Waterloo, Canada

The Dependent Object Types (DOT) calculus serves as a foundation of the Scala programming language, with a machine-verified soundness proof. However, Scala's type system has been shown to be unsound due to null references, which are used as default values of fields of objects before they have been initialized. This paper proposes ιDOT, an extension of DOT for ensuring safe initialization of objects. DOT was previously extended to κDOT with the addition of mutable fields and constructors. To κDOT, ιDOT adds an initialization effect system that statically prevents the possibility of reading a null reference from an uninitialized object. To design ιDOT, we have reformulated the Freedom Before Commitment object initialization scheme in terms of disjoint subheaps to make it easier to formalize in an effect system and prove sound. Soundness of ιDOT depends on the interplay of three systems of rules: a type system close to that of DOT, an effect system to ensure definite assignment of fields in each constructor, and an initialization system that tracks the initialization status of objects in a stack of subheaps. We have proven the overall system sound and verified the soundness proof using the Coq proof assistant.

CCS Concepts: • **Software and its engineering → Formal language definitions**; **Object oriented languages**.

Additional Key Words and Phrases: type safety, dependent objects, DOT, Scala, initialization

1 INTRODUCTION

The Dependent Object Types (DOT) calculus has been proposed as a formal foundation for the Scala programming language. It has been proven sound and the proof has been mechanically verified using Coq [Amin et al. 2016; Rapoport et al. 2017; Rapoport and Lhoták 2019; Rompf and Amin 2016]. Nevertheless, unsoundness has been discovered in Scala because, as a core calculus, DOT does not model all features of the full language, in this case specifically null references [Amin and Tate 2016]. When null references are used to represent missing values, they are easily modeled as a special value of an `Option` type, but it is challenging to model the use of null references as a default value for uninitialized fields of objects. The design of type systems and static analyses to ensure safe initialization of objects is an active area of research [Fähndrich and Xia 2007; Qi and Myers 2009; Servetto et al. 2013; Summers and Mueller 2011b]. To make it possible to model and study object initialization in the context of the DOT calculus, Kabir and Lhoták [2018] defined κDOT, an extension of the calculus with constructors and mutable fields.

In this paper, we propose ιDOT, a type and effect system for safe initialization in κDOT. The ιDOT calculus is informally inspired by ideas from two previous systems, Freedom Before Commitment (FBC) [Summers and Mueller 2011b] and Delayed Types (DT) [Fähndrich and Xia 2007]; in return, our mechanized proof strengthens the formal foundations of these systems. One of the key ideas in FBC is to classify every object as either `free`, meaning that some of its fields might not have been initialized[1], or `committed`, meaning that all of its fields are initialized and, transitively, all objects reachable from those fields are also `committed`. A technically tricky aspect is to soundly

[1] An object may be `free` even though all its fields are initialized if it is not transitively `committed`. Further, even if all fields transitively reachable from the object are initialized at the current time, the object may still be `free` it if can potentially point to an uninitialized object at a future point.

Authors' addresses: Ifaz Kabir, University of Alberta, Canada, ikabir@ualberta.ca; Yufeng Li, University of Waterloo, Canada, yufeng.li@uwaterloo.ca; Ondřej Lhoták, University of Waterloo, Canada, olhotak@uwaterloo.ca.

identify a set of objects that can all be considered committed when initialization of some object finishes, and the structure of the existing hand-written proofs makes them difficult to mechanize in a proof assistant. The DT system is defined in terms of splitting the heap into a set of timed regions. Each region is associated with a time at which it will be merged into the main region called Now. The static type system accumulates a system of constraints between these times, and a pointer is allowed to cross a region boundary only if the system can prove that the region containing the pointer will be merged no earlier than the region into which the pointer points. Although there is no mechanized soundness proof for the DT system, we show that the informal idea of splitting the heap into subheaps is convenient for mechanized reasoning about the tricky relationships between partially initialized objects in FBC. Although we use the idea of subheaps from DT, we show that the complexity of time variables and constraints between them is not necessary to model the simpler invariants of FBC. In particular, to express FBC, it is sufficient to prohibit pointers crossing between any of the subheaps other than the committed Now heap, which we call *free* subheaps, removing the need for constraints on the commitment times of pointers.

The ιDOT system is more than just a mechanized version of FBC applying ideas from DT, which are both calculi for traditional object-oriented languages like Java and C#. In particular, following Scala and DOT, ιDOT supports path-dependent types, which can be sound only if uninitialized paths can be ruled out. The DOT calculi of Amin et al. [2016]; Rompf and Amin [2016] side-step this issue by supporting only variable-dependent types, rather than fully path-dependent types. The pDOT calculus of Rapoport and Lhoták [2019] supports fully path-dependent types, but achieves soundness by requiring pre-existing values for all fields before an object is created, ruling out cyclic data structures involving top level objects. These existing calculi all make simplifying assumptions about fields that are inconsistent with the semantics of fields and field initialization in Scala. The κDOT calculus of Kabir and Lhoták [2018] enhances DOT with a faithful model of Scala constructors and field initialization. However, κDOT does not statically detect the possibility of initialization errors; κDOT models null fields using a bottom-typed non-terminating term and Scala programs with incorrect initialization diverge when translated into κDOT.

The ιDOT system is composed of three complementary concepts. First, ιDOT has a type system based on the type system of κDOT, which is in turn based on the type system of DOT. The techniques for reasoning about path-dependent types from DOT [Amin et al. 2016; Rapoport et al. 2017; Rompf and Amin 2016] generally apply to ιDOT as well; we focus our attention in this paper to aspects specific to initialization. Second, ιDOT has an effect system that models, within a constructor, which fields of the object being constructed have already been initialized and which have not. Third, ιDOT ensures that objects are transitively initialized using an initialization system that splits the heap into subheaps and enforces invariants about pointers that cross subheap boundaries.

This paper makes the following contributions:

- Taking inspiration from DT, we recast the ideas underlying the FBC system using the simpler intuition of free and committed subheaps, which is amenable to mechanized proof.
- We define the ιDOT extension of the κDOT calculus that guarantees safe initialization of objects. The κDOT calculus more faithfully models field initialization in Scala than previous DOT calculi.
- We prove ιDOT sound. Specifically, our proof guarantees that an ιDOT program never reads an uninitialized field of an object.
- Our safety proof is mechanically verified using the Coq proof assistant, and to the best of our knowledge is the first mechanically verified proof of the FBC system. Our Coq proof can be found at https://git.io/dot-init.

```
1  class Fruit(tr: Tree) {
2    val tree : Tree = tr
3  }
4  class Tree(fr : Fruit) {
5    val fruit : Fruit = fr
6  }
7  val mango : Fruit = new Fruit(mangoTree)
8  val mangoTree : Tree = new Tree(mango)
9  val fr = mangoTree.fruit.tree.fruit
10 // java.lang.NullPointerException
```

(a) Bad Initialization

```
11 class Fruit(tr: Tree) {
12   val tree : Tree = tr
13 }
14 class Tree {
15   val fruit : Fruit = new Fruit(this)
16 }
17 //
18 val mangoTree : Tree = new Tree
19 val fr = mangoTree.fruit.tree.fruit
20 // no errors
```

(b) Good Initialization

```
21 class Fruit[T](tr: Tree { type F = T }) {
22   type A = T
23   val tree : Tree { type F = T } = tr
24 }
25 class Tree {
26   type F
27   val fruit : Fruit[this.F] = new Fruit[this.F](this)
28 }
29 val mangoTree : Tree = new Tree
30 val fr = mangoTree.fruit.tree.fruit
```

(c) Path-Dependent Initialization

Fig. 1. Object Initialization in Scala

2 BACKGROUND

Consider the Scala code fragments in Figure 1. In Figures 1a to 1c, we attempt to create an object of type Tree, which we name mangoTree, with a field fruit of type Fruit. The Fruit object has a field fruit that recursively points back to the mangoTree.

The code in Figure 1a is incorrect! It attempts to pass the mangoTree variable to the Fruit constructor. When it is passed to the Fruit constructor, mangoTree has not been initialized yet and a null pointer is passed instead. Thus tree field of mango is null and this later leads to a NullPointerException. The code in Figure 1b rectifies this by passing only non-null pointers to constructors. But the types in this code fragment do not enforce the invariant that mangoTree and mangoTree.fruit.tree are the same tree. The type system allows mangoTree.fruit.tree to point to any Tree.

Figure 1c uses path-dependent types to enforce the invariant that the Fruit of every Tree points back to the same Tree. The Tree type consists of a type member F and a field fruit of the dependent type Fruit[this.F]. The Fruit[T] type consists of a field tree of type Tree, but refined with the type member type F = T. Since DOT does not have generic parameters, we will use the type member A to emulate generics in later DOT examples. In the Tree constructor, we leave the type member F abstract, which means that every Tree has a unique type member this.type.F. In the Fruit constructor, mismatch between the parameter T and the argument, such as new Fruit[mangoTree](new Tree) or new Fruit[Any](mangoTree), would result in type errors. In Figure 1c, the object mangoTree has an an abstract type member called F and a field with a path-dependent type of Fruit[mangoTree.F].

```
31 trait Tree {
32   type F
33   def fruit : Fruit { type A = this.F }
34 }
35 trait Fruit {
36   type A
37   def tree : Tree { type F = this.A }
38 }
39 def kTree(_ : Any) : Fruit =
40   new { tr =>
41     type F = tr.F
42     def fruit : Fruit { type A = tr.F } =
43       new { fr =>
44         type A = tr.F
45         def tree = tr
46       }
47   }
48 val mangoTree : Tree = kTree(new Object)
49 val fr1 = mangoTree.fruit
50 val tr1 = fr1.tree
51 val fr2 = tr1.fruit
```

(a) Example in WadlerFest DOT

```
52 def omega(a: Any): Nothing = omega(a)
53 val kFruit = constructor(tr : Tree) { fr =>
54   type A = tr.F
55   var def tree : Tree { type F = T } =
56     omega(omega)
57 } {
58   fr => fr.tree = tree
59 }
60 val kTree = constructor { tr =>
61   type F
62   var def fruit : Fruit { type A = F } =
63     omega(omega)
64 } { tr =>
65   val fr = new kFruit(tr)
66   tr.fruit = fruit
67 }
68 val mangoTree : Tree = new kTree()
69 val fr1 = mangoTree.fruit
70 val tr1 = fr1.tree
71 val fr2 = tr1.fruit
```

(b) Example in κDOT

Fig. 2. Examples

Fruit[mangoTree.F] in turn has a field tree, which must have the same type as mangoTree. Since the abstract type member mangoTree.F is unique to mangoTree, the field tree of Fruit[mangoTree.F] can only be initialized with mangoTree.

2.1 Formalizing Scala

Path-dependent types are one of the distinguishing features of Scala [Odersky et al. 2006]. Finding a provably type-safe calculus that features path-dependent types has proven to be difficult [Amin et al. 2014], and after a long search, Amin et al. [2016] described the WadlerFest DOT calculus. While it features path-dependent types, several features of Scala are not modelled by WadlerFest DOT, so the example from Figure 1c cannot be expressed directly. However, with the following adjustments, a similar example can be encoded as shown in Figure 2a using a Scala-like syntax.[2] Since WadlerFest DOT does not feature constructors, we define kTree instead as a function that returns an object. In WadlerFest DOT, a field behaves like a Scala method in that it contains a term that is re-evaluated each time the field is read. Accordingly, in the object returned by kTree, we write field declarations using def instead of val. Each time the field read tr1.fruit is evaluated, a new object is allocated. Therefore, fr1 and fr2 in Figure 2a refer to two distinct Fruit objects.

In WadlerFest DOT, we encode type members that remain abstract as type members assigned to themselves. Thus the abstract type member F in line 26 of Figure 1c is encoded as line 41. Notice that, for any Tree with an abstract type member F, the type of the fruit field is path dependent on

[2]WadlerFest DOT does not directly feature traits and type refinement, but Amin et al. [2016] show how they can be encoded.

the Tree itself. So a Fruit assigned to such a field must refer to the Tree in its definition, so it can only be defined in places where we can refer to the Tree, such as the body of the Tree or after the Tree has been let bound. But since fields in WadlerFest DOT are not mutable, if we define the Fruit after the Tree has been let bound, we cannot assign a reference to the Fruit to the fruit field of the Tree, so the Fruit must be defined in the body of the Tree.

From the above, we can see that WadlerFest DOT has at least the following differences from Scala: it has no notion of field mutation, it does not have a notion of constructors, and it does not allow mutual recursion between top-level objects. Not having field mutation and constructors means that WadlerFest DOT makes no distinction between allocating an object and initializing the allocated object. This means that when an object is allocated, it can only point to previously allocated top-level objects, and hence there cannot be recursion between top level objects.

To overcome some of the above limitations of WadlerFest DOT's lazily evaluated fields, Kabir and Lhoták [2018] introduced κDOT. Figure 2b shows an encoding of the example from Figure 1c in κDOT.[3] κDOT eschews anonymous objects for first-class constructors. Fields in κDOT must still be allocated with a term, which is re-evaluated each time the field is read, but fields can be mutated later to refer to heap locations (which do not evaluate further), as done in constructor bodies in the shown code fragment. Therefore, we write field declarations using var def. κDOT does not have a built-in notion of an uninitialized field or a null value, so in this example, we set the initial term of each field to omega(omega), a non-terminating term with type Nothing.[4] When new kTree() in line 68 is evaluated, the fruit field of mangoTree contains a location of a Fruit object, and unlike the WadlerFest DOT version, here fr1 and fr2 in lines 69 and 71 refer to the same object.

2.2 Freedom Before Commitment

The FBC system [Summers and Mueller 2011b] adds an initialization system to Java-like languages by adding initialization qualifiers to object references. Here we discuss the fragment of FBC without nullable types.

