Scala Step-by-Step

Soundness for DOT with Step-Indexed Logical Relations in Iris

PAOLO G. GIARRUSSO, Delft University of Technology, Netherlands
LÉO STEFANESCO, IRIF, Université de Paris & CNRS, France
AMIN TIMANY, imec-DistriNet, KU Leuven, Belgium
LARS BIRKEDAL, Aarhus University, Denmark
ROBBERT KREBBERS, Delft University of Technology, Netherlands

The metatheory of Scala's core type system — the *Dependent Object Types (DOT)* calculus — is hard to extend, like the metatheory of other type systems combining subtyping and dependent types. Soundness of important Scala features therefore remains an open problem in theory and in practice. To address some of these problems, we use a *semantics-first* approach to develop a logical relations model for a new version of DOT, called **guarded DOT (gDOT)**. Our logical relations model makes use of an abstract form of *step-indexing*, as supported by the Iris framework, to model various forms of recursion in gDOT. To demonstrate the expressiveness of gDOT, we show that it handles Scala examples that could not be handled by previous versions of DOT, and prove using our logical relations model that gDOT provides the desired data abstraction. The gDOT type system, its semantic model, its soundness proofs, and all examples in the paper have been mechanized in Coq.[1]

1 INTRODUCTION

The Scala language has an expressive type system that supports, among other features, first-class recursive modules, path dependent types, impredicative type members, and subtyping, achieving strong information hiding. Alas, Scala has struggled for years with type soundness issues and ad-hoc fixes. To address these issues more rigorously, the compiler of the new Scala 3 language (called Dotty) has been designed hand in hand with a new foundational type system — the *Dependent Object Types (DOT)* calculus. This development led to a number of increasingly expressive versions of DOT and type soundness proofs thereof [Amin et al. 2016; Kabir and Lhoták 2018; Rapoport et al. 2017; Rapoport and Lhoták 2016; Rompf and Amin 2016], culminating in the pDOT calculus [Rapoport and Lhoták 2019], and has helped to fix various soundness bugs in Scala 3 [Rompf and Amin 2016].

Despite this exciting development, current DOT versions still lack features necessary to encode full Scala, such as subtyping for recursive types [Rompf and Amin 2016], distributive subtyping [Giarrusso 2019], higher-kinded types [Odersky 2016; Odersky et al. 2016; Stucki 2016], and mutually recursive modules that hide information from each other (which we dub *mutual information hiding*, and motivate in Sec. 1.1). Worse, one of the core DOT features is support for abstract types and data abstraction, but traditional *syntactic type soundness proofs* cannot show that abstract types behave correctly. Supporting these features in pDOT poses the following challenging questions:

(1) How to design a type system that soundly extends pDOT with these features?
(2) How to prove type soundness of such a type system?
(3) How to demonstrate proper support for data abstraction?

Question (1) is challenging because feature interaction in Scala 3 and (p)DOT is prone to unexpected type soundness issues. Question (2) is challenging because current syntactic type soundness proofs for the various DOT variants are extremely intricate, and thus hard to scale to new DOT variants. Indeed, syntactic type soundness proofs are known to be hard to scale to combinations of

[1] Available at https://github.com/Blaisorblade/dot-iris.

Authors' addresses: Paolo G. Giarrusso, Delft University of Technology, Netherlands; Léo Stefanesco, IRIF, Université de Paris & CNRS, Paris, France; Amin Timany, imec-DistriNet, KU Leuven, Leuven, Belgium; Lars Birkedal, Aarhus University, Aarhus, Denmark; Robbert Krebbers, Delft University of Technology, Netherlands.

(path) dependent types and subtyping [Hutchins 2010; Yang and d. S. Oliveira 2017]. While Rapoport et al. [2017] describe a recipe for syntactic proofs for DOT, applying this recipe to pDOT — the most expressive version of DOT to date — involves 7 carefully designed variants of pDOT's typing judgment [Rapoport and Lhoták 2019]. Finally, question (3) is challenging because it cannot be addressed through syntactic type soundness proofs.

To extend pDOT despite these challenges, we eschew traditional syntactic type soundness proofs, and follow a *semantics-first* approach: first, we model each type and typing judgment semantically via a logical relation, *i.e.,* in terms of the program's runtime behavior, instead of a fixed set of syntactic rules. Such a semantic model immediately addresses question (2): only safe programs are semantically typed. To solve question (1), we then derive from the semantic model a sound type system called **guarded DOT (gDOT)**: we give modular soundness proofs of rules that either exist in some DOT variant or are suggested by the model, such as those for mutual information hiding (see Sec. 1.1 and Sec. 3 and 4). Some rules, such as subtyping for recursive types, become easier to prove sound than in past work [Amin et al. 2016; Rompf and Amin 2016]. Other rules require extending gDOT with a "later" type operator ($\triangleright$) [Nakano 2000], which enforces certain *guardedness* restrictions obtained from the semantic model (Sec. 4). Finally, as we demonstrate in Sec. 6.3, logical relation models like ours support proving data abstraction, in particular safety of syntactically ill-typed but semantically safe code, answering question (3).

In the rest of the introduction, we motivate extending pDOT to support mutual information hiding (Sec. 1.1), discuss semantic typing (Sec. 1.2), and present contributions (Sec. 1.3).

1.1 Why Mutual Information Hiding Matters

Scala objects enable encoding a rich module system. Objects can contain not only value members (such as fields and methods), but also *type members*, which enables using objects as modules. These type members are *translucent* [Harper and Lillibridge 1994]. That is, their definition can be either exposed or *abstracted away* from clients, supporting a strong form of information hiding. Moreover, type members can be abstracted away after creation, through *upcasting*. Objects containing type members are first-class values, avoiding the need for a separate module language. Notably, they can be nested, thus supporting *hierarchical* modules, and they can be *mutually recursive*, thus enabling mutually recursive modules. This combination of features enables in particular *mutual information hiding*, that is, mutually recursive modules that hide information from each other.

To demonstrate usefulness of mutual information hiding, consider the example in Fig. 1, adapted from Rapoport and Lhoták [2019], and inspired by the actual implementation of the Scala 3 compiler (Dotty). The example models a system with mutually recursive modules `types` and `symbols`, encoded as members of the object `pcore` and representing separate compilation units. The module `types` represents the API for types. It uses nested classes to model an algebraic data type **`Type`** for types of the object language,[2] which for simplicity can be either the the top-type **`TypeTop`**, or a reference **`TypeRef`** to a symbol `symb`. The module `symbols` represents the API for a symbol table, and defines a nested class **`Symbol`** for symbols, which contain an (optional) type `tpe` and an identifier `id`. Optional types are encoded through the standard type constructor **`Option`**, with constructors **`Some`** and **`None`**, and methods `isEmpty` and `get`. We elaborate on the encoding of **`Option`** in Sec. 6.3.

The classes **`TypeRef`** and **`Symbol`** have value members (`symb` for **`TypeRef`**, and `tpe` and `id` for **`Symbol`**) that are initialized by a corresponding constructor. For instance, after executing `val s = new Symbol(None, 0)`, field `s.id` has value `0`. To achieve strong information hiding, Scala classes are *nominal*, *i.e.,* they can only be constructed through constructors. For instance, Scala rejects

[2]For our purposes, an **`abstract class`** is simply a **`class`** without constructors.

```scala
object pcore {
  object types {
    abstract class Type
    class TypeTop extends Type
    class TypeRef(val symb: pcore.symbols.Symbol) extends Type {
      assert(!symb.tpe.isEmpty) }
    val typeFromTypeRefUnsafe = (t: types.TypeRef) =>
      // relies on TypeRef invariant; only semantically well-typed.
      t.symb.tpe.asInstanceOf[Some[types.Type]].get
  }
  object symbols {
    class Symbol(val tpe: Option[pcore.types.Type], val id: Int)
    // Encapsulation violation, and type error in Scala (but not pDOT)
    // val fakeTypeRef : types.TypeRef =
    //   new { val symb = new Symbol(None, 0) }
  }
}
```

Fig. 1. A (simplified) fragment of the Scala 3 compiler (Dotty), in Scala syntax.

`fakeTypeRef`, which creates an object of type **`TypeRef`** with all the right members (namely, a member `symb` of the right type), because it sidesteps **`TypeRef`**'s constructor.

Nominality helps to enforce *class invariants* — constructors can validate parameters and initialize objects correctly. For instance, **`TypeRef`**'s constructor checks that `symb` does contain a type (using `isEmpty`). Method `fakeTypeRef` would violate this invariant, and is thus rejected. Class invariants can be relied upon by clients. Thanks to the invariants of **`Option`** and **`TypeRef`**, clients can assume that `symb.tpe` is never **`None`**, and that `symb.tpe.get` can be called safely. Indeed, `typeFromTypeRefUnsafe` relies on **`TypeRef`**'s invariant to safely extract the **`Type`** nested inside `symb`. The version of the code with `typeFromTypeRefUnsafe` is not typable in either pDOT or gDOT, as it uses an *unsafe* cast. Nonetheless, this code is in fact safe. While a syntactic type system cannot recognize all semantically safe programs, our semantic model enables proving formally the safety of such examples, similarly to the RustBelt model of Rust [Jung et al. 2018a]. We demonstrate this in Sec. 6.2 and 6.3.