In the system, object references are qualified to be either `free` or `committed`. The qualifier `free` means that the object is under initialization and it is not safe to read a field of the object and treat the address that is read as non-null. The `committed` qualifier means that the object is fully initialized and reading a field would return a non-null address of a (`committed`) object. The type system enforces the following constraints: a field of a `committed` object can only be assigned an address of a `committed` object, and fields of a `free` object are not read.

Inside a constructor, FBC has a definite assignment analysis that ensures that all fields of the object being constructed are assigned non-null addresses. Constructors treat the this variable as `free`. Objects returned from a constructor are considered `committed` if the constructor was passed no arguments or only `committed` arguments, and otherwise constructors return `free` objects.

Convention: To avoid confusing `free` subheaps and the notion of free (i.e. unbound) variables, we use the convention that `free` written in monospace font refers to `free` subheaps.

2.3 Subheap Formulation

Convention: We will often write *assigning to an object* to mean assigning to a field of the object. We will also write that a constructor call or field read *returns an object* to mean that the constructor or field read returns a reference to the object.

In our subheap formulation of FBC, we conceptually split the heap into subheaps: one `committed` subheap containing only fully initialized objects and a stack of `free` subheaps that contain partially

[3] traits and type refinement can be encoded into κDOT similar to how they are encoded in WadlerFest DOT.
[4] If the fields were accessed before they were mutated with a location, the program would diverge.

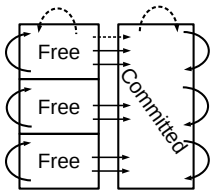


Fig. 3. Heap in Freedom Before Commitment

initialized objects. Figure 3 shows a diagram of such a heap, where the solid arrows show possible links in the heap and the dotted arrows show permitted writes. Only the following field pointers are permitted: a field may contain a pointer to a fully initialized object in the `committed` heap or to an object in the same subheap. The following writes are permitted: to a `committed` object, we may assign only a `committed` object; to a `free` object in the topmost `free` subheap, we may assign either a `committed` object or a `free` object in the same subheap; to a `free` object in a `free` subheap other than the topmost, no field writes are allowed. Field writes in a `free` subheap other than the topmost are only allowed once all subheaps above it are promoted to being part of the `committed` heap and it becomes the topmost subheap. We may only read fields of objects in the `committed` subheap, which return `committed` objects. When objects in the topmost `free` subheap are fully initialized, the entire subheap is promoted to be part of the `committed` subheap, and the next `free` subheap becomes available for allocations and field writes.

We model the FBC constructor calls as follows. If a constructor is called with only `committed` arguments, a new subheap is pushed onto the stack and the new object is allocated on the new subheap. The object will be fully initialized once the constructor completes, so the new subheap can be committed. On the other hand, if a constructor is called with a `free` argument, the new object is allocated in the topmost existing `free` subheap. The object becomes fully initialized only when the `free` argument becomes fully initialized, i.e., when its existing subheap is `committed`. The definite assignment analysis ensures that all fields in a subheap are fully initialized before being promoted.

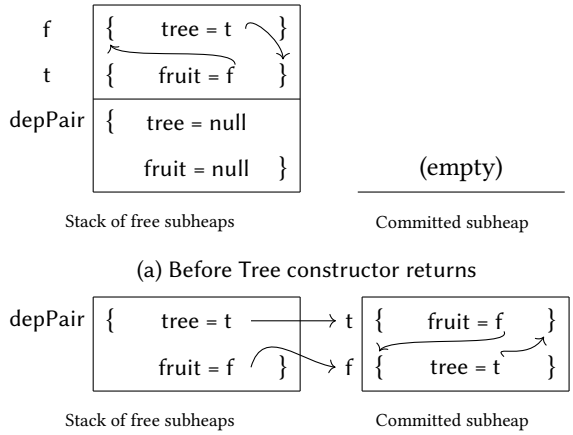
Note that the subheaps are conceptual and there are no runtime costs related to heaps being allocated. There is only one flat runtime heap, but the type and effect system assigns different objects to different subheaps.

For a concrete illustration of these ideas, consider the example in Figure 4, which builds on the example from Figure 1c and shows a case of nested object initialization. In this example, we first call the constructor TreeFruitPair, which calls the constructor Tree, which in turn calls the Fruit constructor with this as the argument, passing in a reference to the Tree under construction. Then, when the Tree object has been constructed, the TreeFruitPair constructor reads the fruit field in line 76. In order for this read to succeed, the TreeFruitPair constructor expects the inner constructor call to return a fully initialized Tree whose field fruit can be read and is non-null.

Below we step through the initialization process for the code in Figure 4. First, the stack of `free` subheaps is empty, so the call to the TreeFruitPair constructor in line 80 creates a `free` subheap for the new object, which we call depPair. When the TreeFruitPair constructor makes a call to the Tree constructor on line 74, no `free` arguments are provided (in fact it is not called with any arguments at all). Thus, a new `free` subheap is allocated for the object, say t, created by the call to the Tree constructor. These two `free` subheaps are shown in Figure 5a. The Tree constructor in Figure 1c calls the Fruit constructor, passing itself as an argument. Since this argument, the tree, is considered `free`, we do not allocate a new `free` subheap when the Fruit constructor is invoked; the new Fruit f is allocated in the same `free` subheap as the Tree t. Figure 5a shows the state of the heap at this

```
72 class TreeFruitPair {
73   val tree : Tree
74     = new Tree
75   val fruit : Fruit[tree.F]
76     = tree.fruit
77 }
78
79 val depPair : TreeFruitPair
80   = new TreeFruitPair
```

Fig. 4. Nested Initialization



(a) Before Tree constructor returns

(b) Before TreeFruitPair constructor returns

Fig. 5. Heaps corresponding to Figure 4

point in time. Note that as stated previously, no pointers are allowed between objects in different `free` subheaps. In particular, the tree and fruit fields in depPair are still null. After the call to the Tree constructor returns to the TreeFruitPair constructor, the topmost `free` subheap consisting of t and f is promoted to `committed`, as shown in Figure 5b. This is justified because we know that all objects in this `free` subheap must be transitively initialized: the definite assignment analysis concludes that every field has been assigned to point to some object, and our invariant ensures that pointers in the `free` subheap point either back into the subheap itself or into the `committed` subheap, whose objects are all transitively initialized.

Now, the only remaining `free` subheap is the one containing depPair, as observed in Figure 5b. Since t and f are now in the `committed` heap, the fields of depPair can be updated to point to them. When the TreeFruitPair constructor returns, a definite assignment analysis shows that all fields of depPair have been assigned, so we promote the remaining `free` subheap to be `committed`.

Convention: We use `free` *object* to mean object in a `free` subheap, and `committed` *object* to mean object in a `committed` subheap.

3 SEMANTICS

We now present the static and dynamic semantics of ιDOT.

3.1 Syntax

The syntax for ιDOT is shown in Figure 6. Shading indicates the differences from κDOT. We need to refer to many different kinds of stacks in our syntax, and we use the convention that any *s* following a capital letter refers to a stack, e.g. *Es* represents a stack of elements of kind *E*.

ιDOT is an object-oriented calculus with first-class functions and constructors. We use the word *literal* to mean syntax used to define functions and constructors. The term language uses an ANF-style grammar and features function and constructor calls, field reads and writes, and two kinds of let bindings: the first binds function and constructor literals and the second binds terms.

Labels and Variables

$$
\begin{array}{rl}
a, b, c & \text{Term Labels} \\
A, B, C & \text{Type Labels} \\
y & \text{Locations} \\
z & \text{Abstract Variables} \\
x, k ::= y \mid z & \text{Variables}
\end{array}
$$

Constructor Definitions

$$
\begin{array}{rl}
d ::= \{a = \texttt{null}\} & \text{Field Definition} \\
\mid \{A = T\} & \text{Type Definition} \\
\mid d \wedge d' & \text{Aggregate Definition}
\end{array}
$$

Heap Definitions

$$
\begin{array}{rl}
hd ::= \{a = \mathsf{null}\} & \text{Null Field} \\
\mid \{a = y\} & \text{Non-Null Field} \\
\mid \{A = T\} & \text{Type Definition} \\
\mid hd \wedge hd' & \text{Aggregate Definition}
\end{array}
$$

Terms

$$
\begin{array}{rl}
t, u ::= x & \text{Variable} \\
\mid \mathsf{new}\ k\,(\vec{x}) & \text{Constructor Call} \\
\mid x.a & \text{Field Read} \\
\mid x.a := x_1 & \text{Field Write} \\
\mid x\,x_1 & \text{Application} \\
\mid \mathsf{let}\ z : T = l\ \mathsf{in}\ u & \text{Literal Binding} \\
\mid \mathsf{let}\ z : T = t\ \mathsf{in}\ u & \text{Let Binding}
\end{array}
$$

Literals and Heap Items

$$
\begin{array}{rl}
l ::= \lambda(z : T).t & \text{Lambda} \\
\mid \kappa(\overrightarrow{z : Q}, z_1 : U)\,\{d\}\,t & \text{Constructor} \\
h ::= l & \text{Literal} \\
\mid v\,(z : T)\ hd & \text{Object}
\end{array}
$$

Types

$$
\begin{array}{rl}
S, T, U ::= \top & \text{Top Type} \\
\mid \bot & \text{Bottom Type} \\
\mid \forall(z : S)T & \text{Dependent Function} \\
\mid \mu(z : T) & \text{Recursive Type} \\
\mid \{a : S..T\} & \text{Field Declaration} \\
\mid \{A : S..T\} & \text{Type Declaration} \\
\mid x.A & \text{Type Projection} \\
\mid S \wedge T & \text{Type Intersection} \\
\mid K(\overrightarrow{z : Q}, z_1 : T) & \text{Constructor Type}
\end{array}
$$

Initialization Qualifiers, Qualified Types

$$
\begin{array}{rl}
i ::= \texttt{free} & \text{Uninitialized} \\
\mid \texttt{committed} & \text{Initialized} \\
Q ::= i\,T & \text{Qualified Type}
\end{array}
$$

Effects, Variable Sets, Effects of Heap Definitions

$$
\begin{array}{rl}
E ::= \emptyset \mid E \cup \{(x, a)\} & \text{Effects} \\
\mathtt{eff}\,(x, hd) ::= \{(x, a) \mid \{a = \mathsf{null}\} \in hd\} & \text{Effects of Heap Definitions}
\end{array}
$$

Contexts

$$
\begin{array}{rl}
\Gamma ::= \varepsilon \mid \Gamma,\ x : T & \text{Typing Context} \\
\Delta ::= \varepsilon \mid \Delta,\ x : i & \text{Initialization Context} \\
\mathcal{E} ::= \varepsilon \mid \mathcal{E},\ x : \{a_1, \ldots, a_n\} & \text{Effect Context}
\end{array}
$$

Frames, Stacks, Heaps, and Configurations

$$
\begin{array}{rl}
F ::= \mathsf{let}\ z : T = \square\ \mathsf{in}\ t & \text{Let Frame} \\
\mid \mathsf{return}\ y & \text{Return Frame} \\
Fs ::= \varepsilon \mid F :: Fs & \text{Frame Stack} \\
Es ::= \varepsilon \mid E :: Es & \text{Effects Stack} \\
\mathcal{E}s ::= \varepsilon \mid \mathcal{E} :: \mathcal{E}s & \text{Effect Context Stack} \\
\Sigma ::= \cdot \mid \Sigma,\ y = h & \text{Heap} \\
c ::= \langle t;\ Fs;\ \Sigma \rangle & \text{Configuration} \\
n ::= \langle y;\ \varepsilon;\ \Sigma \rangle & \text{Answer}
\end{array}
$$

Fig. 6. Syntax of ιDOT

ιDOT has the same types as κDOT with some additional initialization qualifier annotations on constructor types. For the reader's benefit, we reproduce the descriptions by Rapoport et al. [2017] with appropriate modifications and additions. An ιDOT type can be one of the following:

- A *dependent function* type $\forall(x\colon S)T$ is the type of a function with a parameter x of type S, and with the return type T, which can refer to the parameter x.
- A *(dependent) constructor* type $K(\overrightarrow{z\colon i\,T}, z_1\colon U)$ is the type of a constructor with input parameters $\vec{z}$ of qualified types $\overrightarrow{i\,T}$, and with the return type U, which can refer to the parameters $\vec{z}$ and the self-variable z_1. The initialization qualifiers $\vec{i}$ dictate the initialization state of the inputs to the constructor. The initialization state of a partially initialized input is `free` and the initialization state of a fully initialized input is `committed`.
- A *recursive* type $\mu(x\colon T)$ declares an object type T which can refer to its self-variable x.
- A *(bounded) field declaration* $\{a\colon S..T\}$ states that the field labelled a has a *setter* type S and a *getter* type T. The *setter* type S means that we may assign locations of type S to the field. The *getter* type T means that reading the field a returns an address of type T.
- A *type declaration* $\{A\colon S..T\}$ specifies that an abstract type member A is a subtype of T and a supertype of S.
- A *type projection* $x.A$ is the type assigned to the type member labelled A of the object x (ANF allows type projection only on variables).
- An *intersection* type $S \wedge T$ is the most general subtype of both S and T.
- The *bottom* type $\bot$ and the *top* type $\top$ correspond to the bottom and top of the subtyping lattice, and are analogous to Scala's Nothing and Any.

Constructors and constructor field definitions in ιDOT deserve special attention. In a constructor $\kappa(\overrightarrow{z\colon i\,T}, z_1\colon U)\,\{d\}\;t$, $\vec{z}$ are the input parameters with qualified input types $\overrightarrow{i\,T}$ and z_1 represents the this/self variable found in other object-oriented languages. The definitions d declare the field and type members of objects which will be created by the constructor. Since objects in ιDOT are dependently and recursively typed, the type member definitions in d as well as the output type U can refer to the parameters $\vec{z}$ and z_1.

In κDOT, constructor field definitions were of the form $\{a = t'\}$, where t' was an initial term that could refer to the input parameters $\vec{z}$ and the this variable z_1. When a constructor with the field definition $\{a = t'\}$ was called, an object with the field $\{a = t'\}$ was allocated, which could later be mutated. Thus, fields of objects in κDOT were never uninitialized, side-stepping the initialization problem. In ιDOT, $\{a = \mathsf{null}\}$ is syntax for an uninitialized field; null is neither a literal nor a term. Definitions in a constructor (i.e. the d in $\kappa(z : Q, z_1\colon U)\,\{d\}$), can only have field definitions of the form $\{a = \mathsf{null}\}$. Type member definitions ($\{A = T\}$) in constructors and objects in ιDOT are the same as those in κDOT, which in turn are the same as in WadlerFest DOT objects.

3.2 Operational Semantics

The operational semantics is expressed using an abstract machine whose transitions are defined in Figure 7. A configuration $\langle t;\ Fs;\ \Sigma\rangle$ consists of the focus of execution t, a frame stack Fs representing the current evaluation context, and a heap Σ. Heaps and frame stacks are described below.

Heap definitions are lists of type definitions, null fields, or non-null fields. A type definition $\{A = T\}$ declares the type label A as an alias for the type T. *Null fields* $\{a = \mathsf{null}\}$ are used to represent an uninitialized field, and *non-null fields* $\{a = y\}$ represent a field that contains the location y. Objects $\nu\,(z\colon T)\,hd$ are a type annotation together with a heap definition. The type annotation T can refer to the recursive this variable z. *Heap items* are objects or literals (functions/constructors), and a *heap* is a map from locations to heap items. In a heap binding of an object $y = \nu\,(z\colon T)\,hd$, the

$$\frac{y = v\,(z\colon T) \ldots \left\{a = \boxed{y'}\right\} \ldots \in \Sigma}{\langle y.a;\ Fs;\ \Sigma\rangle \longmapsto \left\langle \boxed{y'}\ ;\ Fs;\ \Sigma\right\rangle} \quad \text{(Project)}$$

$$\frac{y = v\,(z\colon T) \ldots \{a = \mathsf{null}\} \ldots \in \Sigma \vee y = v\,(z\colon T) \ldots \{a = y'\} \ldots \in \Sigma \qquad \Sigma' = \Sigma\,[y = v\,(z\colon T) \ldots \{a = y_1\} \ldots]}{\langle y.a := y_1;\ Fs;\ \Sigma\rangle \longmapsto \langle y_1;\ Fs;\ \Sigma'\rangle} \quad \text{(Assignment)}$$

$$\frac{y = \lambda\,(z\colon T).t \in \Sigma}{\langle y\,y_1;\ Fs;\ \Sigma\rangle \longmapsto \langle [y_1/z]\,t;\ Fs;\ \Sigma\rangle} \quad \text{(Application)}$$

$$\frac{\overrightarrow{y_2} = \overrightarrow{y, y_1} \qquad \overrightarrow{z_2} = \overrightarrow{z, z_1} \qquad k = \kappa(\overrightarrow{z\colon T}, z_1\colon U)\,\{d\}\ t \in \Sigma}{\langle \mathsf{new}\ k\,(\overrightarrow{y})\,;\ Fs;\ \Sigma\rangle \longmapsto \left\langle [\overrightarrow{y_2/z_2}]t;\ \mathsf{return}\ y_1 :: Fs;\ \Sigma,\ y_1 = v\left(z_1\colon [\overrightarrow{y/z}]U\right)\ [\overrightarrow{y_2/z_2}]d\right\rangle} \quad \text{(New)}$$

$$\langle y_1;\ \mathsf{return}\ y :: Fs;\ \Sigma\rangle \longmapsto \langle y;\ Fs;\ \Sigma\rangle \quad \text{(Return)}$$

$$\langle y;\ \mathsf{let}\ z : T = \square\ \mathsf{in}\ t :: Fs;\ \Sigma\rangle \longmapsto \langle [y/z]\,t;\ Fs;\ \Sigma\rangle \quad \text{(Let-Loc)}$$

$$\langle \mathsf{let}\ z : T = l\ \mathsf{in}\ t;\ Fs;\ \Sigma\rangle \longmapsto \langle [y/z]\,t;\ Fs;\ \Sigma,\ y = l\rangle \quad \text{(Let-Lit)}$$

$$\langle \mathsf{let}\ z : T = t\ \mathsf{in}\ u;\ Fs;\ \Sigma\rangle \longmapsto \langle t;\ \mathsf{let}\ z : T = \square\ \mathsf{in}\ u :: Fs;\ \Sigma\rangle \quad \text{(Let-Push)}$$

Fig. 7. Operational Semantics for ιDOT

definitions in *hd* can refer to any location in the heap including y, and the type annotation T can refer to any location in the heap other than y.

Let frames represent the continuation of executing let bindings of terms and *return frames* represent the continuation of constructor calls. A *frame stack* represents an evaluation context.

The operational semantics for ιDOT is similar to that of κDOT. The main difference is that field reads return variables rather than arbitrary terms. In addition, field assignment replaces a null or a location instead of replacing an arbitrary term.

In ιDOT, we do not allow reading a null field as a machine operation, so a variable can never refer to null. This is different from Scala, where variables can refer to null. Amin and Tate [2016] show how this can trick the Scala type system into believing null is an object containing bad bounds, making the type system unsound. We discuss ιDOT adaptations of their example in Section 7.1.

3.3 Type System

The typing rules for ιDOT are very similar to those of κDOT and other DOT calculi. However, unlike other DOT calculi, the ιDOT type system is not sound on its own. The unsoundness comes from the fact that the type system does not prevent attempts to read a null field, and thus the abstract machine can get stuck without returning an answer.

The typing rules for ιDOT are described in Figure 8 (Heap Definition Typing), Figure 9 (Term Typing), Figure 10 (Literal Typing), and Figure 11 (Subtyping). The few differences from κDOT are highlighted with a shaded background. Some of the following descriptions of the rules are adapted from the paper by Rapoport et al. [2017].

The Figure 8 (Heap Definition Typing in ιDOT) rules are used for typing both plain definitions (*d*) in the (K-I) rule and heap definitions (*hd*) in Definition 4.1 (Heap Correspondence). The (Def-Trm)

$$\Gamma \vdash \{A = T\} : \{A : T..T\} \quad \text{(Def-Typ)}$$

$$\frac{\Gamma \vdash y : T}{\Gamma \vdash \{a = y\} : \{a : T..T\}} \quad \text{(Def-Trm)}$$

$$\Gamma \vdash \{a = \texttt{null}\} : \{a : T..T\} \quad \text{(Def-Trm-Null)}$$

$$\frac{\Gamma \vdash d_1 : T_1 \qquad \Gamma \vdash d_2 : T_2 \qquad \mathsf{dom}(d_1), \mathsf{dom}(d_2) \text{ disjoint}}{\Gamma \vdash d_1 \wedge d_2 : T \wedge U} \quad \text{(And-Def)}$$

Fig. 8. Heap Definition Typing in ιDOT

rule is only used for heap definitions. Note that the rules imply that type definitions are initially typed with equal upper and lower bounds and field definitions are initially typed with equal getter and setter types. This implies that constructor calls and heap objects are also typed with equal bounds, but the bounds can later be relaxed via subtyping.

The (Var), (Rec-I), (Rec-E), and (And-I) rules are used to type variables. The (All-E), (K-E), ({}-E), (:=-I) rules are used to type the four other basic types of terms, applications, constructor calls, field reads, and field assignments. The (Let) and (Lit-I) rules are used to type let and literal bindings. For literal bindings, the (All-I) and (K-I) rules in Figure 10 are used to type the literals themselves. The subsumption rule (Sub) is used to type terms with supertypes of their more precise types.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{(Var)} \qquad \frac{\Gamma \vdash x : \forall(z : T)U \qquad \Gamma \vdash x_1 : T}{\Gamma \vdash x\, x_1 : [x_1/z]U} \quad \text{(All-E)}$$

$$\frac{\Gamma \vdash x : \{a : T..U\}}{\Gamma \vdash x.a : U} \quad \text{({}-E)} \qquad \frac{\Gamma \vdash k : K(\overrightarrow{z : \mathit{i}\ T}, z_1 : U) \qquad \Gamma \vdash \overrightarrow{x : T}}{\Gamma \vdash \mathsf{new}\ k\,(\vec{x}) : \mu(z_1 : \overrightarrow{[x/z]}U)} \quad \text{(K-E)}$$

$$\frac{\Gamma \vdash x : \{a : T..U\} \qquad \Gamma \vdash x_1 : T}{\Gamma \vdash x.a := x_1 : U} \quad \text{(:=-I)}$$

$$\frac{\Gamma \vdash t : T \qquad x \notin \mathsf{fv}(U) \qquad \Gamma, x : T \vdash u : U}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : U} \quad \text{(Let)}$$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash x : \mu(x : T)} \quad \text{(Rec-I)}$$

$$\frac{\Gamma \vdash x : T \qquad \Gamma \vdash x : U}{\Gamma \vdash x : T \wedge U} \quad \text{(And-I)}$$

$$\frac{\Gamma \vdash x : \mu(z : T)}{\Gamma \vdash x : [x/z]T} \quad \text{(Rec-E)}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash T <: U}{\Gamma \vdash t : U} \quad \text{(Sub)}$$

$$\frac{\Gamma \vdash l : T \qquad x \notin \mathsf{fv}(U) \qquad \Gamma, x : T \vdash u : U}{\Gamma \vdash \mathsf{let}\ x = l\ \mathsf{in}\ u : U} \quad \text{(Lit-I)}$$

Fig. 9. Term Typing in ιDOT

The rules in Figure 11 are the standard DOT subtyping rules, with the (Fld-<:-Fld) rule taken from κDOT. The (Top) and (Bot) rules establish ⊤ and ⊥ as the top and bottom of the subtyping lattice, and the (Refl) and (Trans) rules establish the reflexivity and transitivity of the subtyping

$$\frac{\Gamma, x : T \vdash t : U \qquad x \notin \mathsf{fv}(T)}{\Gamma \vdash \lambda(x : T).t : \forall(x : T)U} \tag{All-I}$$

$$\frac{\Gamma, \overrightarrow{x : T}, x_1 : U \vdash d : U \qquad \overrightarrow{x} \notin \mathsf{fv}\left(\overrightarrow{T}\right) \qquad \Gamma, \overrightarrow{x : T}, x_1 : U \vdash t : T'}{\Gamma \vdash \kappa(\overrightarrow{x : \ i \ T}, x_1 : U)\{d\}\, t : K(\overrightarrow{x : \ i \ T}, x_1 : U)} \tag{K-I}$$

Fig. 10. Literal Typing in ιDOT

$$\Gamma \vdash T <: \top \quad (\textsc{Top}) \qquad \Gamma \vdash \bot <: T \quad (\textsc{Bot}) \qquad \Gamma \vdash T <: T \quad (\textsc{Refl})$$

$$\Gamma \vdash T \wedge U <: T \quad (\textsc{And}_1\text{-}<:) \qquad \Gamma \vdash T \wedge U <: U \quad (\textsc{And}_2\text{-}<:)$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash S <: U}{\Gamma \vdash S <: T \wedge U} \quad (<:\text{-}\textsc{And})$$

$$\frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A} \quad (<:\text{-}\textsc{Sel}) \qquad \frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash x.A <: T} \quad (\textsc{Sel}\text{-}<:)$$

$$\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash \{a : T_1..U_1\} <: \{a : T_2..U_2\}} \quad (\textsc{Fld}\text{-}<:\text{-}\textsc{Fld})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash \{A : S_1..T_1\} <: \{A : S_2..T_2\}} \quad (\textsc{Typ}\text{-}<:\text{-}\textsc{Typ})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, x : S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(x : S_1)T_1 <: \forall(x : S_2)T_2} \quad (\textsc{All}\text{-}<:\text{-}\textsc{All})$$

$$\frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \quad (\textsc{Trans})$$

Fig. 11. Subtyping in ιDOT

relation. Basic subtyping of intersection types of intersection types are supported through the ($\textsc{And}_1$-<:), ($\textsc{And}_2$-<:), and (<:-And) rules. Field, type member, and dependent function typing are contravariant in the setter type, lower bound, and parameter type, and covariant in the getter type, upper bound, and result type by the (Fld-<:-Fld), (Typ-<:-Typ), and (All-<:-All) rules respectively. The (<:-Sel) and (Sel-<:) rules define subtyping rules between a path-dependent type $x.A$ and bounds on the type member A. Note that the (<:-Sel) and (Sel-<:) rules use typing for variables (i.e. paths) as a premise, so subtyping and term typing depend on each other.

3.4 Initialization Invariants

In the FBC system, references to items in the heap are given an initialization type which can be one of `free` or `committed`[5]. References to objects that can potentially lead to a null reference via transitive field reads are given the type `free`. References to objects which are fully initialized, i.e. objects which do not lead to a null via transitive field reads, are given the type `committed`.

Given our subheap formulation of FBC, we need to ensure the following.

(1) `Free` objects are not reachable (via field reads) from `committed` objects.
(2) `Committed` objects are fully initialized.

For invariant 1, we ensure the following.

(3) A reference to a `free` object is never assigned to a field of a `committed` object.
(4) A reference to a `free` object is allowed to be assigned to a field of a `free` object only if both objects are in the top-most `free` subheap.
(5) Function bodies do not refer to objects that are part of a `free` subheap.

For invariant 2 we ensure the following.