Although Scala can enforce the desired abstraction in the example, pDOT cannot. To explain why, we show in Fig. 2 the translation of the example (minus `typeFromTypeRefUnsafe` and the `assert` in **`TypeRef`**'s constructor) into pDOT syntax. As the translation is verbose, we focus on key aspects.

First, we create objects through syntax $\nu x.\, \{\bar{d}\}$, where x is the *self variable* that refers to the object being created, and $\bar{d}$ is a list of type and value member definitions. The definition of the top-level object pcore uses the self variable *pcore* to create the mutual dependency between the subobjects types and symbols, represented as value members. For brevity, we write the *type declarations* of each member together with their definitions. In the core pDOT syntax, declarations would not appear in object bodies, but in their types — we would write $\nu x.\, \{\bar{d}\} : \mu x.\, \{\bar{T}\}$, where $\{\bar{T}\}$ contains type declarations for all members in $\bar{d}$, which can refer to each other through self variable x.

Second, while (p)DOT does not have native support for higher-kinded types, **`Option`**`[T]` can be encoded as $\mathit{options}.\mathsf{Option} \wedge \{\mathsf{A} :: \bot .. T\}$, exposing the type T of elements as type member `A`.[3]

Third, while classes are native constructs in Scala, they are encoded through abstract types in (p)DOT. To model that classes are nominal (*i.e.,* that they can only be created through constructors), only an upper bound on the abstract type is exposed. Hence, nominality and enforcement of class

[3]This encoding of higher-kinded types is insufficient for full Scala [Odersky et al. 2016], motivating the search for higher-kinded DOT [Odersky 2016; Stucki 2016].

```
let options = ... in let pcore = v pcore. {
  types = v types. {
    Type       >: ⊥ = ⊤
    TypeTop    >: ⊥ = types.Type
    newTypeTop : ⊤ → types.TypeTop = λ_. v_. {}
    TypeRef    >: ⊥ = types.Type ∧ {symb : pcore.symbols.Symbol}
    newTypeRef : pcore.symbols.Symbol → types.TypeRef
               = λs. { v_. {symb = s} }
  }
  symbols = v symbols. {
    Symbol     >: ⊥ = { tpe : options.Option ∧ {A :: ⊥ .. pcore.types.Type}; id : Nat }
    newSymbol  : (options.Option ∧ {A :: ⊥ .. pcore.types.Type}) → Nat → symbols.Symbol
               = λt i. v_. {tpe = t; id = i}
  }
} in ...
```

Fig. 2. The (simplified) fragment of Dotty from Fig. 1, in pDOT syntax, minus `typeFromTypeRefUnsafe` and the assertion. This code is not well-typed as-is in pDOT (see text).

invariants translate to proper data abstraction. As shorthand, we write $A >: L = U$ for a type member that is *defined* to be equal to U, but *declared* to have lower bound L and upper bound U. For example, the bounds on TypeRef are $\bot$ .. *pcore*.types.Type $\wedge$ {symb : *pcore*.symbols.Symbol}. Due to the lower bound ($\bot$, the empty type), clients of types cannot construct a TypeRef themselves. The upper bound (*pcore*.types.Type $\wedge$ {symb : *pcore*.symbols.Symbol}) exposes that TypeRef is a subtype of Type, and that it has a value member symb.

The code in Fig. 2 properly models the desired information hiding between the recursively defined subobjects types and symbols. Alas, pDOT cannot type this code, as pDOT requires that all (recursive) objects $\nu(x : T).\ \{\vec{d}\}$ must have a *precise self type* T [Rapoport and Lhoták 2019]. Informally, T is precise if the bounds L .. U of all type members that appear hereditarily in T satisfy $L = U$ — *i.e.,* if the recursively defined object does not contain any abstract type members (we define this notion formally in Fig. 4). In this case, the restriction implies that the object *pcore* cannot be typed, for instance because type member TypeRef is imprecise.[4]

pDOT's restrictions to precise self types appear necessary: pDOT with imprecise self types has known counterexamples to type soundness (see Sec. 3). To the best of our knowledge, gDOT is the first DOT variant that supports sound imprecise self types.

1.2 Why Semantics First

To find sound typing rules for imprecise self types, we approach the problem *semantics-first* — we first design a semantic model, and then derive the sound gDOT type system from it. Our model is based on an old idea going back to at least Milner [1978]: we formalize the meaning of DOT types *semantically* (using logical relations); by mapping syntactic types T to *semantic types* $\mathcal{V}[\![T]\!] \in \text{SemType}$. Semantic types $\text{SemType} \triangleq (\text{Var} \to \text{SemVal}) \to \text{SemVal} \to \text{Prop}$ are predicates on values (or equivalently, functions from values to propositions) that take as a parameter an environment mapping variables to values (since types can contain variables that point to values).

[4] In pDOT one can construct the top-level object pcore with a precise self type, and only after it is constructed use subsumption to weaken the bounds on the type members. This way, one can achieve information hiding for clients of pcore, but not information hiding between types and symbols. Hence, this is not a complete solution.

We then show that if a closed expression has a certain type T, then any result of evaluating that expression satisfies $\mathcal{V}[\![\, T \,]\!]_\varnothing$ (the *fundamental theorem*). However, we need a novel idea to handle abstract types, in particular because DOT type members are *impredicative*: that is, type members can describe values containing in turn type members, without any stratification (see Sec. 8).

To explain our gDOT model, we first sketch a naive semantics that is simple, but *unsound* because of DOT impredicativity. We then explain how we can use *step-indexing* [Appel and McAllester 2001], a common technique to deal with circularities, to give a more refined but sound model.

In (g)DOT, ignoring both base values and paths, a value can be a variable, a function value (a λ-abstraction), or an *object* (a finite map from member labels to semantic types or values). If we think of semantic types as predicates (*i.e.,* functions from values to propositions Prop), then we can describe such values using a recursive domain equation of the following form:

$$\begin{aligned} \mathsf{SemType} &= (\mathsf{Var} \to \mathsf{SemVal}) \to \mathsf{SemVal} \to \mathsf{Prop} \\ \mathsf{SemVal} &\cong \mathsf{Var} + \{\, \lambda x.\, e \,\} + (\mathsf{Label} \xrightarrow{\text{fin}} (\mathsf{SemType} + \mathsf{SemVal})) \end{aligned} \tag{Domain-Bad}$$

Intuitively, such a naive semantics would justify (p)DOT. But it would also be *unsound*, because it is well-known that there are no solutions to the above recursive domain equation in ordinary set theory due to the negative recursive occurrence of SemType.

Luckily, we can use the Iris logic [Jung et al. 2016, 2018b, 2015; Krebbers et al. 2017a] and an abstract form of step-indexing [Appel et al. 2007; Birkedal et al. 2011] to stratify our definition and build a sound version of this semantics. Our stratified equation is written as follows:

$$\begin{aligned} \mathsf{SemType} &= (\mathsf{Var} \to \mathsf{SemVal}) \to \mathsf{SemVal} \to \mathsf{iProp} \\ \mathsf{SemVal} &\cong \mathsf{Var} + \{\, \lambda x.\, e \,\} + (\mathsf{Label} \xrightarrow{\text{fin}} (\blacktriangleright \mathsf{SemType} + \mathsf{SemVal})) \end{aligned} \tag{Domain}$$

Here iProp is the Iris universe of propositions. Moreover, now the recursive occurrence of SemType is *guarded* by a "later" $\blacktriangleright$, a contractive type operator that restricts how we can manipulate semantic types. Formally, this can now be understood as a recursive domain equation in the category of complete ordered families of equivalences (COFEs) and hence solved (see [America and Rutten 1989; Birkedal et al. 2010] for more details).

Using the solution to the recursive domain equation we obtain a sound model for gDOT. Informally, we consider this model "canonical", as we have taken a straightforward but naive semantics, and done the smallest possible change to turn it into a sound semantics using step-indexing. Interestingly, our recursive domain equation differs from prior work on step-indexed logical relations, which focused mostly on modeling general references, and thus had to solve the so-called "type-world circularity" [Ahmed 2004; Birkedal et al. 2011]. Since such type systems in prior work do not support dependent types, values cannot contain types, and as such, the domain of values was not recursively defined (it was simply the set of syntactic values).

1.3 Contributions

To sum up, we take a *semantics-first* approach to take a fresh look at several open problems of Scala's core calculus pDOT. Through the semantics-first approach we obtain the new **guarded DOT (gDOT)** calculus, which enforces certain *guardedness* restrictions by extending the type system with a "later" operator ($\triangleright$). The later operator makes it possible to add a number of novel and provably sound typing rules, *e.g.,* to support imprecise self types and mutual information hiding, that were unsound in prior versions of DOT. Unfortunately, gDOT's later operator also comes with a price — it hinders typing programs that were accepted by previous (p)DOT versions. Yet, despite these limitations we demonstrate that we can encode many challenging examples from the Scala and (p)DOT literature, as well as new examples that could not be handled before.