(6) A constructor initializes all fields of the this/self variable of the constructor.
(7) All fields of all objects in a `free` subheap are initialized before the subheap is promoted to be part of the `committed` heap.

Invariant 5 deserves some explanation. Suppose $y_{\texttt{comm}}$ is a reference to an object in the `committed` subheap and $y_{\texttt{free1}}$ and $y_{\texttt{free2}}$ are references to objects in (possibly different) `free` subheaps. We want to prevent function bodies from making an assignment of the form $y_{\texttt{comm}}.a := y_{\texttt{free1}}$ because calling the function before the object referred to by $y_{\texttt{free1}}$ is promoted to be part of the `committed` heap would violate invariant 3. We additionally want to prevent function bodies from making assignments of the form $y_{\texttt{free1}}.a := y_{\texttt{free2}}$, because if the function is called after $y_{\texttt{free1}}$ becomes part of the `committed` subheap, but $y_{\texttt{free2}}$ is still `free`, we would again violate invariant 3. The above is achieved by disallowing variables of initialization type `free` from appearing in function bodies[6], i.e. variables occurring free (unbound) in functions and constructors are only bound to variables of initialization type `committed`. Functions are allowed to call constructors which allocate new `free` subheaps, but functions cannot modify previously allocated `free` subheaps.

ιDOT has an effect system and an initialization system based on FBC and together they consist of the following four judgments. We will describe the judgments in detail in Sections 3.5 and 3.6.

- $\vdash_e t : E$ says that the term t has the effects E. Here effects track which fields will be initialized when t is executed, and the effect judgment will be used in constructor bodies to ensure invariant 6 and in typing our abstract machine to ensure invariant 7.
- $\Gamma; \Delta \vdash_i t : i$ states that in the typing context Γ and initialization context Δ, the term t has initialization state i. This means that t is well-typed according to the initialization system, i.e. it only does `free`-to-`free` assignments between variables in Δ or newly allocated `free` objects. Additionally, if t evaluates to a location y, the location will be in a `free` subheap or the `committed` subheap depending on i. This judgment is used inside constructor bodies to allow assigning to the this/self variable while ensuring that invariant 3 is satisfied.
- $\Gamma; \Delta \vdash_c t$ says that the term t does not contain free (i.e. unbound) variables that are qualified as `free` in Δ, does not try to allocate objects to the top-most `free` subheap, and has initialization type `committed`. For every free (i.e. unbound) variable x in the term t, $\Delta(x) = \texttt{committed}$, constructor calls in t allocate new `free` subheaps and evaluate to `committed` references, and

[5]The system of Summers and Mueller [2011b] additionally features `unclassified` as an initialization type, a supertype of `free` and `committed`. It is useful when the type system features nullable types and field reads on `free` types are allowed. Our Coq mechanization does support `unclassified`, which is used in some of the extensions discussed in Section 6.
[6]We do allow variables of type `free` to appear in type annotations and in type member definitions. This allows functions to have types dependent on such variables.

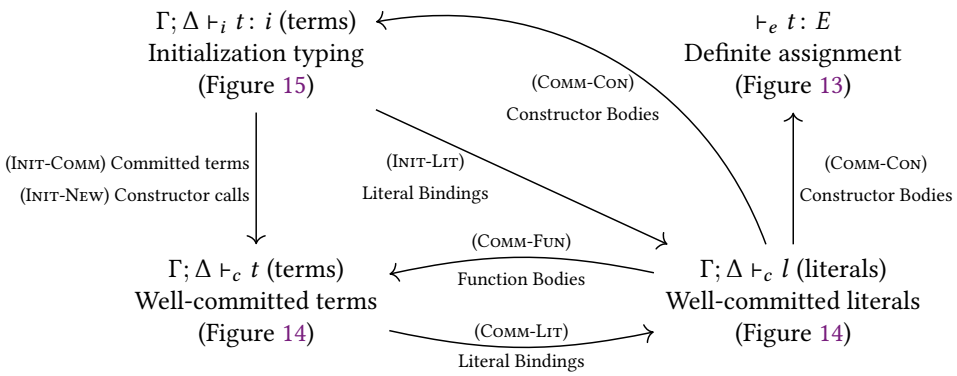


Fig. 12. Dependency of ιDOT typing judgments

$$\frac{}{\vdash_e t : \emptyset}\ (\textsc{Ignore-Eff}) \qquad \frac{}{\vdash_e x.a := x_1 : \{(x, a)\}}\ (\textsc{Asgn-Eff})$$

$$\frac{x \notin \mathsf{fv}(E) \qquad \vdash_e u : E}{\vdash_e \mathsf{let}\ x = l\ \mathsf{in}\ u : E}\ (\textsc{Lit-Eff}) \qquad \frac{\vdash_e t : E_1 \qquad x \notin \mathsf{fv}(E_2) \qquad \vdash_e u : E_2}{\vdash_e \mathsf{let}\ x = t\ \mathsf{in}\ u : E_1 \cup E_2}\ (\textsc{Let-Eff})$$

Fig. 13. Definite Assignment in ιDOT

if t evaluates to a location y, y will have initialization state `committed`. This judgment will be used to type function bodies to ensure invariant 5.

- $\Gamma; \Delta \vdash_c l$ states that the literal l is safe to be allocated on the `committed` subheap. For every free (i.e. unbound) variable x in l, $\Delta(x) =$ `committed`. Therefore, the function or constructor l does not perform writes on variables that are `free` in Δ unless they are passed as arguments.

Since terms can let-bind functions and constructors, functions can be let-bound in constructor bodies, and constructors can be let-bound in function bodies, the above judgments are interdependent. Figure 12 shows a dependency graph of the above judgments.

We will call assignments of the form $y_{\texttt{free1}}.a := y_{\texttt{free2}}$ `free-to-free` *assignments*. Suppose $y_{\texttt{comm2}}$ is a reference to an `committed` object. Assignments of the form $y_{\texttt{comm}}.a := y_{\texttt{comm2}}$, which we call `committed-to-committed` *assignments*, are always safe. `Committed-to-committed` assignments do not cause objects in the `committed` subheap to become less initialized (invariant 2).

3.5 Effect System

The effect system is used to conservatively track definite assignments. Effects are sets of pairs (x, a) where x is a reference to an object and a is a field label. If a term t has the effect (x, a), then evaluating t will definitely assign a value to field a of x. If a constructor is allocating an object with fields $\{a_1, \ldots, a_n\}$, the initialization system requires the constructor body to have the effects $\{(\mathsf{this}, a_1), \ldots, (\mathsf{this}, a_n)\}$. Figure 13 shows the effect system in ιDOT. The (Ignore-Eff) rule conservatively gives at least the empty set of definite assignments to every term. The (Asgn-Eff) rule tracks single assignments. The (Lit-Eff) rule is used to track assignments made by the body of literal bindings. The (Let-Eff) rule is used to group together the effects of t and u in let $x = t$ in u.

The effect system plays a more central role in the small-step setting of ιDOT than in the original big-step safety proof of FBC by Summers and Mueller. In our safety proof, we will extend the effect system to configurations of our abstract machine (Definitions 4.7 to 4.9 and Figure 16). For each `free` subheap, we aggregate all the required effects of all objects in the subheap and track whether they have been satisfied after each step of execution. If the required effects of the subheap have been satisfied and we have nothing left to execute, then every field in the subheap has been initialized, and thus it will be safe to promote the subheap to become part of the `committed` subheap.

3.6 Initialization System

The core problem that the ιDOT initialization system tries to solve is initializing cyclic data-structures. To create a cycle, we need to allow writes of the form $x_1.a := x_2$, where both x_1 and x_2 refer to objects that have some uninitialized fields. But this is unsafe if the object x_1 will be considered fully initialized before the object x_2. In our subheap formulation of FBC, this happens if x_1 refers to an object in a higher `free` subheap than the object referred to by x_2. Since the x_1 object would get promoted to being `committed` before the x_2 object, we could read the field $x_1.a$ and treat the read address as `committed` although the object referred to by x_2 is still in a `free` subheap.

In our initialization system, preventing this manifests itself in how we type constructors. A constructor body can refer both to variables in its enclosing scope, as well as to explicit parameters of the constructor. We want to prevent a constructor from writing to a variable x in its enclosing scope if x has initialization type `free` since the object referred to by x may get promoted to the `committed` heap before the constructor is called. If the constructor assumes x is `free`, it is allowed to assign `free` locations (such as the this/self variable) to x's fields. But once x is promoted, these field updates are unsafe. We also want to prevent a constructor from assigning x to fields of the this/self variable, since we may try to promote the this/self variable to be part of the `committed` heap before x, violating the invariant that there are no references from the committed subheap to free subheaps (invariant 1). However, the above restriction does not mean that constructors can never assign the this/self variable to fields of objects in `free` subheaps. References to `free` objects can be used as constructor arguments. When we call a constructor `new` $k\,(x)$ with x `free`, the constructed object will be allocated in the same `free` subheap as x and the constructed object will be promoted to `committed` at the same time as the object referred to by x.

The judgements of the initialization system ($\Gamma; \Delta \vdash_i t : i, \Gamma; \Delta \vdash_c t$, and $\Gamma; \Delta \vdash_c l$) use two contexts, a typing context Γ to lookup the types of constructors, and an initialization context Δ to lookup the initialization state of variables. In our subheap formulation, the typing context gives types to all items in the heap, but the initialization context only gives initialization types to items in the `committed` subheap and the top-most `free` subheap under consideration. So we have $(x : \texttt{free}) \in \Delta$ only if x refers to an object that belongs to the top-most `free` subheap, and $(x : \texttt{committed}) \in \Delta$ if x refers to an item in the `committed` subheap. This prevents the initialization system from allowing `free`-to-`free` assignments between objects that belong to distinct `free` subheaps (invariant 4).

The initialization system gives initialization states to variables, terms, and literals. Literals are always allocated on the `committed` subheap, so are always given the initialization state `committed`.

We use similar notation $\vdash_c t$ and $\vdash_c l$ to signal that free (i.e. unbound) variables in t and l are `committed`, but the judgments have different uses. $\vdash_c t$ is used to restrict the field writes and constructor calls in function bodies, whereas $\vdash_c l$ is used in literal bindings.

The rules for the initialization system are shown in Figures 14 and 15 and are described below.

3.6.1 Committed Terms and Literals. The judgments of the form $\Gamma; \Delta \vdash_c t$ in Figure 14 are used to restrict terms from performing writes to variables that are `free` or performing new allocations on existing `free` subheaps. The (Comm-Var) rule restricts the variables used in the judgment

$$\frac{\Delta(x) = \texttt{committed}}{\Gamma; \Delta \vdash_c x} \text{(Comm-Var)}$$

$$\frac{\Gamma; \Delta \vdash_c x \qquad \Gamma; \Delta \vdash_c \overrightarrow{x}}{\Gamma \vdash k : K(\overrightarrow{z : \texttt{committed}\, T}, z_1 : T)} \text{(Comm-New)}$$

$$\frac{\Gamma; \Delta \vdash_c x \qquad \Gamma; \Delta \vdash_c x_1}{\Gamma; \Delta \vdash_c x\, x_1} \text{(Comm-App)}$$

$$\frac{\Gamma; \Delta \vdash_c t \qquad \Gamma, x : T; \Delta, x : \texttt{committed} \vdash_c u}{\Gamma; \Delta \vdash_c \texttt{let}\, x : T = t\, \texttt{in}\, u} \text{(Comm-Let)}$$

$$\frac{\Gamma; \Delta \vdash_c x}{\Gamma; \Delta \vdash_c x.a} \text{(Comm-Read)}$$

$$\frac{\Gamma; \Delta \vdash_c l \qquad \Gamma, x : T; \Delta, x : \texttt{committed} \vdash_c u}{\Gamma; \Delta \vdash_c \texttt{let}\, x : T = l\, \texttt{in}\, u} \text{(Comm-Lit)}$$

$$\frac{\Gamma; \Delta \vdash_c x \qquad \Gamma; \Delta \vdash_c x_1}{\Gamma; \Delta \vdash_c x.a := x_1} \text{(Comm-Asn)}$$

$$\frac{\Gamma, z : T; \Delta, z : \texttt{committed} \vdash_c t}{\Gamma; \Delta \vdash_c \lambda(z : T).t} \text{(Comm-Fun)}$$

$$\frac{\vdash_e t : \texttt{eff}\,(x, d) \qquad \Gamma, \overrightarrow{z : T}, z_1 : U; \Delta|^{\texttt{free}}_{\emptyset}, \overrightarrow{z : i}, z_1 : \texttt{free} \vdash_i t : i}{\Gamma; \Delta \vdash_c \kappa(\overrightarrow{z : i\, T}, z_1 : U)\, \{d\}\, t} \text{(Comm-Con)}$$

Fig. 14. Committed Terms and Literals in ιDOT

to be `committed`. The (Comm-App) rule ensures that arguments passed to functions are always `committed`. Similarly, the (Comm-New) rule ensures that constructor applications in this judgment also use only `committed` arguments. The rule allocates a new `free` subheap when the constructor is called which will be merged into the `committed` subheap when the constructor returns. The (Comm-Read) rule ensures that field reads are only performed on `committed` variables. The (Comm-Asn) rule ensures that only `committed` variables are written to fields of `committed` variables. The (Comm-Lit) rule ensures that we let bind well-typed literals in this judgment. The (Comm-Let) rule lifts the judgment to the bound term and body of a let binding.

The judgments of the form $\Gamma; \Delta \vdash_c l$ in Figure 14 ensure that bodies of constructor and function literals are well-typed according to the initialization system. The (Comm-Fun) rule ensures that the body t of a function accesses only `committed` variables. This is required because we do not want to allow writing an uninitialized object to the field of a `committed` object, breaking the invariant that `committed` objects are transitively initialized. The premise of (Comm-Fun) assumes that the parameter z is `committed`, which is guaranteed by (Comm-App).