Concretely, this paper makes the following contributions:

Syntax

$$
\begin{array}{rll}
\mathsf{TyLabel} \ni \mathsf{A} & & \textit{Type member labels} \\
\mathsf{ValLabel} \ni \mathsf{a} & & \textit{Term member labels} \\
\mathsf{Label} \ni l & ::= \mathsf{a} \mid \mathsf{A} & \textit{Member labels} \\
\mathsf{Val} \ni v & ::= x \mid \lambda x.\, e \mid v x.\, \{\bar{d}\} & \textit{Values} \\
\mathsf{Expr} \ni e & ::= v \mid e\ e \mid e.\mathsf{a} \mid \mathbf{coerce}\ e & \textit{Expressions/terms} \\
\mathsf{Path} \ni p, q & ::= v \mid p.\mathsf{a} & \textit{(Pre)paths} \\
\mathsf{DefBody} \ni d & ::= p \mid T & \textit{Definition bodies} \\
\mathsf{DefList} \ni \bar{d} & ::= l = d \mid \bar{d}; \bar{d} & \textit{Definition lists} \\
\mathsf{ECtx} \ni K & ::= [\,] \mid K\ e \mid v\ K \mid K.a \mid \mathbf{coerce}\ K & \textit{Evaluation contexts} \\
\mathsf{Type} \ni L, S, T, U, V, W & ::= \top \mid \bot \mid S \wedge T \mid S \vee T \mid \forall x : S.\, T & \textit{(Pre)types} \\
& \quad \mid \{\mathsf{a} : T\} \mid \{\mathsf{A} :: L\ ..\ U\} \mid p.\mathsf{A} \mid p.\mathbf{type} \mid \mu x.\, T \mid \triangleright T & \\
\mathsf{TyCtx} \ni \Gamma & ::= \varepsilon \mid \Gamma, x : T & \textit{Typing contexts}
\end{array}
$$

Member selection (looking up label l in value v finds definition d)

$$v.l \searrow d \triangleq \exists x, \bar{d}.\ v = v x.\, \{\bar{d}\} \wedge \mathsf{lookup}\ l\ (\bar{d}[x := v]) = d$$

Operational semantics (call-by-value head reduction $e \to_{\mathsf{h}} e'$, and its closure $e \to_{\mathsf{t}} e'$ over contexts)

$$(\lambda x.\, e)\ v \to_{\mathsf{h}} e[x := v] \qquad \frac{v.a \searrow p}{v.a \to_{\mathsf{h}} p} \qquad \mathbf{coerce}\ v \to_{\mathsf{h}} v \qquad \frac{e \to_{\mathsf{h}} e'}{K[e] \to_{\mathsf{t}} K[e']}$$

Fig. 3. pDOT/gDOT syntax and operational semantics. New gDOT constructs have a shaded background.

- We motivate extending pDOT with support for *imprecise self types*, to enable type abstractions between mutually recursive objects, despite the known difficulties (Sec. 1.1 and Sec. 3).
- After summarizing the pDOT calculus (Sec. 2), we introduce our new gDOT calculus (Sec. 4).
- We introduce a novel technique, based on step-indexed logical relations, to give a semantic model of impredicative type members, and use it to prove soundness of gDOT (Sec. 5).
- We demonstrate gDOT's expressivity by encoding various examples, and demonstrate its support for data abstraction by proving semantic typing of functions whose correctness relies on gDOT's support for data abstraction (Sec. 6).
- We mechanize gDOT and all proofs in this paper in Coq using the Iris framework (Sec. 7).

2 BACKGROUND: PDOT

Before we present gDOT in Sec. 4, we summarize the pDOT calculus [Rapoport and Lhoták 2019], which gDOT uses as a basis. To simplify the comparison, we use a reformulation of pDOT that is closer to gDOT, but that is in essence the same as the original calculus.

2.1 Syntax and Operational Semantics

The pDOT and the gDOT calculi share syntax and operational semantics, which are presented in Fig. 3. For brevity, we ignore primitives like numerals and addition. Unlike in Sec. 1.2, we define syntactic values and types, not semantic ones (we return to semantic values in Sec. 5.3). pDOT values are either functions $\lambda x.\, e$ or objects $v x.\, \{\bar{d}\}$. An object contains a map from labels to definitions $\bar{d}$, which can reference the whole object through the *self variable* x, modeling the `this` variable in Scala. A definition d can be a *type member* $\{\mathsf{A} = T\}$, where A is a type label and T is a type, or a *term member* $\{a = p\}$, where a is a label and p is a *(pre)path*. Though they are central to the type

Expression typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash e : T_2}\ (\text{T-Sub}) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : T}{\Gamma \vdash p : T}\ (\text{T-Path})$$

$$\frac{\Gamma \mid x : T \vdash \{\bar{d}\} : T}{\Gamma \vdash \nu x.\{\bar{d}\} : \mu x.T}\ (\text{T-}\{\}\text{-I}) \qquad \frac{\Gamma \vdash e : \{\mathsf{a} : T\}}{\Gamma \vdash e.\mathsf{a} : T}\ (\text{T-}\{\}\text{-E}) \qquad \frac{\Gamma, x : S \vdash e : T \qquad x \notin \mathsf{FV}(S)}{\Gamma \vdash \lambda x.\, e : \forall\, x : S.\, T}\ (\text{T-}\forall\text{-I})$$

$$\frac{\Gamma \vdash e_1 : S \to T \qquad \Gamma \vdash e_2 : S}{\Gamma \vdash e_1\ e_2 : T}\ (\text{T-}\forall\text{-E}) \qquad \frac{\Gamma \vdash e : \forall\, z : S.\, T \qquad \Gamma \vdash p : S}{\Gamma \vdash e\ p : T[z := p]}\ (\text{T-}\forall\text{-E}_p)$$

Path typing $\boxed{\Gamma \vdash_{\mathbf{p}} p : T}$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash_{\mathbf{p}} x : T}\ (\text{P-Var}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : T[x := p]}{\Gamma \vdash_{\mathbf{p}} p : \mu x.T}\ (\text{P-}\mu\text{-I}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : \mu x.T}{\Gamma \vdash_{\mathbf{p}} p : T[x := p]}\ (\text{P-}\mu\text{-E}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : T_1 \qquad \Gamma \vdash_{\mathbf{p}} p : T_2}{\Gamma \vdash_{\mathbf{p}} p : T_1 \wedge T_2}\ (\text{P-}\wedge\text{-I})$$

$$\frac{\Gamma \vdash_{\mathbf{p}} p : T \qquad \Gamma \vdash T <: U}{\Gamma \vdash_{\mathbf{p}} p : U}\ (\text{P-Sub}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : \{a : T\}}{\Gamma \vdash_{\mathbf{p}} p.a : T}\ (\text{P-Fld-E}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p.a : T}{\Gamma \vdash_{\mathbf{p}} p : \{a : T\}}\ (\text{P-Fld-I})$$

$$\frac{\Gamma \vdash_{\mathbf{p}} p : q.\mathbf{type} \qquad \Gamma \vdash_{\mathbf{p}} q : T}{\Gamma \vdash_{\mathbf{p}} p : T}\ (\text{P-Sngl-Trans}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : q.\mathbf{type} \qquad \Gamma \vdash_{\mathbf{p}} q.a : T}{\Gamma \vdash_{\mathbf{p}} p.a : q.a.\mathbf{type}}\ (\text{P-Sngl-E})$$

Definition typing $\boxed{\Gamma \mid x : V \vdash \{\bar{d}\} : T}$

$$\frac{\Gamma, x : V \vdash v : T \qquad \mathbf{tight}\ T}{\Gamma \mid x : V \vdash \{a = v\} : \{a : T\}}\ (\text{D-Val}) \qquad \frac{\Gamma, x : V \mid z : x.\mathsf{a}.\mathbf{type} \wedge T \vdash \{\bar{d}\} : T \qquad \mathbf{tight}\ T}{\Gamma \mid x : V \vdash \{\mathsf{a} = \nu z.\{\bar{d}\}\} : \{\mathsf{a} : \mu z.T\}}\ (\text{D-Val-New})$$

$$\frac{}{\Gamma \mid x : V \vdash \{\mathsf{A} = T\} : \{\mathsf{A} :: T\ ..\ T\}}\ (\text{D-Typ}) \qquad \frac{\Gamma, x : V \vdash_{\mathbf{p}} p : T}{\Gamma \mid x : V \vdash \{a = p\} : \{a : p.\mathbf{type}\}}\ (\text{D-Path-Sngl})$$

$$\frac{\Gamma \mid x : V \vdash \{\bar{d}_1\} : T_1 \qquad \Gamma \mid x : V \vdash \{\bar{d}_2\} : T_2 \qquad \mathrm{dom}\,\bar{d}_1, \mathrm{dom}(\bar{d}_2)\ \text{disjoint}}{\Gamma \mid x : V \vdash \{\bar{d}_1; \bar{d}_2\} : T_1 \wedge T_2}\ (\text{D-And})$$

***Tight* (or *precise*) types** $\boxed{\mathbf{tight}\ T}$

$$\mathbf{tight}\ T = \begin{cases} L = U & \text{if } T = \{\mathsf{A} :: L\ ..\ U\} \\ \mathbf{tight}\ U & \text{if } T = \mu(x : U) \text{ or } T = \{\mathsf{a} : U\} \\ \mathbf{tight}\ U \text{ and } \mathbf{tight}\ V & \text{if } T = U \wedge V \\ \mathsf{True} & \text{otherwise} \end{cases}$$

Fig. 4. pDOT rules for expression typing, path typing, and definition typing. Path typing is a special case of expression typing in (p)DOT, but not in our presentation or in gDOT.

system, type members do not affect the operational semantics. Type and term members can be projected out from objects using *member selectors*, respectively $e.a$ and $p.\mathsf{A}$. A path is either a value v or a selection $p.a$. We rely on the type system to reject nonsensical paths such as $(\lambda x.\, e).a$.

Like in *storeless DOT* [Amin 2016, Ch. 3], we use a conventional substitution-based call-by-value semantics. Substitution of variables by *values* is written as $\chi[x := v]$, where χ ranges over all syntactic classes. We write $e \to_{\mathsf{h}} e'$ for head reduction, and write $e \to_{\mathsf{t}} e'$ for its closure under call-by-value evaluation contexts K. Head reduction has three rules: the usual call-by-value β-reduction

Top, bottom, and intersection types $\boxed{\Gamma \vdash T_1 <: T_2}$

$$\Gamma \vdash T <: \top \ (<:\text{-}\top) \qquad \Gamma \vdash T_1 \wedge T_2 <: T_1 \ (\wedge_1\text{-}<:) \qquad \Gamma \vdash T_1 \wedge T_2 <: T_2 \ (\wedge_2\text{-}<:) \qquad \Gamma \vdash \bot <: T \ (\bot\text{-}<:)$$

$$\frac{\Gamma \vdash T <: U_1 \qquad \Gamma \vdash T <: U_2}{\Gamma \vdash T <: U_1 \wedge U_2} \ (<:\text{-}\wedge) \qquad \Gamma \vdash T <: T \ (<:\text{-}\textsc{Refl}) \qquad \frac{\Gamma \vdash S <: T \qquad \Gamma \vdash T <: U}{\Gamma \vdash S <: U} \ (<:\text{-}\textsc{Trans})$$

Type members

$$\frac{\Gamma \vdash_{\mathbf{p}} p : \{\mathsf{A} :: L \,..\, U\}}{\Gamma \vdash L <: p.\mathsf{A}} \ (<:\text{-}\textsc{Sel}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : \{\mathsf{A} :: L \,..\, U\}}{\Gamma \vdash p.\mathsf{A} <: U} \ (\textsc{Sel}\text{-}<:)$$

Co/contra-variant subtyping

$$\frac{\Gamma \vdash T <: U}{\Gamma \vdash \{a : T\} <: \{a : U\}} \ (\textsc{Fld}\text{-}<:\text{-}\textsc{Fld}) \qquad \frac{\Gamma \vdash L_2 <: L_1 \qquad \Gamma \vdash U_1 <: U_2}{\Gamma \vdash \{\mathsf{A} :: L_1 \,..\, U_1\} <: \{\mathsf{A} :: L_2 \,..\, U_2\}} \ (\textsc{Typ}\text{-}<:\text{-}\textsc{Typ})$$

$$\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma, x : T_2 \vdash U_1 <: U_2}{\Gamma \vdash \forall\, x : T_1.\, U_1 <: \forall\, x : T_2.\, U_2} \ (\forall\text{-}<:\text{-}\forall)$$

Singleton types

$$\frac{\Gamma \vdash_{\mathbf{p}} p : q.\mathbf{type} \qquad T_1 \cong^*_{p:=q} T_2}{\Gamma \vdash T_1 <: T_2} \ (\textsc{Sngl}_{pq}\text{-}<:) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p : q.\mathbf{type} \qquad T_1 \cong^*_{p:=q} T_2}{\Gamma \vdash T_2 <: T_1} \ (\textsc{Sngl}_{qp}\text{-}<:)$$

Fig. 5. pDOT rules for subtyping.

for function values, evaluation of member selectors, and evaluation of coercions **coerce**. As DOT objects are recursive, the member lookup relation $v.l \searrow d$ for an object $v = vx.\, \{\bar{d}\}$ substitutes the self variable x by v before looking up l in the substitution result. Last, coercions applied to values simply reduce away in one evaluation step, which will become significant in Sec. 5.3. Coercions get their name because they appear in gDOT's subsumption rule (T-Sub) in Fig. 6.

2.2 Type System

We now present the DOT (pre)types (Fig. 3) and type system (Fig. 4 and 5). We focus on the typing rules that are essential to the rest of the paper, and defer to Rapoport and Lhoták [2019] for the remaining ones. A term in DOT can be typed either with a dependent function type $\forall\, x : S.\, T$, where x can appear in T, a record type $\{a : T\}$ or $\{\mathsf{A} :: S..\, T\}$, a path selection $p.\mathsf{A}$, a singleton type $p.\mathbf{type}$, or an object type $\mu x.\, T$. In addition, pDOT features the usual top ($\top$), bottom ($\bot$), and intersection ($\wedge$) types.[5] We use $\{\bar{T}\}$ as sugar for intersections, *e.g.*, we let $\{\mathsf{A} :: L \,..\, U;\, \mathsf{a} : T\}$ be syntactic sugar for $\{\mathsf{A} :: L \,..\, U\} \wedge \{\mathsf{a} : T\}$. Distinct members $\bar{T}$ of type $\mu x.\, \{\bar{T}\}$ cannot refer to each other directly, but only through the self variable x.

The typing judgments contain a context Γ, which is a list of mappings $x : T$ from variables x to types T. Contexts are dependently typed, but with non-standard scoping: in context $\Gamma_1, x : T, \Gamma_2$, the variable x is bound not only in Γ_2, but also in T.

The expression typing judgment $\Gamma \vdash e : T$ (which also covers values), and subtyping judgment $\Gamma \vdash T_1 <: T_2$, as well as the subsumption rule (T-Sub), are standard. We write $\Gamma \vdash T_1 <: T_2 <: T_3$ for having both $\Gamma \vdash T_1 <: T_2$ and $\Gamma \vdash T_2 <: T_3$. In our reference version of DOT, we follow WadlerFest DOT [Amin et al. 2016] and pDOT by leaving out subtyping rules for recursive types, which we will add back in gDOT in Sec. 4. Dependent function types $\forall\, x : S.\, T$ support standard rules for contravariant subtyping and introduction. Non-dependent functions can be applied to an arbitrary

[5] Some versions of DOT [Rompf and Amin 2016] have union ($\vee$) types, but pDOT does not. We readd union types in gDOT.

argument expression using (T-∀-E). Dependent functions can only be applied to a path p using (T-∀-E$_p$) as we can only substitute paths into types using path substitution $T[x := p]$.

Typing of object values $vx.\{\bar{d}\}$ in rule (T-{}-I) depends on the *definition typing judgment* $\Gamma \mid x : V \vdash \{\bar{d}\} : T$. Here, the binding for the self variable $x : V$ (which refers to the object being constructed) is placed in a stoup (*i.e.,* one-element context) instead of the context Γ because it has a special role in rules (D-Val) and (D-Val-New) to type value definitions. These rules require constructed objects to have *precise* self types by using Rapoport and Lhoták's predicate **tight** T [2019].

Paths p are the only terms that can appear in types, through selections p.A and singleton types p.**type**. Paths p that appear in types always start with a variable, *i.e.,* they do not contain values. Paths are typed using the *path typing judgment* $\Gamma \vdash_{\mathtt{P}} p : T$, which, unlike expression typing, also guarantees that evaluating p terminates. Intuitively, a type selection p.A refers to the type definition for member A in the result of evaluating p. However, rules (Sel-<:) and (<:-Sel) relate p.A only to the upper and lower bound of A in the type of p, not the definition of A: this ensures that abstract types are indeed abstract. Finally, (Typ-<:-Typ) enables making a type member of p (more) abstract, by simply upcasting p.

Intuitively, the singleton type p.**type** contains a value v if path p is statically guaranteed to evaluate to v. We say that two paths p and q *alias each other* when $\Gamma \vdash_{\mathtt{P}} p : q$.**type** is derivable, which guarantees that both p and q evaluate to the same value. Aliases can be created among others using rule (D-Val-New), which combines (D-Val) and (T-{}-I), but also records that z and $x.a$ are aliases. Aliasing can be *hidden* through (P-Sngl-Trans): if x is initialized with p, but x's type is not a subtype of p.**type**, then x will not alias p. The rules (Sngl$_{pq}$-<:) and (Sngl$_{qp}$-<:) use the relation $T \cong^*_{p:=q} U$, which is the reflexive transitive closure of Rapoport and Lhoták's path replacement [2019]. Informally, $T \cong^*_{p:=q} U$ means that zero or more occurrences of p in T are replaced by q in U, with the rest of T, including any other occurrence of p, left unchanged.

3 UNSOUNDNESS OF DOT WITH IMPRECISE SELF TYPES

As discussed in Sec. 1.1, imprecise self types are useful to support certain forms of information hiding, but current versions of DOT do not support them soundly. In this section we indicate why the restriction to *tight* (or *precise*) types (see Fig. 4) excludes the example in Fig. 2 from Sec. 1.1, but is necessary for soundness of current versions of DOT.

To construct a typing derivation for the example in Fig. 2 we initially give type members such as $\{\mathsf{TypeRef} = (\mathit{pcore}.\mathsf{types}.\mathsf{Type} \wedge \{\mathsf{symb} : \mathit{pcore}.\mathsf{symbols}.\mathsf{Symbol}\})\}$ a concrete type, *i.e.,* we give them exact lower and upper bounds. This is needed to type constructors such as newTypeRef, as their bodies need to know the concrete types of the objects they construct. The bodies of symbols and types are then typed using (T-{}-I). Since these bodies still have concrete types, we need the subsumption rule (T-Sub) to upcast them to give them types S and T in which type members like TypeRef are abstract. Finally, we need (D-Val) to type $\{\mathsf{symbols} : S\}$ and $\{\mathsf{types} : T\}$ for the right types S and T, but this is impossible — since S and T contain abstract types, they violate the **tight** T side-condition on (D-Val), which prevents typing this example.