The (Comm-Con) rule deserves special attention since it is where all relations of the initialization system interact together. The notation $\texttt{eff}\,(x, d)$ means the set of pairs (x, a) such that $\{a = \texttt{null}\} \in d$. The premise $\vdash_e t : \texttt{eff}\,(x, d)$ ensures that the constructor body definitely assigns to every field of the object it will construct. Judgments of the form $\Gamma; \Delta \vdash_i t : i$ will be described in more detail below in Section 3.6.2, but they allow allocations on the topmost `free` subheap and writes to `free` terms such as z_1, the this/self variable. To ensure that these additional write operations remain safe, we prevent the constructor body from referring to `free` variables in its enclosing scope by removing them from the initialization context Δ using the notation $\Delta|^{\texttt{free}}_{\emptyset}$. In general, we write $\Delta|^{\texttt{free}}_{vs}$ for the restriction of Δ that contains all `committed` variables from Δ but only those `free` variables from Δ that are also in vs. The constructor body is typed with the initialization context $\Delta|^{\texttt{free}}_{\emptyset}$ extended with initialization types for the constructor parameters and the this/self variable. Since these rules restrict literals (constructors and lambda functions) to refer only to variables from their enclosing scope that are `committed`, these literals can be allocated directly on the `committed` subheap.

$$\frac{\Delta(x) = i}{\Gamma; \Delta \vdash_i x : i} \text{ (Init-Var)} \qquad \frac{\Gamma; \Delta \vdash_c t}{\Gamma; \Delta \vdash_i t : \texttt{committed}} \text{ (Init-Comm)}$$

$$\frac{\Gamma; \Delta \vdash_c k \qquad \Gamma; \Delta \vdash_i \overrightarrow{x : i} \qquad \Gamma \vdash k : K(\overrightarrow{z : i\, T}, z_1 : U)}{\Gamma; \Delta \vdash_i \mathsf{new}\, k\, (\overrightarrow{x}) : \texttt{free}} \text{ (Init-New)}$$

$$\frac{\Gamma; \Delta \vdash_i x : \texttt{free} \qquad \Gamma; \Delta \vdash_i x_1 : \texttt{free}}{\Gamma; \Delta \vdash_i x.a := x_1 : \texttt{free}} \text{ (Init-Asn-Free)}$$

$$\frac{\Gamma; \Delta \vdash_i x : i \qquad \Gamma; \Delta \vdash_i x_1 : \texttt{committed}}{\Gamma; \Delta \vdash_i x.a := x_1 : \texttt{committed}} \text{ (Init-Asn-Comm)}$$

$$\frac{\Gamma; \Delta \vdash_i t : i \qquad \Gamma, x : T; \Delta, x : i \vdash_i u : i'}{\Gamma; \Delta \vdash_i \mathsf{let}\, x : T = t\, \mathsf{in}\, u : i'} \text{ (Init-Let)}$$

$$\frac{\Gamma; \Delta \vdash_c l \qquad \Gamma, x : T; \Delta, x : \texttt{committed} \vdash_i u : i'}{\Gamma; \Delta \vdash_i \mathsf{let}\, x : T = l\, \mathsf{in}\, u : i'} \text{ (Init-Lit)}$$

Fig. 15. Initialization for Terms in ιDOT

3.6.2 Initialization State of Terms. The rules in Figure 15 qualify terms with an initialization qualifier. When $\Gamma; \Delta \vdash_i t : i$ holds, if the term t evaluates to a location y, then y will have initialization type i. These rules are applied inside a constructor body. In a constructor body, in addition to all operations allowed by `committed` terms via the (Init-Comm) rule, we allow additional operations.

- The (Init-New) rule allows creating new `free` objects. In our subheap formulation, the constructor allocates the object in the same subheap as the `free` variables in Δ, ensuring that the constructed object will become `committed` at the same time as the `free` objects in Δ.
- The (Init-Asn-Comm) rule allows assigning `committed` locations to fields of `free` objects.
- The (Init-Asn-Free) rule allows assigning `free` locations to fields of `free` objects.

Notice that $\Gamma; \Delta \vdash_c t$ is more restrictive than $\Gamma; \Delta \vdash_i t : \texttt{committed}$. For example, we have $\Gamma; x_1 : \texttt{free}, x_2 : \texttt{committed} \vdash_i \mathsf{let}\, y = x_1\, \mathsf{in}\, x_2 : \texttt{committed}$, but $\Gamma; x_1 : \texttt{free}, x_2 : \texttt{committed} \not\vdash_c \mathsf{let}\, y = x_1\, \mathsf{in}\, x_2$ since x_1 is `free`.

We designed the initialization system to be mostly independent of the type system; only the (Comm-New) and (Init-New) rules use the type system. This was so that we could understand and reason about the initialization system mostly free of the bad bounds problem of DOT calculi.

4 CONFIGURATION TYPING

We prove the type and initialization safety by proving progress and preservation lemmas in the style of Wright and Felleisen [1994]. Since the operational semantics of ιDOT is defined in terms of runtime configurations, we must extend the type and initialization system to runtime configurations.

Configuration typing is a list of properties that are satisfied by an empty configuration of a well-typed program, and that continue to hold as the program executes. In κDOT, configuration typing related a context, a configuration, and a type; it was of the form $\Gamma \vdash \langle t; Fs; \Sigma \rangle : U$. Since a configuration represents an overall program, with t filling in the hole in the evaluation context represented by Fs and running with the heap Σ, configuration typing gives the overall configuration a type U. This means that if the program terminates, the answer would be of type U.

Configuration typing for ιDOT is more complex than κDOT since we need more precise invariants about the heap. Field writes have an effect on initialization and field reads require fields to contain non-null locations, i.e they require certain effect obligations to have been fulfilled. Thus we need to

define initialization effect invariants and show that they are preserved, in addition to showing that typing invariants are preserved. For this, we introduce a notion of effects and effect contexts.

Effect contexts, $\mathcal{E}$, will map locations to sets of field names, and will track all the fields that need to be initialized before we can consider the object at the location to be null-free.

ιDOT's configuration typing has the form $\Gamma;\ \Delta\ ;\ \mathcal{E}s\ \vdash \langle t;\ Fs;\ \Sigma\rangle :\ \Big(\ Es\ , U\Big)$, where Γ is a typing context, Δ is an initialization context, $\mathcal{E}s$ is a stack of effect contexts, and Es is a stack of effects. Like in κDOT, U is the type of the eventual answer if the program terminates. The additional invariants are discussed in more detail in Section 4.5, but we describe the components now. The initialization context Δ has the same domain as the heap Σ and tracks the initialization state of the heap items in Σ. The stack of effect contexts $\mathcal{E}s$ serves two purposes. $\mathcal{E}s$ corresponds to the stack of `free` subheaps in our subheap formulation of FBC. The domain of each $\mathcal{E} \in \mathcal{E}s$ is the set of heap locations in the corresponding subheap. Mappings in each effect context of $\mathcal{E}s$ track which object fields in the heap Σ are still uninitialized. The stack of effects Es conservatively tracks the effects of t and Fs, and has the same length as $\mathcal{E}s$. If $\mathcal{E}s = \mathcal{E}_1 :: \ldots :: \mathcal{E}_n$ and $Es = E_1 :: \ldots :: E_n$, then for every $i \in \{1, \ldots, n\}$, $\{(x, a) \mid a \in \mathcal{E}_i(x)\} \subset E_i$ and objects in the subheap corresponding to $\mathcal{E}_i$ can be safely promoted after the effects E_i have been performed. Note that Δ here gives initialization states to the entire heap; subsets of Δ will be used to type the focus of execution t and terms in the frame stack Fs. Promoting a subheap to be part of the `committed` subheap simply updates mappings in Δ from `free` to `committed`.

For a concrete example of how the above notation corresponds to our subheap formulation of FBC, consider the heap configuration of the code fragment in Figure 4, captured immediately after entering the Fruit constructor. This is at a slightly earlier point of execution than Figure 5a, where the Tree constructor was called allocating a new subheap, which in turn called the Fruit constructor allocating an object on the same subheap, but no field assignments have been made, so both t.fruit and f.tree are null. In this heap configuration, the objects in Σ that belong to the top-most `free` subheap are f and t, where the following bindings are present in Σ: $\mathsf{f} = \nu\,(z\colon \mathsf{Fruit})\,\{\mathsf{tree} = \mathsf{null}\}$ $\mathsf{t} = \nu\,(z\colon \mathsf{Tree})\,\{\mathsf{fruit} = \mathsf{null}\}$. At the point of capture, the topmost $\mathcal{E}$ in $\mathcal{E}s$ would contain $\mathsf{f} : \{\mathsf{tree}\}$, $\mathsf{t} : \{\mathsf{fruit}\}$ and the topmost E of the Es would contain $\{(\mathsf{f}, \mathsf{tree}), (\mathsf{t}, \mathsf{fruit})\}$, with $\Delta(\mathsf{f}) = \Delta(\mathsf{t}) = \texttt{free}$. When we get to the same point of execution as Figure 5a, since the field assignments have been performed, E would be empty and $\mathcal{E}$ would be $\mathsf{f} : \emptyset$, $\mathsf{t} : \emptyset$ since no fields of f and t remain uninitialized. Promoting the `free` subheap corresponding to $\mathcal{E}$ to be `committed` would then involve updating f and t to be `committed` in Δ as well as popping the empty $\mathcal{E}$ and E from $\mathcal{E}s$ and Es.

4.1 Heap Correspondence

Heap correspondence extends the type system to items in the heap. It checks that the typing context Γ contains a precise type for each item in the heap, i.e. without applying any subtyping rules at the top level. Heap correspondence in ιDOT is similar to heap correspondence in κDOT [Kabir and Lhoták 2018] and store correspondence in WadlerFest DOT [Amin et al. 2016; Rapoport et al. 2017].

Definition 4.1 (Heap Correspondence). For a context Γ and a heap Σ, we say that Γ *corresponds* to Σ, written $\Gamma \sim \Sigma$, if Γ and Σ have the same domain, and for all $x : T \in \Gamma$ and $x = h \in \Sigma$,

- if $h = \lambda\,(z\colon S)\,.t$, then $\Gamma \vdash h\colon T$ using the (All-I) rule, or
- if $h = \kappa(\overrightarrow{z\colon S}, z_1 : U)\,\{d\}\,t$, then $\Gamma \vdash h\colon T$ using the (K-I) rule, or
- if $h = \nu\,(z\colon U)\,hd$ for some object $\nu\,(z\colon U)\,hd$, then $T = \mu(z\colon U)$ and $\Gamma \vdash hd\colon\ [^x/_z]\,U$.

4.2 Well-Committed and Free Heaps

When encoding the FBC system, we need to split up the heap into the committed heap and a stack of `free` subheaps. The following definitions serve this purpose.

$$\frac{\Gamma \vdash S <: U}{\Gamma; \Delta; \varepsilon :: \varepsilon \vdash \varepsilon : \ (S, i) \Rightarrow \ (\emptyset :: \varepsilon, U)} \quad \text{(Stack Empty)}$$

$$\frac{\begin{array}{c} x \notin \mathsf{fv}(T) \qquad \Gamma, x : S \vdash t : T \\ \vdash_e t : E_1 \qquad \Gamma; (\Delta, x : i_1)|^{\mathsf{free}}_{\mathsf{dom}(\mathcal{E}) \cup \{x\}} \vdash_i t : i_2 \\ \Gamma, \Delta, \mathcal{E} :: \mathcal{E}s \vdash Fs : \ (T, i_2) \Rightarrow \ (E_2 :: Es, U) \end{array}}{\Gamma; \Delta; \mathcal{E} :: \mathcal{E}s \vdash \mathsf{let}\ x = \square\ \mathsf{in}\ t :: Fs : \ (S, i_1) \Rightarrow \ (E_1 \cup E_2 :: Es, U)} \quad \text{(Stack Let)}$$

$$\frac{\begin{array}{c} \Gamma \vdash x : T \qquad x \in \mathsf{dom}(\mathcal{E}) \\ \Gamma, \Delta, \mathcal{E} :: \mathcal{E}s \vdash Fs : \ (T, \mathtt{free}) \Rightarrow \ (E_2 :: Es, U) \end{array}}{\Gamma, \Delta, \mathcal{E} :: \mathcal{E}s \vdash \mathsf{return}\ x :: Fs : \ (S, i) \Rightarrow \ (E_2 :: Es, U)} \quad \text{(Stack Return-Free)}$$

$$\frac{\begin{array}{c} \Gamma \vdash x : T \qquad x \in \mathsf{dom}(\mathcal{E}) \\ \Gamma, \Delta, \mathcal{E}s \vdash Fs : \ (T, \mathtt{committed}) \Rightarrow \ (Es, U) \end{array}}{\Gamma, \Delta, \mathcal{E} :: \mathcal{E}s \vdash \mathsf{return}\ x :: Fs : \ (S, i) \Rightarrow \ (\emptyset :: Es, U)} \quad \text{(Stack Return-Comm)}$$

Fig. 16. Stack Typing in ιDOT

4.2.1 Committed Objects and Heaps. Similar to the heap correspondence, we define a correspondence between the committed part of the heap and the initialization context. This formalizes the invariant that items in the `committed` subheap may never refer to objects in `free` subheaps.

Definition 4.2 (Committed Heap Object). A heap object $v\,(z : T)\,hd$ is *well-committed* under Δ, if it contains no null fields and every field points to a location that is `committed` according to Δ.

Definition 4.3 (Committed Heap Items). A heap item is *well-committed* under Γ, Δ, if it is either a well-committed heap object under Δ or a literal l such that $\Gamma, \Delta \vdash_c l$.

Definition 4.4 (Well-committed Heap). A heap Σ is *well-committed* under Γ, Δ, if Γ, Δ, Σ all have the same domain and whenever $\Delta(x) = \mathtt{committed}$, $\Sigma(x)$ is a *well-committed* heap item under Γ, Δ.

4.2.2 Free Objects and Heaps. We now define a correspondence between the `free` parts of the heap and a stack of effect contexts.

Definition 4.5 (Free Heap Object). An object $y_1 = v\,(z : T)\,hd \in \Sigma$ is a *free heap object* under $\Delta, \mathcal{E}$ if $\Delta(y_1) = \mathtt{free}$, $\mathcal{E}(y_1) = \{a \mid \{a = \mathsf{null}\} \in hd\}$, and for all $\{a = y_2\} \in hd$, either $\Delta(y_2) = \mathtt{free}$ and $y_2 \in \mathsf{dom}(\mathcal{E})$, or $\Delta(y_2) = \mathtt{committed}$.

The above simply states that the non-null fields of the object represented by y_1 point only to objects in the `free` subheap represented by $\mathcal{E}$ or to items in the `committed` subheap. It also states that $\mathcal{E}$ has enough field names to ensure that the fields of y_1 will eventually be assigned.

Definition 4.6 (Free Subheap). An effect context $\mathcal{E}$ *corresponds to a free subheap* of Σ under Δ, if for every $y \in \mathsf{dom}(\mathcal{E})$ there is a $y = v\,(z : T)\,hd \in \Sigma$ that is a free heap object under $\Delta, \mathcal{E}$.

Definition 4.7 (Free Heap Stack). A stack of effect contexts $\mathcal{E}s$ is a *free heap stack* of Σ under Γ, Δ, if the $\mathcal{E}s$ have disjoint domains, each $\mathcal{E}$ corresponds to a free subheap of Σ under Δ, and Σ is *well-committed* under Γ, Δ.

4.3 Stack Typing

The stack typing rules for ιDOT are shown in Figure 16. Stacks represent evaluation contexts, and stack typing judgments are of the form $\Gamma, \ \Delta\ , \ \mathcal{E}s\ \vdash Fs : \ \Big(T, \ i\ \Big) \Rightarrow \ \Big(\ Es\ , U\Big)$. The frame stack Fs

is treated as a function whose parameter is a focus of execution to be provided and which must have type T and initialization type i. When provided such a focus of execution, the whole stack will have type U and will perform the effects and subheap promotions in Es in addition to any effects of the focus of execution. The domain of the top element of $\mathcal{E}s$ represents the `free` variables that the focus of execution may perform field writes on.

The (Stack Empty) rule types the base case of an empty stack. The subtyping premise in this rule allows us to avoid defining subtyping between stacks and between configurations.

The (Stack Let) rule mirrors the (Init-Let) and (Let-Eff) rules and the typing rule for let bindings. It is the only rule that contributes effects to the effect stack: in a judgment $\Gamma, \Delta, \mathcal{E}s \vdash Fs\colon\ (T, i) \Rightarrow (E :: Es, U)$, the E contains the effects from let frames until the next commitment point (commitment points are introduced by (Stack Return-Comm), which is explained subsequently). From $\vdash_e t\colon E_1$, we know that $(y, a) \in E_1$ only if t contains a sub-term of the form $y.a := x$.

The $(\Delta, x : i_1)|^{\mathsf{free}}_{\mathsf{dom}(\mathcal{E})\cup\{x\}}$ premise in (Stack Let) deserves some explanation. In $\mathsf{let}\ x = \square\ \mathsf{in}\ t$, x refers to an unevaluated computation, and t will be evaluated once x is fully evaluated. The initialization system will ensure that if $i_1 = \mathtt{free}$, x will evaluate to a location in the subheap represented by $\mathcal{E}$ and if $i_1 = \mathtt{committed}$, x will evaluate to a location in the `committed` subheap. The term t in $\mathsf{let}\ x = \square\ \mathsf{in}\ t$ must not mutate objects in other `free` subheaps, but it should be allowed to mutate x. Therefore, we check t in the restricted initialization context $(\Delta, x : i_1)|^{\mathsf{free}}_{\mathsf{dom}(\mathcal{E})\cup\{x\}}$ which contains x and the topmost `free` subheap, but removes all other free subheaps from Δ.

The (Stack Return-Comm) and (Stack Return-Free) rules ensure that, after a constructor call, the stack is typed with the type of the allocated object. The premise $x \in \mathsf{dom}\,(\mathcal{E})$ ensures that the returned reference will be in the subheap $\mathcal{E}$. The (Stack Return-Comm) rule represents a commitment point where a subheap represented by $\mathcal{E}$ will be promoted when this frame is popped.

4.4 Effect Correspondence

Effect contexts track the fields that need to be initialized to consider a `free` subheap fully initialized, and effect sets track which fields will be initialized when subterms of the configuration term and the stack are executed. We define effect correspondence to relate the two.

Definition 4.8 (Effect Correspondence). Let $\mathcal{E}s = \mathcal{E}_1 :: \ldots :: \mathcal{E}_n :: \varepsilon$ be an effect context stack and $Es = E_1 :: \ldots :: E_n :: \varepsilon$ be an effect set stack. We say that $\mathcal{E}s$ *corresponds* to Es, denoted $\mathcal{E}s \sim Es$, if for all $i \in \{1, \ldots, n\}$, $\{(x, a) \mid a \in \mathcal{E}_i\,(x)\} \subset E_i$.

$\mathcal{E}\,(x)$ is the set of effects required to definitely assign the fields of the object at x. Therefore, the range of $\mathcal{E}$ contains the effects needed to definitely assign all the fields of all the objects in $\mathcal{E}$. If $\mathcal{E} :: \mathcal{E}s \sim E :: Es$, then E contains all the effects in $\mathcal{E}$ (and possibly more), so performing all the effects in E ensures that all fields in the topmost `free` subheap $\mathcal{E}$ are assigned.

4.5 Typing a Configuration

We finally bring all the preceding definitions together in the concept of configuration typing, which tracks all the runtime invariants necessary for proving type safety.

Definition 4.9 (Configuration Typing). We write $\Gamma; \Delta; \mathcal{E} :: \mathcal{E}s \vdash \langle t;\ Fs;\ \Sigma\rangle :\ (E :: Es, U)$, if

(1) $\Gamma, \Delta, \mathcal{E} :: \mathcal{E}s \vdash Fs\colon\ (T, i) \Rightarrow (E_2 :: Es, U)$, which means that Fs, the continuation for the current focus of execution t, when provided with a term of type T and initialization type i, will perform effects E_2 (in addition to those performed by t) and yield an answer of type U.
(2) Γ is an inert context according to the criteria of Rapoport et al. [2017],
(3) $\Gamma \sim \Sigma$, which means that the heap Σ corresponds to Γ,
(4) $\Gamma \vdash t\colon T$,

(5) $\mathcal{E} :: \mathcal{E}s$ is a free heap stack of Σ under Γ, Δ, which describe the `free` subheaps and the uninitialized fields of Σ,
(6) $\Gamma, \Delta|^{\mathsf{free}}_{\mathsf{dom}(\mathcal{E})} \vdash_i t : i$, which means that t has initialization type i while only using `committed` variables or `free` variables present in $\mathsf{dom}(\mathcal{E})$,
(7) $E = E_1 \cup E_2$,
(8) $\vdash_e t : E_1$, which means that t will definitely perform effects E_1, and
(9) $\mathcal{E} :: \mathcal{E}s \sim E :: Es$, which means E contains all effects in $\mathcal{E}$ and Es contains all effects in $\mathcal{E}s$.

Item 1 is related to all of the type, effect, and initialization systems. Items 2 to 4 are related to the type system. Item 5 is an invariant about the heap. Items 6 to 9 are invariants about effects. Items 5 and 9 together say that the program will definitely initialize all fields in Σ. Inert types and contexts (item 2) are technical criteria for DOT type preservation and progress proofs. Since our focus is on initialization safety, we refer the reader to Rapoport et al. [2017] for discussion of inertness.

5 TYPE AND INITIALIZATION SAFETY

In ιDOT we state the safety of the type system and the initialization system as follows.

Theorem 5.1 (Type and Initialization Safety). *If* $\vdash t : T$ *and* $\vdash_i t :$ `committed`*, then either the initial configuration* $\langle t; \varepsilon; \cdot\rangle$ *diverges or* $\langle t; \varepsilon; \cdot\rangle \longmapsto^* \langle y; \varepsilon; \Sigma\rangle$ *for some answer* $\langle y; \varepsilon; \Sigma\rangle$.

We prove the theorem by extending the progress and preservation lemmas of Kabir and Lhoták [2018] for κDOT, which in turn are an extension of the type safety proof of Rapoport et al. [2017] for WadlerFest DOT. Our safety proof requires additional steps since we have an initialization system in addition to the type system. Firstly, in our progress lemma, we need to take initialization information into account. This is mainly needed to show the safety of field reads: a field read $x.a$ will not get stuck if x is fully initialized. Secondly, our preservation lemma is more complicated since we have to show that both type and initialization invariants are preserved by reduction steps.

Lemma 5.2 (Progress). *If* $\Gamma; \Delta; \mathcal{E}s \vdash c : (Es, U)$*, then either* c *is an answer or* $c \longmapsto c'$ *for some* c'.

Lemma 5.3 (Preservation). *If* $\Gamma; \Delta; \mathcal{E}s \vdash c : (Es, U)$ *and* $c \longmapsto c'$*, then there exist* $\Gamma', \Delta', \mathcal{E}s', Es'$ *such that* $\Gamma{+}{+}\Gamma'; \Delta'; \mathcal{E}s' \vdash c' : (Es', U)$.

Lemma 5.4 (Initial Configuration Well-Typed). *If* $\vdash t : T$ *and* $\vdash_i t :$ `committed`*, then* $\varepsilon, \varepsilon, \varepsilon :: \varepsilon \vdash \langle t; \varepsilon; \cdot\rangle : (\emptyset :: \varepsilon, T)$.

5.1 Preserving Heap and Effect Invariants

The ιDOT safety proof requires several lemmas about preserving heap invariants after the heap is modified. In particular, the configuration typing requires that the stack of effect contexts correspond to a free heap stack. Their proofs all proceed by an induction on the stack of effect contexts.

Lemma 5.5 (Allocating Literals). *If* $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* Γ *and* Δ *and* $\Gamma; \Delta \vdash_c l$*, then for any fresh* y *and any* T*,* $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* $\Sigma, y = l$ *under* $\Gamma, y : T$ *and* $\Delta, y :$ `committed`.

Lemma 5.6 (Assigning to Committed Objects). *If* $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* Γ, Δ *and* $\Delta(y_1) =$ `committed`*,* $\Delta(y_2) =$ `committed`*,* $y_1 = \nu(z : T) \ldots \{a = y_3\} \ldots \in \Sigma$*, then* $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* $\Sigma[y_1 = \nu(z : T) \ldots \{a = y_2\} \ldots]$ *under* Γ, Δ.

Lemma 5.7 (Assigning to Free Objects). *Suppose the following hold.*
- $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* Γ, Δ,
- $y_1 = \nu(z : T) \ldots \{a = null\} \ldots \in \Sigma$ *or* $y_1 = \nu(z : T) \ldots \{a = y_3\} \ldots \in \Sigma$ *for some* y_3*, and*

- $y_2 \in \mathsf{dom}\,(\mathcal{E})$ *or* $\Delta\,(y_2) = \texttt{committed}$.

Then $\mathcal{E}\,[y_1 = (\mathcal{E}\,(y_1) \setminus \{a\})] :: \mathcal{E}s$ *is a free heap stack of* $\Sigma\,[y_1 = \nu\,(z\colon T) \ldots \{a = y_2\} \ldots]$ *under* Γ, Δ.[7]

Lemma 5.8 (Allocating Free Objects). *Suppose*

- $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* Γ, Δ,
- *all fields in hd are null, and*
- $E = \{(y, a) \mid \{a = null\} \in hd\}$.

Then $\mathcal{E}, y = E :: \mathcal{E}s$ *is a free heap stack of* $\Sigma,\ y = (\nu\,(z\colon T)\,hd)$ *under* $\Gamma,\ y\colon T$ *and* $\Delta,\ y\colon \texttt{free}$.

Lemma 5.9 (Allocating a New Free Subheap). *Suppose*

- $\mathcal{E}s$ *is a free heap stack of* Σ *under* Γ, Δ,
- *all fields in hd are null, and*
- $E = \{(y, a) \mid \{a = null\} \in hd\}$.

Then $y = E :: \mathcal{E}s$ *is a free heap stack of* $\Sigma,\ y = (\nu\,(z\colon T)\,hd)$ *under* $\Gamma,\ y\colon T$ *and* $\Delta,\ y\colon \texttt{free}$.

Lemmas 5.8 and 5.9 show that when allocating a new object, y, we have a choice to allocate it in an existing topmost `free` subheap or in a new `free` subheap. Objects are allocated by calling a constructor and if the constructor call was typed with (Init-New), we allocate on the topmost subheap, and if the constructor call was typed with (Comm-New), we allocate on a new subheap.

Lemma 5.10 (Promoting a Free Subheap). *Suppose*

- $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* Γ, Δ,
- *for all* $y \in \mathsf{dom}\,(\mathcal{E})$, $\mathcal{E}\,(y) = \emptyset$, *and*
- $\mathsf{dom}\,(E) = \{\vec{y}\}$.

Then $\mathcal{E}s$ *is a free heap stack of* Σ *under* $\Gamma, \Delta[\overrightarrow{y = \texttt{committed}}]$.