Unfortunately, as shown by Rapoport and Lhoták [2019], removing this side-condition is unsound, similarly to other desirable generalizations of DOT rules. We discuss two such generalizations:

- Seemingly, to type our example, one could type $\{\mathsf{symbols} : S'\}$ and $\{\mathsf{types} : T'\}$ with tight types T' and S' using (D-Val), and upcast them via subsumption to non-tight types T and S. But this attempt fails, because DOT lacks the following subsumption rule for term members:

$$\frac{\Gamma \mid x : V \vdash \{a = p\} : \{a : T_1\} \qquad \Gamma, x : V \vdash T_1 <: T_2}{\Gamma \mid x : V \vdash \{a = p\} : \{a : T_2\}}\ \text{(D-Path-Sub-Bad)}$$

Amin [2016, Sec. 3.5.5] showed that such a rule is unsound.

- The rule (D-Typ) restricts type members $\{\mathsf{A} = T\}$ to have tight types $\{\mathsf{A} :: T .. T\}$. One may wonder if this rule could be generalized to non-tight bounds as follows:

$$\frac{\Gamma, x : V \vdash L <: T \qquad \Gamma, x : V \vdash T <: U}{\Gamma \mid x : V \vdash \{\mathsf{A} = T\} : \{\mathsf{A} :: L .. U\}}\ \text{(D-Typ-Abs-Bad)}$$

Amin [2016, Sec. 3.5.5] showed that such a rule is unsound as well.

All these rules break DOT type soundness in a similar way. A closed value with type $\{\mathsf{A} :: L .. U\}$ is a *witness* that $L <: U$, and allows upcasting L to U. Closed values with *bad bounds* [Amin 2016], such as $\top <: \bot$, enable casting values across arbitrary types, and thereby breaking type soundness. All of the aforementioned unsound rules enable constructing closed values with type $\mu\_.\ \{\mathsf{A} :: \top .. \bot\}$, from which one can deduce the inconsistent subtyping $\top <: \bot$. For example, using the unsound rule (D-Typ-Abs-Bad) displayed above, one can show that $\nu x.\ \{\mathsf{A} = \top\}$ has said type:

$$\frac{\dfrac{x : \{\mathsf{A} :: \top .. \bot\} \vdash \top <: \top <: \bot}{\varepsilon \mid x : \{\mathsf{A} :: \top .. \bot\} \vdash \{\mathsf{A} = \top\} : \{\mathsf{A} :: \top .. \bot\}}\ \text{(D-Typ-Abs-Bad)}}{\varepsilon \vdash \nu x.\ \{\mathsf{A} = \top\} : \mu x.\ \{\mathsf{A} :: \top .. \bot\}}\ \text{(T-\{\}-I)}$$

In the premise of (T-{}-I), the context is extended with an unsound subtyping witness, the self variable x. This witness enables proving $\top <: \bot$, hence proving (unsoundly) that type definition $\{\mathsf{A} = \top\}$ is between its bounds.

To rule out such unsound circular derivations, all previous calculi in the DOT family use the same solution — they restrict object creation to tight (*i.e.,* precise) self types, so that rule (T-{}-I) becomes sound. If T is a precise self type, it can only carry proofs for subtypings of the form $U <: U$, which are always true. To enforce this restriction, DOT puts the **tight** T side-condition on (D-Val), and eschews rules like (D-Path-Sub-Bad) and (D-Typ-Abs-Bad). While this ensures soundness, it rules out imprecise self types, and therefore useful forms of data abstraction.

Our gDOT calculus takes a different route — it imposes a *guardedness* condition on the self variable to ensure it is not used in circular way. Hence, we can soundly support imprecise self types, *i.e.,* allow variants of rules like (D-Val) without the **tight** T side-condition, and (D-Path-Sub-Bad) and (D-Typ-Abs-Bad). To realize such a guardedness condition, we give the self variable a different and *weaker* type. Since DOT provides no suitable candidate, we will extend the language of DOT types. These changes enable us to type examples including the one in Fig. 2 from Sec. 1.1.

Does this make Scala unsound? One might wonder if the aforementioned counterexamples to type soundness affect Scala's support for imprecise self types; but we are unable to encode the counterexamples in Scala. To the best of our understanding, that is because counterexamples, like $\nu x.\ \{\mathsf{A} = \top\}$ from this section, rely on transitivity of subtyping to deduce $x : \{\mathsf{A} :: L .. U\} \vdash L <: U$ from $x : \{\mathsf{A} :: L .. U\} \vdash L <: x.\mathsf{A} <: U$. This use of transitivity is not admissible the Scala compiler's (*i.e.,* Dotty's) algorithmic type system [Hu and Lhoták 2020; Nieto 2017]. Nevertheless, it is not at all clear that all such counterexamples are forbidden by Dotty, nor how to prove soundness of imprecise self types by relying on the absence of transitivity.

4 THE GDOT TYPE SYSTEM

To support imprecise self types while avoiding the soundness problems from Sec. 3, our guarded DOT (gDOT) calculus imposes a *guardedness condition* that ensures that the self variable x in recursive objects $\nu(x : T).\ \{\bar{d}\}$ is not used in a cyclic way. We enforce this condition by extending DOT with a "later" type operator ($\triangleright$), so that x can be given type $\triangleright\, T$ instead of T. The type $\triangleright\, T$ is

weaker than T, in the sense that it cannot be used directly in the construction of the object's body $\bar{d}$. Instead, one needs to take a program step to eliminate the later, *i.e.,* to turn $\triangleright T$ into T. The most prominent places where the later type ($\triangleright$) appears in gDOT are:

$$\frac{\Gamma \mid x : \triangleright T \vdash \{\bar{d}\} : T}{\Gamma \vdash \nu x.\, \{\bar{d}\} : \mu x.\, T} \qquad \frac{\Gamma, x : V, z : S \vdash t : T \qquad z \notin \mathsf{FV}(S)}{\Gamma \mid x : \triangleright V \vdash \{\mathsf{a} = \lambda z.\, t\} : \{\mathsf{a} : \forall z : S.\, T\}} \qquad \frac{\Gamma \vdash e : \triangleright T}{\Gamma \vdash \textbf{coerce}\ e : T}$$

The first rule is gDOT version of the introduction rule for recursive objects (T-{}-I), which puts $x : \triangleright V$ instead of $x : V$ in the context. The later can be eliminated using the other rules — either by constructing a function, or by taking an explicit step using gDOT's **coerce** construct.

Our use of the later type operator ($\triangleright$) is inspired by, and *reflects* into gDOT, the later modality ($\triangleright$) in step-indexed (program) logics such as Iris (as we will see in Sec. 5.3). Such logics provide the principle of *Löb induction*, which allows proving proposition P under the induction hypothesis $\triangleright P$, and which we use to prove (T-{}-I) sound. In Löb induction, the later in the induction hypothesis $\triangleright P$ can be eliminated by taking a program step.

By ensuring that self variables are only used in a guarded way, we can remove the **tight** T side-condition of rules (D-Val) and (D-Val-New), generalize (D-Path) to (D-Path-Singl), and add sound versions of (D-Typ-Abs-Bad) and (D-Path-Sub-Bad) to gDOT. Such rules become sound because rule (T-{}-I) only types the self variable as $\triangleright V$, and thereby prevents unsound circular derivations. Notably, the counterexamples from Sec. 3 are ruled out because it is impossible to deduce $\top <: \bot$ from $x : \triangleright \{A :: \top .. \bot\}$, just like one cannot deduce False from $\triangleright$ False in step-indexed logics.

Using gDOT's sound version of (D-Val-New), the example in Fig. 2 becomes well-typed without changes: imprecise self types enable hiding the definition of TypeRef and Symbol from each other, ensuring mutual information hiding.

The typing rules and subtyping rules of gDOT are displayed in Fig. 6 and 7. In this section we highlight some of the important parts. To enable flexible bookkeeping of laters, we generalize the subtyping and path typing judgments to their *delayed* variants (Sec. 4.1). We explain the guardedness restrictions gDOT imposes on type selectors (Sec. 4.2), and how laters can be eliminated through function introduction (Sec. 4.3). We show that in addition to aforementioned new rules, gDOT also supports a number of other rules that prior versions of DOT did not support (Sec. 4.4).

4.1 Delayed Judgments

Upon inspiration by Pfenning and Davies [2001], we extend subtyping $\Gamma \vdash S\ {}^{i}{<:}^{j}\ T$ and path typing $\Gamma \vdash_{\mathbf{p}} p :^{i} T$ judgments with so called *delays* $i, j \in \mathbb{N}$, allowing one to deal with laters flexibly.

Most of gDOT's delayed subtyping rules are generalizations of pDOT rules to arbitrary delays, except for a guardedness restriction in (Sel-<:) (see Sec. 4.2). As such, delayed subtyping $\Gamma \vdash S\ {}^{i}{<:}^{j}\ T$ with $i = j = 0$ corresponds to ordinary subtyping $\Gamma \vdash S <: T$. In addition, the rules (Later-<:), (<:-Later) make it possible to push laters into the delay, and (<:-Add-Later) ensures that $\triangleright T$ is a supertype of T. From (Later-<:), (<:-Later), and transitivity, we obtain $\Gamma \vdash S\ {}^{i}{<:}^{j}\ T$ iff $\Gamma \vdash \triangleright^{i} S <: \triangleright^{j} T$. One may thus wonder why not always use $\Gamma \vdash \triangleright^{i} S <: \triangleright^{j} T$? The reason is that application of rules like ($\wedge_1$-<:) may be blocked by occurrences of later, but by using delayed subtyping we can push the laters into the delay and unblock such rules.

Using gDOT's subsumption rule (T-Sub), given $\Gamma \vdash T_1\ {}^{0}{<:}^{i}\ T_2$ we can coerce an expression e of type T_1 into $\textbf{coerce}^{i}\ e$ of type T_2. Coercions are not allowed in paths, so gDOT's subsumption rule (P-Sub) for path typing is different — it involves the delayed path typing judgment $\Gamma \vdash_{\mathbf{p}} p :^{i} T$, which tracks of the number of delays obtained from delayed subsumption.[6] We can derive rule (P-Later), saying that that $\Gamma \vdash_{\mathbf{p}} p :^{i} \triangleright T$ implies $\Gamma \vdash_{\mathbf{p}} p :^{i+1} T$, but not the converse.

[6] Rule (P-Sub) is restricted, so that it cannot *reduce* the judgment index i, for reasons due to our semantic model.

Expression typing (rules (T-Var), (T-{}-E), (T-∀-E), and (T-∀-E$_p$) are unchanged and thus elided) $\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : T_1 \qquad \Gamma \vdash T_1 \ {}^{0}{<:}^{\mathrm{i}}\ T_2}{\Gamma \vdash \mathbf{coerce}^{i}\ e : T_2}\ (\text{T-Sub}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{0} T}{\Gamma \vdash p : T}\ (\text{T-Path}) \qquad \frac{\Gamma \mid x : \triangleright T \vdash \{\bar{d}\} : T}{\Gamma \vdash \nu x.\, \{\bar{d}\} : \mu x.\, T}\ (\text{T-}\{\}\text{-I})$$

$$\frac{\Gamma_1 \gg_{\triangleright} \Gamma_2 \qquad \Gamma_2, x : S \vdash e : T \qquad x \notin \mathsf{FV}(S)}{\Gamma_1 \vdash \lambda x.\, e : \forall\, x : S.\, T}\ (\text{T-}\forall\text{-I-Strong})$$

Path typing (rule (P-∧-I) is derivable) $\boxed{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T}$

$$\frac{\Gamma \vdash x : T}{\Gamma \vdash_{\mathbf{p}} x :^{0} T}\ (\text{P-Var}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T[z \coloneqq p]}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \mu z.\, T}\ (\text{P-}\mu\text{-I}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \mu z.\, T}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T[z \coloneqq p]}\ (\text{P-}\mu\text{-E})$$

$$\frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T \qquad \Gamma \vdash T \ {}^{\mathrm{i}}{<:}^{\mathrm{i+j}}\ U}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i+j}} U}\ (\text{P-Sub}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \{\mathsf{a} : T\}}{\Gamma \vdash_{\mathbf{p}} p.\mathsf{a} :^{\mathrm{i}} T}\ (\text{P-Fld-E}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p.\mathsf{a} :^{\mathrm{i}} T}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \{\mathsf{a} : T\}}\ (\text{P-Fld-I})$$

$$\frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} q.\mathbf{type} \qquad \Gamma \vdash_{\mathbf{p}} q :^{\mathrm{i}} T}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T}\ (\text{P-Sngl-Trans}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} q.\mathbf{type} \qquad \Gamma \vdash_{\mathbf{p}} q.\mathsf{a} :^{\mathrm{i}} T}{\Gamma \vdash_{\mathbf{p}} p.\mathsf{a} :^{\mathrm{i}} q.\mathsf{a}.\mathbf{type}}\ (\text{P-Sngl-E})$$

$$\frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} p.\mathbf{type}}\ (\text{P-Sngl-Refl}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} q.\mathbf{type}}{\Gamma \vdash_{\mathbf{p}} q :^{\mathrm{i}} \top}\ (\text{P-Sngl-Inv})$$

Definition typing (rule (D-Path-Singl) is derivable) $\boxed{\Gamma \mid x : V \vdash \{\bar{d}\} : T}$

$$\frac{\Gamma, x : V \vdash v : T}{\Gamma \mid x : V \vdash \{\mathsf{a} = v\} : \{\mathsf{a} : T\}}\ (\text{D-Val}) \qquad \frac{\Gamma, x : V \mid z : x.\mathsf{a}.\mathbf{type} \wedge \triangleright T \vdash \{\bar{d}\} : T}{\Gamma \mid x : V \vdash \{\mathsf{a} = \nu z.\, \{\bar{d}\}\} : \{\mathsf{a} : \mu z.\, T\}}\ (\text{D-Val-New})$$

$$\frac{\Gamma, x : V \vdash \triangleright L \ {}^{0}{<:}^{0}\ \triangleright T \qquad \Gamma, x : V \vdash \triangleright T \ {}^{0}{<:}^{0}\ \triangleright U}{\Gamma \mid x : V \vdash \{\mathsf{A} = T\} : \{\mathsf{A} :: L \,..\, U\}}\ (\text{D-Typ-Abs}) \qquad \frac{\Gamma, x : V \vdash_{\mathbf{p}} p :^{0} T}{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T\}}\ (\text{D-Path})$$

$$\frac{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T_1\} \qquad \Gamma, x : V \vdash T_1 \ {}^{0}{<:}^{0}\ T_2}{\Gamma \mid x : V \vdash \{\mathsf{a} = p\} : \{\mathsf{a} : T_2\}}\ (\text{D-Path-Sub})$$

$$\frac{\Gamma \mid x : V \vdash \{\bar{d}_1\} : T_1 \qquad \Gamma \mid x : V \vdash \{\bar{d}_2\} : T_2 \qquad \mathrm{dom}(\bar{d}_1), \mathrm{dom}(\bar{d}_2)\ \text{disjoint}}{\Gamma \mid x : V \vdash \{\bar{d}_1; \bar{d}_2\} : T_1 \wedge T_2}\ (\text{D-And})$$

Notable derivable typing rules

$$\frac{\Gamma \vdash e : p.\mathsf{A} \qquad \Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \{\mathsf{A} :: L \,..\, U\}}{\Gamma \vdash \mathbf{coerce}^{i+1}\ e : U}\ (\text{T-Sel-Unfold}) \qquad \frac{\Gamma, x : S \vdash e : T \qquad x \notin \mathsf{FV}(S)}{\Gamma \vdash \lambda x.\, e : \forall\, x : S.\, T}\ (\text{T-}\forall\text{-I})$$

$$\frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T_1 \qquad \Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T_2}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T_1 \wedge T_2}\ (\text{P-}\wedge\text{-I}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \triangleright T}{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i+1}} T}\ (\text{P-Later}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} q.\mathbf{type}}{\Gamma \vdash_{\mathbf{p}} q :^{\mathrm{i}} p.\mathbf{type}}\ (\text{P-Sngl-Sym})$$

$$\frac{\Gamma, x : V \vdash_{\mathbf{p}} p : T}{\Gamma \mid x : V \vdash \{a = p\} : \{a : p.\mathbf{type}\}}\ (\text{D-Path-Singl}) \qquad \frac{\Gamma, x : V, z : S \vdash t : T \qquad z \notin \mathsf{FV}(S)}{\Gamma \mid x : \triangleright V \vdash \{\mathsf{a} = \lambda z.\, t\} : \{\mathsf{a} : \forall\, z : S.\, T\}}\ (\text{D-}\forall)$$

Fig. 6. gDOT rules for expression typing, path typing, and definition typing.

Bounded, distributive subtyping lattice $\boxed{\Gamma \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ T_2}$

$$\Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{i}} \top \ (<:\text{-}\top) \qquad \Gamma \vdash T_1 \wedge T_2 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T_1 \ (\wedge_1\text{-}<:) \qquad \Gamma \vdash T_1 \wedge T_2 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T_2 \ (\wedge_2\text{-}<:) \qquad \Gamma \vdash \bot \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T \ (\bot\text{-}<:)$$

$$\frac{\Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{j}} \ U_1 \qquad \Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{j}} \ U_2}{\Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{j}} \ U_1 \wedge U_2} \ (<:\text{-}\wedge) \qquad \Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T \ (<:\text{-}\textsc{Refl}) \qquad \frac{\Gamma \vdash S \ ^{\mathrm{i}}<:^{\mathrm{j}} \ T \qquad \Gamma \vdash T \ ^{\mathrm{j}}<:^{\mathrm{k}} \ U}{\Gamma \vdash S \ ^{\mathrm{i}}<:^{\mathrm{k}} \ U} \ (<:\text{-}\textsc{Trans})$$

$$\Gamma \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T_1 \vee T_2 \ (<:\text{-}\vee_1) \qquad \Gamma \vdash T_2 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T_1 \vee T_2 \ (<:\text{-}\vee_2) \qquad \frac{\Gamma \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ U \qquad \Gamma \vdash T_2 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ U}{\Gamma \vdash T_1 \vee T_2 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ U} \ (\vee\text{-}<:)$$

$$\Gamma \vdash (S \vee T) \wedge U \ ^{\mathrm{i}}<:^{\mathrm{i}} \ (S \wedge U) \vee (T \wedge U) \ (\textsc{Distr}\text{-}\wedge\text{-}\vee)$$