Lemma 5.10 is where much of our infrastructure about effects finally pays off! Subheaps start out by being allocated via Lemma 5.9 and afterwards additional allocations on the same subheap may follow via Lemma 5.8. We then remove all the field names in the effect context $\mathcal{E}$ via Lemma 5.7, which ensures that after assignments, objects in the `free` subheap point only inside the same `free` subheap or to the `committed` heap. Once all the field names in the effect context $\mathcal{E}$ have been removed, i.e. $\mathcal{E}\,(y) = \emptyset$ for all $y \in \mathsf{dom}\,(\mathcal{E})$, every object in this free subheap must be fully initialized. Thus we can promote this heap to be fully `committed`.

5.2 Substitution Lemma for Initialization

Small-step type safety proofs make use of a substitution lemma. The substitution lemma is used to show that when parameters are replaced by values in a function, constructor, or let binding body, the type of the body is preserved. Since ιDOT has a type system, an effect system, and an initialization system, we need substitution lemmas for all of them. Below we state the substitution lemmas for all of the systems so that we may compare and contrast them.

Lemma 5.11 (Substitution Lemma for Effects). *If* $\vdash_e t\colon E$, *then* $\vdash_e [y/x]\,t\colon [y/x]\,E$.

Lemma 5.12 (Substitution Lemma for Typing). *If* $\Gamma_1, x : T, \Gamma_2 \vdash t\colon U$ *and* $\Gamma_1, [y/x]\,\Gamma_2 \vdash y\colon [y/x]\,T$, *then* $\Gamma_1, [y/x]\,\Gamma_2 \vdash [y/x]\,t\colon [y/x]\,U$.

Lemma 5.13 (Substitution Lemma for Initialization). *Suppose the following conditions hold.*

- $\Gamma_1, x : T, \Gamma_2;\ \Delta_1, x : i_1, \Delta_2 \vdash_i t\colon i_2$,
- $\Gamma_1, [y/x]\,\Gamma_2 \vdash y\colon [y/x]\,T$, *and*

[7] In our Coq formulation of this lemma, we require an additional premise that $\Gamma \sim \Sigma$. This is used to ensure that objects in the heap are well-formed. Otherwise, an object can have two uninitialized fields with the same name $\{a = \mathsf{null}\}$, and if we only update the first field, would not be able to remove a from $\mathcal{E}\,(y_1)$.

- $(\Delta_1, \Delta_2)(y) = i_1$.

Then $\Gamma_1, [y/x]\,\Gamma_2;\ \Delta_1, \Delta_2 \vdash_i [y/x]\,t : i_2$.

The substitution lemma for the initialization system in ιDOT takes an interesting form. Since our initialization system depends on the type system, our proof of Lemma 5.13 (Substitution Lemma for Initialization) uses Lemma 5.12 (Substitution Lemma for Typing), and the second condition in Lemma 5.13 is used for this reason.

5.3 Preservation

We prove Lemma 5.3 (Preservation) by an induction on the typing of the focus of execution, with the variable case breaking down into further cases based on the frame stack. Except for the subtyping case which is easily discharged by the induction hypothesis, the cases align with the rules of Figure 7 (Operational Semantics for ιDOT). Below we discuss the cases, focusing on the initialization safety.

The (Project), (Return), (Assignment), and (Let-Push) cases are simpler since they do not make use of the effect or initialization substitution lemmas.

We split the (Return) case into two subcases depending on whether the returned location is `free` or `committed`. We discuss the `committed` case since it makes use of effect correspondence. Inverting stack typing tells us that the stack has the empty effect (i.e. $E_1 = \emptyset$ in configuration typing). By inverting the effect of the focus of execution (a location in this case) we know that the focus of execution also has the empty effect (i.e. $E_2 = \emptyset$ in configuration typing). By effect correspondence, $E_1 \cup E_2 = \emptyset$ over-approximates the effects required by the current free subheap, so we get that for all $y \in \mathrm{dom}(\mathcal{E})$, $\mathcal{E}(y) = \emptyset$. This allows us to use Lemma 5.10 (Promoting a Free Subheap) to promote the free subheap and return the location in the return frame as `committed`.

The (Assignment) case makes use of the assignment heap invariant lemmas (Lemmas 5.6 and 5.7).

For the (Project) case, since we are in a well-typed configuration, the focus of execution $y.a$ must be well-initialized. Inverting this initialization yields that $y.a$ is `committed` and y is `committed`. Since the heap must be well-committed, the object that y points to must be `committed`, so its a field must contain a `committed` location. Hence, we reduce to a location that is `committed`, as required. Since both $y.a$ and y may only have the empty effect, the effect conditions are preserved.

The (Let-Push) case involves only minor inversions of effect and initialization typing.

The rest of the cases involve substitution and make use of substitution lemmas from Section 5.2.

The (Application) case needs to ensure that substituting a committed argument into the body of the function produces a committed term.

The (Let-Loc), and (Let-Lit) cases need to ensure that substituting a concrete variable of the appropriate initialization type produces a term of the initialization expected by the stack.

The (New) rule splits into two cases by inverting initialization typing. In the first case (from (Comm-New) and (Init-New)), an object is allocated into a new subheap; in the second case, we are creating an object in the same subheap. We use the effect substitution lemma to show that the constructor body has the effect of initializing the object and that effect correspondence is preserved, and the initialization substitution lemma to show that the invoked constructor body is well-typed.

6 EXTENSIONS OF ιDOT

6.1 Free Literals Extension

In the above sections of this paper, we described a simple object initialization system. But the system presented can be limiting when encoding Scala constructs in it. In this section we discuss an extension of the ιDOT calculus that gets around some of these limitations.

Literal bindings in the base ιDOT calculus must be typed using the (Init-Lit) rule, whose premise $\Gamma; \Delta \vdash_c l$ prevents function and constructor literals from using variables with initialization type

free from their enclosing scope. This means that inside a constructor body, we cannot define function literals that refer to the this variable of the constructor and then assign that literal to a field of the this variable. Methods in Scala generally need to access the this variable to read or mutate fields of the object on which the method was called. Hence function literals cannot be assigned as analogues of methods while the object is being initialized — method-like fields must be filled with dummy function literals and backpatched after the object is committed à la Landin's knot. This is unlike Scala where methods become available immediately after a constructor returns.

To remedy this, we introduce free literals, which are allocated on a free subheap and may use variables from the same subheap. We add the following rule to the initialization system.

$$\frac{\Gamma; \Delta[\overrightarrow{\mathsf{dom}\,(\Delta) = \mathtt{committed}}] \vdash_c l \qquad \Gamma, x : T; \Delta, x : \mathtt{free} \vdash_i u : i'}{\Gamma; \Delta \vdash_i \mathsf{let}\ x : T = l\ \mathsf{in}\ u : i'} \quad \text{(Init-Lit-Free)}$$

In this rule, the literal l is typed assuming that all variables in Δ are committed, but in the let binding body u, we treat the binding variable x as free. In the body u, the free initialization of x prevents applying x to any arguments, but it does allow assigning x to fields of free objects.

In a constructor body, we can use the new rule to define function literals which use the this variable of the constructor and assign it to a field of the object being constructed. The function literal cannot be called with arguments until its assumption that the object referred to by the this variable is committed becomes true. When the object is fully initialized and becomes committed, since the literal belonged to the same subheap, the literal also becomes committed. At this point, we can read a reference to the function literal from the object and call the function with arguments.

Bindings of the form $x : \mathtt{free}$ in Δ represent heap items in the topmost free subheap. The body of the literal l can use these variables, since l is typed assuming these variables are committed. In our subheap formulation, when $\mathsf{let}\ x : T = l\ \mathsf{in}\ u$ is executed, the literal l is allocated on the topmost free subheap, which is why we give the binding variable an initialization state of free when typing u. The initialization system prevents literals in a free subheap from being called with arguments, but allows assigning references to the literals to be assigned to fields of objects in the same free subheap via the (Init-Asn-Free) rule in Figure 15.

To prove type safety, since we can now allocate literals on free subheaps, we need a definition for *free heap literals* similar to Definition 4.5 (Free Heap Object). We also need to update Definition 4.6 (Free Subheap) to allow for free heap literals. Furthermore, similar to Lemma 5.8 (Allocating Free Objects), we need a lemma for allocating free literals on free subheaps. These definitions are detailed below.

Definition 6.1 (Free Heap Literal). We say that $y_1 = l \in \Sigma$ is a *free heap literal* under $\Gamma, \Delta, \mathcal{E}$ if

- $\Delta\,(y_1) = \mathtt{free}$,
- $\mathcal{E}\,(y_1) = \emptyset$, and
- $\Gamma; \Delta[\overrightarrow{\mathsf{dom}\,(\mathcal{E}) = \mathtt{committed}}] \vdash_c l$.

Definition 6.2 (Free Heap). We say that an effect context $\mathcal{E}$ *corresponds to a free subheap* of Σ under Γ, Δ if, for every $y = E \in \mathcal{E}$, there is either a $y = v\,(z : T)\ hd \in \Sigma$ that is a free heap object under $\Delta, \mathcal{E}$, or there is a $y = l \in \Sigma$ that is a free heap literal under $\Gamma, \Delta, \mathcal{E}$.

Lemma 6.3 (Allocating Free Literals). *Suppose*

- $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* Γ, Δ,
- $\Gamma; \Delta[\overrightarrow{\mathsf{dom}\,(\mathcal{E}) = \mathit{committed}}] \vdash_c l$.

Then $\mathcal{E}, y = \emptyset :: \mathcal{E}s$ *is a free heap stack of* Σ, $y = l$ *under* Γ, $y : T, \Delta$, $y : \mathtt{free}$.

Lastly, we need to update the proof of Lemma 5.10 (Promoting a Free Subheap) to handle free literals and solve extra cases in the initialization substitution and preservation lemmas. The statement of Lemma 5.10 and rest of the soundness proof is unchanged. We have mechanized the type safety proof of this extension by modifying safety proof of the base calculus with the above changes.

6.2 Local Initialization Extension

Another technical limitation is that we may only safely read from an object after it has been promoted to `committed`, which ensures that all objects it points to are transitively initialized. However, often, when initializing objects, it is useful and natural to read a field of another object in the same `free` subheap, and the transitively initialized property is not required (i.e. if we just require $x.a$ itself and never read any of its fields such as $x.a.b$, then we really only need $x.a \neq \mathsf{null}$). We may weaken the requirements for reading from an object by introducing a new initialization type `local` that captures the notion of having non-null fields, but not transitively like `committed`.

We observe that the definite assignment requirements of ιDOT guarantee that all fields of an object returned by a constructor will always be non-null. We modify the (Init-New) rule to the following so that this guarantee is reflected in the initialization system.

$$\frac{\Gamma; \Delta \vdash_c k \qquad \Gamma; \Delta \vdash_i \overrightarrow{x : i} \qquad \Gamma \vdash k : K(\overrightarrow{z : i\, T}, z_1 : U)}{\Gamma; \Delta \vdash_i \mathsf{new}\, k\,(\overrightarrow{x}) : \texttt{local}}$$

An object of initialization type `local` resides in a `free` subheap and its fields have been initialized, but the fields may point to items in the same `free` subheap (or to items in the `committed` subheap). Since `local` objects are just `free` objects with additional guarantees which allow for additional operations listed below, `local` is a subtype of `free`.

To allow field reads on objects which are initialized, but not transitively, we add the initialization type `unclassified`, a supertype of `committed` and `free`, and add the following rule to the initialization system.

$$\frac{\Gamma; \Delta \vdash_i x : \texttt{local}}{\Gamma; \Delta \vdash_i x.a : \texttt{unclassified}} \qquad \text{(Local-Read)}$$

A constructor can assign references to either kind of subheap to its object. Thus, when reading a field, we cannot know if the field refers to an object in the `committed` subheap or an object in the `free` subheap. Thus a field read on a `local` object must be treated as `unclassified`.

Figure 17 shows a Hasse diagram of the subtyping semi-lattice. A consequence of having initialization types which are in a subtyping relationship is that we must now add the following subtyping rule to our initialization typing system.

$$\frac{\Delta(x) = i \qquad i <: j}{\Gamma; \Delta \vdash_i x : j} \qquad \text{(Init-Var-Sub)}$$

With the additional rules in place, we can add the line **val** fruitTree = fruit.tree right after line 27 in Figure 1 and have the program accepted by the initialization system. Previously, fruit would have been `free`, and the field read fruit.tree would have been invalid. But under the new typing rules, fruit is `local` and the field read is now allowed.

With subtyping added to the mix, some of our previous lemmas about initialization types proven using simple case analysis now become (straightforward) induction on the proof of initialization typing. Further, we need to maintain more precise invariants pertaining to `local` objects. To do this, we define what it means for an object to be properly localized, like we did for Definition 4.5 (Free Heap Object).

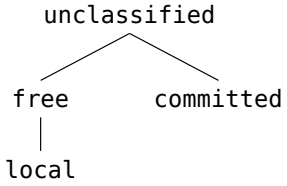


Fig. 17. Subtyping Semi-lattice for Locally Initialized Extension

Definition 6.4 (Localized Heap Item). We say that $y = \nu\,(z\colon T)\,hd \in \Sigma$ is a *localized heap item* under $\Delta, \mathcal{E}$ if $\Delta\,(y) = \texttt{local}$, each field in hd is non-null (there does not exist any $\{a = \mathsf{null}\} \in hd$), and $\mathcal{E}\,(y_1) = \emptyset$, and for all $\{a = y_2\} \in hd$, either $\Delta\,(y_2) = \texttt{free} \vee \Delta\,(y_2) = \texttt{local}$ and $y_2 \in \mathsf{dom}\,(\mathcal{E})$, or $\Delta\,(y_2) = \texttt{committed}$.

Then, we lift the localized heap item condition to entire subheaps like in Definition 4.6 (Free Subheap).