Later types

$$\Gamma \vdash \triangleright T \ ^{\mathrm{i}}<:^{\mathrm{i+1}} \ T \ (\textsc{Later}\text{-}<:) \qquad \Gamma \vdash T \ ^{\mathrm{i+1}}<:^{\mathrm{i}} \ \triangleright T \ (<:\text{-}\textsc{Later}) \qquad \Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \triangleright T \ (<:\text{-}\textsc{Add-Later})$$

Type members

$$\frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \{\mathsf{A} :: L \,..\, U\}}{\Gamma \vdash \triangleright L \ ^{\mathrm{i}}<:^{\mathrm{i}} \ p.\mathsf{A}} \ (<:\text{-}\textsc{Sel}) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \{\mathsf{A} :: L \,..\, U\}}{\Gamma \vdash p.\mathsf{A} \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \triangleright U} \ (\textsc{Sel}\text{-}<:)$$

Recursive types

$$\frac{\Gamma, x : \triangleright^{i} T_1 \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ T_2}{\Gamma \vdash \mu x.\, T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ \mu x.\, T_2} \ (\mu\text{-}<:\text{-}\mu) \qquad \frac{x \notin T}{\Gamma \vdash \mu x.\, T \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T} \ (\mu\text{-}<:) \qquad \frac{x \notin T}{\Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \mu x.\, T} \ (<:\text{-}\mu)$$

Co/contra-variant subtyping

$$\frac{\Gamma \vdash T \ ^{\mathrm{i}}<:^{\mathrm{i}} \ U}{\Gamma \vdash \{\mathsf{a} : T\} \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \{\mathsf{a} : U\}} \ (\textsc{Fld}\text{-}<:\text{-}\textsc{Fld}) \qquad \frac{\Gamma \vdash \triangleright L_2 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \triangleright L_1 \qquad \Gamma \vdash \triangleright U_1 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \triangleright U_2}{\Gamma \vdash \{\mathsf{A} :: L_1 \,..\, U_1\} \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \{\mathsf{A} :: L_2 \,..\, U_2\}} \ (\textsc{Typ}\text{-}<:\text{-}\textsc{Typ})$$

$$\frac{\Gamma \vdash \triangleright T_2 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \triangleright T_1 \qquad \Gamma, x : \triangleright^{i+1} T_2 \vdash \triangleright U_1 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \triangleright U_2}{\Gamma \vdash \forall x : T_1.\, U_1 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ \forall x : T_2.\, U_2} \ (\forall\text{-}<:\text{-}\forall)$$

Singleton types (rule ($\textsc{Sngl}_{qp}$-<:) is derivable and thus elided)

$$\frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} q.\mathbf{type} \qquad T_1 \cong^{*}_{p:=q} T_2}{\Gamma \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T_2} \ (\textsc{Sngl}_{pq}\text{-}<:) \qquad \frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T \qquad \Gamma \vdash p.\mathbf{type} \ ^{\mathrm{i}}<:^{\mathrm{i}} \ q.\mathbf{type}}{\Gamma \vdash q.\mathbf{type} \ ^{\mathrm{i}}<:^{\mathrm{i}} \ p.\mathbf{type}} \ (\textsc{Sngl}\text{-}<:\text{-}\textsc{Sym})$$

$$\frac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} T}{\Gamma \vdash p.\mathbf{type} \ ^{\mathrm{i}}<:^{\mathrm{i}} \ T} \ (\textsc{Sngl}\text{-}<:\text{-}\textsc{Self})$$

Notable derivable subtyping rules

$$\frac{\Gamma \vdash T_1 \ ^{\mathrm{i+1}}<:^{\mathrm{j+1}} \ T_2}{\Gamma \vdash \triangleright T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ \triangleright T_2} \ (<:\text{-}\textsc{Later-Shift}) \qquad \frac{\Gamma, x : \triangleright^{i} T_1 \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ T_2}{\Gamma \vdash \mu x.\, T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ T_2} \ (\textsc{Bind-1}) \qquad \frac{\Gamma, x : \triangleright^{i} T_1 \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ T_2}{\Gamma \vdash T_1 \ ^{\mathrm{i}}<:^{\mathrm{j}} \ \mu x.\, T_2} \ (\textsc{Bind-2})$$

Fig. 7. gDOT rules for subtyping.

As an example, using delayed typing we can build derivations similar to:

$$\Gamma \vdash \triangleright (T_1 \wedge T_2) <: \triangleright T_1 \qquad \frac{\Gamma \vdash x : \mu z.\, \{\mathsf{a} : \triangleright \{\mathsf{A} :: L \,..\, z.\mathsf{b}.\mathsf{B}\}; \mathsf{b} : \{\mathsf{B} :: L' \,..\, U\}\}}{\Gamma \vdash \triangleright x.\mathsf{a}.\mathsf{A} <: \triangleright^2 x.\mathsf{b}.\mathsf{B} <: \triangleright^3 U}$$

4.2 Type Selections

The rules (<:-Sel) and (Sel-<:) for selectors derive $\Gamma \vdash \triangleright L \ {}^{\mathrm{i}}{<:}^{\mathrm{i}} \ p.\mathsf{A}$ and $\Gamma \vdash p.\mathsf{A} \ {}^{\mathrm{i}}{<:}^{\mathrm{i}} \triangleright U$ from $\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \{\mathsf{A} :: L \mathinner{..} U\}$. These rules include a guardedness restriction in the form of a later, as imposed by our semantic model. Intuitively, the model imposes this restriction because semantic types occur under a later operator ($\blacktriangleright$) in the recursive domain equation (Sec. 1.2).

The presence of the later makes rule (Sel-<:) weaker than the corresponding rule in pDOT. Luckily, we can adapt programs by inserting laters and eliminating them through our expression subsumption rule, as shown by rule (T-Sel-Unfold), which is derived as follows:

$$\dfrac{\Gamma \vdash e : p.\mathsf{A} \qquad \dfrac{\dfrac{\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} \{\mathsf{A} :: L \mathinner{..} U\}}{\Gamma \vdash p.\mathsf{A} \ {}^{\mathrm{i}}{<:}^{\mathrm{i}} \triangleright U}\ (\textsc{Sel-<:}) \qquad \dfrac{}{\Gamma \vdash \triangleright U \ {}^{\mathrm{i}}{<:}^{\mathrm{i+1}} U}}{\Gamma \vdash p.\mathsf{A} \ {}^{0}{<:}^{\mathrm{i}} \ p.\mathsf{A} \ {}^{\mathrm{i}}{<:}^{\mathrm{i+1}} U}}{\Gamma \vdash \mathbf{coerce}^{i+1} \ e : U}\ (\textsc{T-Sub})$$

We believe all pDOT programs can be adapted to gDOT in this fashion, as discussed in Sec. 9.

Dual to the rules (<:-Sel) and (Sel-<:), which only give the bounds L and U of $p : \{\mathsf{A} :: L \mathinner{..} U\}$ under a later, the rule (D-Typ-Abs) for introduction of type members $\{\mathsf{A} = T\} : \{\mathsf{A} :: L \mathinner{..} U\}$ only requires subtyping of T with respect to bounds L and U under a later.

4.3 Function Introduction

The rule (T-∀-I-Strong) enables eliminating a later from each variable in the context when introducing a function. At the core of this rule we find the judgment $\Gamma_1 \gg_{\triangleright} \Gamma_2$, which is defined as the reflexive congruence closure under $\triangleright$, $\wedge$ and $\vee$ over $\triangleright T \gg_{\triangleright} T$. While rule (D-∀) is a common special case of (T-∀-I-Strong), stripping a later from the typing contexts created by (D-Val-New) requires the additional generality of (T-∀-I-Strong).

4.4 Other Typing Rules

Distributivity. Rule (Distr-∧-∨), together with other derived rules, makes gDOT's subtyping a lattice *distributive*, helping to deal with the interaction of intersection and union types.[7] This rule revealed itself necessary in Sec. 6.3. For lack of space, we omit here other (primitive and derived) typing rules that help distribute certain type constructors over each other.

Recursive Types. gDOT supports subtyping for recursive types [Rompf and Amin 2016], via rule (μ-<:-μ) (*cf.* Rompf and Amin's rule (BindX)), and allows one to drop unused μ binders via rules (μ-<:) and (<:-μ). The latter rules enable one to derive Rompf and Amin's rule (Bind1) and their conjectured rule (Bind2). These rules are absent from WadlerFest DOT and pDOT, requiring the insertion of redundant let bindings in some programs.

Singleton Types. Unlike in pDOT, gDOT path aliasing is reflexive and symmetric, as it is in Scala. Moreover, gDOT has various rules from Scala that are missing in pDOT (and that are to the best of our knowledge not derivable there):

- Aliasing is reflexive, *i.e.,* any well-typed path aliases itself (P-Sngl-Refl).
- If p aliases q, *i.e.,* $\Gamma \vdash_{\mathbf{p}} p :^{\mathrm{i}} q.\mathbf{type}$, then q is well-typed (P-Sngl-Inv).
- Aliasing is symmetric (Sngl-<:-Sym).
- Singleton type $p.\mathbf{type}$ is a subtype of any type T of p (Sngl-<:-Self).