Definition 6.5 (Free Heap). We say that an effect context $\mathcal{E}$ *corresponds to a free subheap* of Σ under Γ, Δ if, for every $y = E \in \mathcal{E}$, there is either a $y = \nu\,(z\colon T)\,hd \in \Sigma$ that is a free heap object *or a localized heap object* under $\Delta, \mathcal{E}$.

The remaining changes to existing lemmas amount to showing that we can safely promote an object from `free` to `local` after returning from its constructor:

Lemma 6.6 (Promoting a Localized Heap Item). *Suppose that*
- $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* Γ, Δ*, and*
- $y \in \mathsf{dom}\,(\mathcal{E})$ *has all its effects fulfilled (i.e.* $\mathcal{E}\,(y) = \emptyset$*)*

Then $\mathcal{E} :: \mathcal{E}s$ *is a free heap stack of* Σ *under* $\Gamma, \Delta[y = \texttt{local}]$.

The main difference between Lemma 6.6 (Promoting a Localized Heap Item) and Lemma 5.10 (Promoting a Free Subheap) is that Lemma 5.10 promotes all objects in the current subheap while Lemma 6.6 promotes only the single object y which we know to have non-null fields. With Lemma 6.6, the rest of the soundness proof is straightforward.

We have mechanized the type safety proof of the Locally Initialized Extension of ιDOT.

7 DISCUSSION

7.1 ιDOT without the Initialization System is Unsound

Amin and Tate [2016] showed that null values make Scala unsound. Figure 18 shows one of their examples adapted to ιDOT.[8] The example would type-check under ιDOT's type system, but not its initialization system. In the example, we use $ty.Low$ as an alias for an object type and $ty.Up$ as an alias for a function type. We define a function *coerce*, which returns its second argument of type $ty.Low$, but ascribed the type $ty.Up$ using the subtyping relation $ty.Low <: lb.M <: ty.M <: ty.Up$, which is derived from the (Sel-<:) and (<:-Sel) rules of DOT. The rest of the code attempts to create the reference bounded, so that it can be passed to coerce to upcast obj to ty.Up.

The kUnsafeObj constructor creates an object with an uninitialized field of type $\{M\colon ty.Low..ty.Up\}$. This field is then read and let bound to bounded and passed to coerce. In DOT and κDOT, fields store terms instead of just references. In a DOT or κDOT version of the program, instead of leaving the field uninitialized, we would assign it a divergent term.

[8] We inline the function upcast of Amin and Tate to simplify the example.

$$
\begin{array}{l}
\mathsf{let}\ ty\colon \{Low\colon \{a\colon \top\}\} \wedge \{Up\colon \forall(x\colon \top)\top\} = \ldots\ \mathsf{in} \\
\mathsf{let}\ coerce\colon \forall(lb\colon \{M\colon ty.Low..ty.Up\})\forall(x\colon ty.Low)ty.Up = \\
\quad \lambda(lb\colon \{M\colon ty.Low..ty.Up\}).\lambda(x\colon \{a\colon \top\}).x\ \mathsf{in} \\
\mathsf{let}\ kUnsafeObj = \kappa(z\colon \{a\colon \{M\colon ty.Low..ty.Up\}\})\,z\ \mathsf{in} \\
\mathsf{let}\ obj = \mathsf{new}\ kUnsafeObj()\ \mathsf{in} \\
\mathsf{let}\ bounded = obj.a\ \mathsf{in} \\
\mathsf{let}\ fun\colon ty.Up = coerce\ bounded\ obj\ \mathsf{in} \\
fun\ obj
\end{array}
$$

Fig. 18. Example in κDOT

DOT and κDOT, Scala, and ιDOT all take different approaches to running their respective versions of the program. In DOT and κDOT, obj.a would execute but diverge since obj.a would contain a divergent term. In Scala and ιDOT, obj.a would contain a null reference. Scala would read this null reference and go on to execute coerce(bounded,obj), but then throw a ClassCastException. In ιDOT, since there are no reduction rules for reading null references from fields, the abstract machine would get stuck trying to execute obj.a.

ιDOT's initialization system would reject the code in Figure 18. The constructor $kUnsafeObj$ fails the definite assignment check of the (Comm-Con) rule, so the program would be rejected.

7.2 The Type of null

Scala defines the Null type to be a subtype of all reference types [Odersky et al. 2006], and defines null to be the only instance of this type. Based on Amin and Tate [2016], it is not clear that this idea of null having any reference type is safe. Reading Rapoport et al. [2017] would suggest that null should not have any reference type, but should only have inert types. In this paper, we demonstrated that if null is considered only as a temporary placeholder for a non-null reference, it is safe to give null any type.

7.3 First Class Constructors

Scala does not have first class constructors. Scala offers second class constructors through its class system, but to the best of our knowledge there are no existing DOT calculi with a class system. We used κDOT-style first class constructors because we did not want to take on the burden of adding a class system to the calculus. However there are other languages with first class constructors, e.g. JavaScript.

The lack of a class system and using first class constructors add certain design complexities to the language - constructors are now typed by the DOT type system and are passed by reference.

We use the type system to keep track of the number and initialization types of constructor arguments rather than have the initialization system keep track of arguments. This introduces a dependence between the type and initialization systems, and prevents us from combining the type and initialization contexts into a single context like the one in the original FBC system. We have to remove initialization assumptions from our initialization context when typing constructors, but if we also remove the corresponding typing assumptions, we may not be able to convince the type system that certain references have constructor types since we may have removed too many subtyping assumptions. An alternative would have been to have the initialization type separately keep track of construction arguments and their initialization types, but this would complicate the initialization system by adding additional initialization types.

8 RELATED WORK

Relation to Freedom Before Commitment. Our work ports the FBC [Summers and Mueller 2011b] initialization system to a new calculus in the WadlerFest DOT [Amin et al. 2016] family called ιDOT. In their technical report, Summers and Mueller [2011a] formally prove the soundness of FBC for a Java/C#-like language, which we will call FBC-lang in this section. The formalism for ιDOT differs from FBC in the language, the abstract machine, and the safety proof.

The ιDOT language is based on DOT, which is structurally typed, whereas FBC-lang is a nominally typed class-based language. Unlike FBC-lang, ιDOT does not support nullable types and does not support initialization generics. This was a deliberate design decision on our part, since we wanted to build a core calculus with only the features necessary for studying object initialization in the DOT calculus. However, the ιDOT type system also is in many ways more expressive since it supports path-dependent types with $\top$ and $\bot$ types. This allows it to model features such as generics and family polymorphism, which FBC-lang lacks.

ιDOT uses a different style of abstract machine than FBC-lang. The ιDOT machine only has a heap, a stack, and a control term, similar to the mark 1 machine of Sestoft [1997]. FBC-lang uses a CESK-style machine [Felleisen and Friedman 1987], with stack frames (environments), and stack variables which are updated during execution. The FBC-lang machine explicitly throws exceptions, whereas the ιDOT machine can get stuck.

In their safety proofs, ιDOT and FBC-lang have slightly different, but related, concerns about initialization. Since the ιDOT abstract machine does not allow reading null fields, the ιDOT progress lemma shows that well-typed configurations do not get stuck. The FBC-lang safety proof instead shows that null dereference exceptions are not thrown. The differences in the abstract machines used also affect the safety proof. The FBC-lang safety proof reasons about objects reachable from stack variables before and after execution, and shows that the appropriately initialized values are reachable from stack variables after execution. They define an object to be *locally initialized* if all its non-nullable fields are initialized, and an object to be *globally initialized* if all objects (transitively) reachable from it are locally initialized. The FBC-lang proof shows that if all the arguments to a constructor are globally initialized before the execution of the constructor, then the object returned by the constructor is globally initialized. The ιDOT safety proof does not use global reachability criteria, and instead reasons only about local reachability, and shows that global constraints on the heap are maintained by the allowed updates.

Relation to Initialization Systems. Our subheap formulation was influenced by the Delayed Types system [Fähndrich and Xia 2007] and the Flexible Initialization system of Haack and Poll [2009].

In the Delayed Types system, there is a stack of delayed times and a distinct time `Now`. When objects of the latest delayed time are fully initialized, they are promoted to having the time `Now`. Promoting all objects of the latest time to `Now` is similar to how we promote the top-most subheap to the committed subheap. In some ways, their system is more flexible than ours, since they allow pointers from an object of a lower delayed time to a later delayed time. This additional flexibility comes at the cost of a more complicated initialization system that has to make comparisons between different delayed times, whereas ιDOT only has two initialization states, `committed` and `free`.

The system of Haack and Poll [2009] does not try to ensure that objects are fully initialized, but is instead concerned with initializing immutable objects. It defines subheaps of objects that are mutable for initialization and later promoted to being immutable. The system does not ensure that objects are sufficiently initialized and hence may throw null pointer exceptions.

While most related work targets existing languages with constructors, Servetto et al. [2013] propose *placeholders* as a novel language feature for creating groups of cyclically-linked objects.

Relation to DOT Calculi. ιDOT is a modification of the κDOT calculus [Kabir and Lhoták 2018]. In κDOT, fields of objects can contain arbitrary terms, which are evaluated on field reads. ιDOT restricts fields to contain only null pointers or pointers to other heap items. This makes the ιDOT type system unsound on its own, since it does not prevent programs from reading uninitialized fields. To recover soundness, ιDOT adds an effect and initialization typing system.

The pDOT calculus [Rapoport and Lhoták 2019] is an extension of WadlerFest DOT with support for full path-dependent types of the form $x.foo.bar.A$, whereas WadlerFest DOT, κDOT, and ιDOT only support types of the form $x.A$. Unlike WadlerFest DOT and κDOT, where fields contain arbitrary terms, in pDOT, fields only contain lambda literals, other objects, or paths. In pDOT, top-level objects behave like entire subheaps that are defined and allocated all at once.

Objects in pDOT are not mutable, so the object that a path points to does not change during execution. The pDOT calculus relies on objects being immutable and paths being normal forms to introduce full-path dependent types to pDOT. In ιDOT, a simple way to introduce stable paths would be to disallow mutation on committed objects, and to modify the (Init-Asn-Comm) rule in Figure 15 to only allow assignments to `free` variables. This would allow us to have typing and subtyping rules on full path-dependent types. But this still would not be enough to model some pDOT programs that require path dependent typing during initialization. For stability of paths, we need to ensure that fields are only assigned once. In addition, to allow full path assignments to fields during initialization, ιDOT would require a much stronger definite assignment analysis.

9 CONCLUSION

We have presented ιDOT, an extension of the Dependent Object Types (DOT) calculus with features for reasoning about object initialization and an effect system inspired by the Freedom Before Commitment scheme [Summers and Mueller 2011b] to guarantee that all objects are safely initialized before their fields can be read. We have verified this guarantee using the Coq proof assistant. ιDOT can serve as a solid foundation for amending the Scala language with features to statically track object initialization state and eliminate the known unsoundness due to null references reported by Amin and Tate [2016].

ACKNOWLEDGMENTS

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

There were many helpful discussions with many people while working on ιDOT. In particular, we are indebted to the following people:

- Marianna Rapoport for sharing her insights into DOT and providing feedback on early versions of this paper,
- Werner Dietl for sharing his knowledge about existing initialization systems and for pointing us to FBC,
- Abel Nieto for sharing his insights into initialization in Scala-like languages, and
- Martin Odersky and Fengyun Liu for sharing their insights on the uses of the unclassified initialization type in FBC which helped us simplify our calculus.

REFERENCES

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

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. 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, Andrew P. Black and Todd D. Millstein (Eds.). ACM, New York, NY, USA, 233–249. https://doi.org/10.1145/2660193.2660216

Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA, 838–848. https://doi.org/10.1145/2983990.2984004

Manuel Fähndrich and Songtao Xia. 2007. Establishing Object Invariants with Delayed Types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/1297027.1297052

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

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

Ifaz Kabir and Ondřej Lhoták. 2018. κDOT: Scaling DOT with Mutation and Constructors. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala* (St. Louis, MO, USA) *(Scala 2018)*. ACM, New York, NY, USA, 40–50. https://doi.org/10.1145/3241653.3241659

Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos, Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Lex Spoon, Erik Stenman, and Matthias Zenger. 2006. An Overview of the Scala Programming Language (2. Edition). (2006). http://infoscience.epfl.ch/record/85634

Xin Qi and Andrew C. Myers. 2009. Masked Types for Sound Object Initialization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. ACM, New York, NY, USA, 53–65. https://doi.org/10.1145/1480881.1480890

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A Simple Soundness Proof for Dependent Object Types. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 46 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133870

Marianna Rapoport and Ondřej Lhoták. 2019. A Path to DOT: Formalizing Fully Path-dependent Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 145 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360571

Tiark Rompf and Nada Amin. 2016. Type Soundness for Dependent Object Types (DOT). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, New York, NY, USA, 624–641. https://doi.org/10.1145/2983990.2984008

Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The Billion-Dollar Fix - Safe Modular Circular Initialisation with Placeholders and Placeholder Types. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7920)*, Giuseppe Castagna (Ed.). Springer, Berlin, Heidelberg, 205–229. https://doi.org/10.1007/978-3-642-39038-8_9

Peter Sestoft. 1997. Deriving a Lazy Abstract Machine. *Journal of Functional Programming* 7, 3 (May 1997), 231–264. https://doi.org/10.1017/S0956796897002712

Alexander J. Summers and Peter Mueller. 2011a. *Freedom Before Commitment : Simple Flexible Initialisation for Non-Null Types.* Technical Report 716. ETH Zurich.

Alexander J. Summers and Peter Mueller. 2011b. Freedom Before Commitment: A Lightweight Type System for Object Initialisation. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 1013–1032. https://doi.org/10.1145/2048066.2048142

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093