Using (P-Sngl-Refl) and (Sngl-<:-Self) we derive pDOT's primitive rule (P-∧-I). Using all four rules we derive (P-Sngl-Sym), which in turn gives pDOT's primitive rule ($\textsc{Sngl}_{qp}$-<:).

[7] The dual rule of (Distr-∧-∨) is derivable using a standard proof.

5 SEMANTIC SOUNDNESS

We define a semantic model using the technique of logical relations to prove type soundness of gDOT: well-typed expressions do not go wrong, *i.e.,* they are safe in the following sense:

Definition 5.1 (Safety). *A term e is* safe*, if, for all e' such that $e \to_{\mathsf{t}}^{*} e'$, term e' is not stuck, that is, e' is either a value or e' can reduce.*

Theorem 5.2 (Type soundness). *If $\varepsilon \vdash e : T$, then e is safe.*

The main ingredient of a semantic soundness proof is the semantic typing judgment $\Gamma \vDash e : T$, which expresses what programs are safe in terms of their *behavior*. The semantic typing judgment is different and more flexible than syntactic typing judgment $\Gamma \vdash e : T$, which is defined inductively and dictates what programs are safe using a fixed set of rules. First, we can prove each typing rule as a lemma. For example, the rule (D-Val) becomes the lemma:

$$\Gamma, x : \triangleright V \vDash v : T \qquad \text{implies} \qquad \Gamma \mid x : V \vDash \{a = v\} : \{a : T\}$$

Stating typing rules as lemmas on a semantic model has a tangible benefit — it enables varying existing rules and experimenting with new rules. The semantic model will suggest necessary restrictions or generalizations to make these rules sound. This is how we designed many of the new typing rules of gDOT in Sec. 4.

Second, beyond proving typing lemmas, we can prove semantic typing judgments for programs that are not syntactically well-typed, *e.g.,* for programs that make use of unsafe casts whose correctness relies on (g)DOT's support for data abstraction. In Sec. 6 we give examples of proofs for such programs, including the function `typeFromTypeRefUnsafe` from Sec. 1.1.

We now highlight interesting aspects to our semantic model. As explained in Sec. 1.2, to define the semantic typing judgment $\Gamma \vDash e : T$, one needs to define a mapping from syntactic types T to *semantic types* $\mathcal{V}[\![T]\!] \in \mathsf{SemType}$, which express what values are safe for a given type T. To model impredicative type members in (g)DOT, we can use *step-indexing* to model SemType as the solution to the recursive domain equation Eq. (Domain) (ignoring paths and primitives):

$$\mathsf{SemType} = (\mathsf{Var} \to \mathsf{SemVal}) \to \mathsf{SemVal} \to \mathsf{iProp}$$
$$\mathsf{SemVal} \cong \mathsf{Var} + \{\, \lambda x.\, e \,\} + (\mathsf{Label} \xrightarrow{\text{fin}} (\blacktriangleright \mathsf{SemType} + \mathsf{SemVal}))$$

This recursive domain equation could *in principle* be solved directly in Iris (see Sec. 1.2). However, we will solve it *indirectly*. Instead of letting values contain semantic types, we let values contain so-called *stamps*, which refer to semantic types indirectly through a *stamping context*, which in turn appears in the domain of semantic types in Iris. We can thus keep the syntax of values first-order (which aids mechanization in Coq), and reuse Iris's support for *saved predicates* [Jung et al. 2016].

This section is organized as follows. We describe the version of gDOT with stamps — called *stamped gDOT* — in Sec. 5.1, the Iris logic in Sec. 5.2, and finally our semantic model in Sec. 5.3.

5.1 Stamped gDOT

To enable reasoning about syntactic and semantic values in a uniform way, we introduce a new variant of gDOT, called *stamped gDOT*. In contrast to *unstamped gDOT* (the variant of gDOT that we have used until now), type definitions in stamped gDOT store types only *indirectly* — values store *stamps*, *i.e.,* identifiers that are mapped to types by a *stamping context* that appears in the typing judgment. In this section we present a *syntactic version* of stamped gDOT, whose stamping context maps stamps to syntactic types. In Sec. 5.3, to construct the semantic model of gDOT, we present a *semantic version*, whose stamping context maps stamps to semantic types.

$$\triangleright\text{-Intro}\quad P \vdash_I \triangleright P \qquad \triangleright\text{-Mono}\quad \frac{P \vdash_I Q}{\triangleright P \vdash_I \triangleright Q} \qquad \triangleright\text{-Impl}\quad \triangleright(P \Rightarrow Q) \vdash_I (\triangleright P \Rightarrow \triangleright Q) \qquad \text{Impl-}\triangleright\quad (\triangleright P \Rightarrow \triangleright Q) \vdash_I \triangleright(P \Rightarrow Q)$$

$$\text{Löb}\quad (\triangleright P \Rightarrow P) \vdash_I P \qquad \text{Saved-Pred-Agree}\quad (s \rightsquigarrow \varphi_1) \wedge (s \rightsquigarrow \varphi_2) \vdash_I \triangleright(\varphi_1 = \varphi_2)$$

Fig. 8. A selection of proof rules of the considered fragment of Iris.

To disambiguate between unstamped and stamped gDOT, we color the syntax of unstamped gDOT in blue, and that of stamped gDOT in purple. Unstamped and stamped gDOT share their syntax *except* for definition bodies, which are respectively:

$$\text{DefBody} \ni d ::= p \mid T \qquad\qquad \text{DefBody} \ni d ::= p \mid \sigma, s$$

No distinction is needed between unstamped and stamped types, because types cannot contain values or definitions. In stamped gDOT definitions, *stamps* $s \in \text{Stamp}$ are simply identifiers, and are accompanied by a *deferred substitution* σ that accumulates the substitutions applied to the value containing the definition. Stamps s are mapped to types T by a *stamping context* g. The syntax of deferred substitutions and stamping contexts is:

$$\text{Subst} \ni \sigma ::= \varnothing \mid \sigma, x := v \qquad\qquad \text{StampCtx} \ni g ::= \varnothing \mid g, s := T$$

The typing judgments of stamped gDOT resemble those of unstamped gDOT, but are additionally indexed by a stamping context g, which is threaded unchanged through all typing rules. Rule (D-Typ-Abs) is modified, to retrieve the type definition using stamps and deferred substitution:

$$\frac{T = g(s)[\sigma] \qquad \Gamma, x : \triangleright V \vdash_g \triangleright L \; {}^0{<:}^0 \; \triangleright T \qquad \Gamma, x : \triangleright V \vdash_g \triangleright T \; {}^0{<:}^0 \; \triangleright U}{\Gamma \mid x : V \vdash_g \{\mathsf{A} = \sigma, s\} : \{\mathsf{A} :: L \,..\, U\}}$$

The operational semantics of stamped gDOT also resembles that of unstamped gDOT. While type members do not affect the operational semantics of either gDOT version, substitutions affect type members — either directly, in unstamped gDOT, or indirectly though the deferred substitution σ, in stamped gDOT. This becomes particularly relevant in the semantic counterpart of stamped gDOT that we consider in Sec. 5.3.

The following theorem shows that any well-typed unstamped gDOT expression e can be translated into a corresponding well-typed stamped gDOT expression e', such that e are e' are in bisimulation, denoted $e \approx e'$. The bisimulation guarantees that e is safe iff e' is safe.

Theorem 5.3 (Stamping). *If $\Gamma \vdash e : T$ in unstamped gDOT, then for any stamping context g, there exists an expression e' in stamped gDOT, and stamping context $g' \supseteq g$ with $\Gamma \vdash_{g'} e' : T$ and $e \approx e'$.*

To combine derivations with different g, we use monotonicity of stamped typing over stamp table extension. The proof is by mutual induction on derivations of all the typing judgments: each case translates subexpressions in sequence and adds entries to the stamp context. Deferred substitutions are initialized to the identity.

5.2 The Iris Logic

The Iris framework provides a programming language independent separation logic, which we instantiate with the stamped gDOT language. Since (stamped) gDOT is a pure language, we make little use of Iris's support for separation logic, but we crucially use its support for abstract step-indexing and the corresponding tactics in Coq (see Sec. 7). For the purpose of this paper, we thus

focus on the following fragment of Iris:

$$\tau ::= \mathbf{0} \mid \mathbf{1} \mid \mathsf{iProp} \mid \blacktriangleright\tau \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau \mid \mathsf{Expr} \mid \mathsf{Val} \mid \ldots$$
$$t, u, P, Q, \varphi ::= x \mid \lambda x : \tau.\, t \mid t(u) \mid \mathsf{False} \mid \mathsf{True} \mid t =_\tau u \mid P \wedge Q \mid P \vee Q \mid P \Rrightarrow Q$$
$$\mid \exists\, x : \tau.\ P \mid \forall\, x : \tau.\ P \mid \triangleright P \mid \Box P \mid \mu\, x : \tau.\ t \mid s \rightsquigarrow \varphi \mid \ldots$$

Since Iris is a higher-order logic, its grammar includes the simply-typed lambda-calculus with a number of primitive types and terms operating on these types. Most important is the type iProp of Iris propositions, and the types for all syntactical categories of stamped gDOT (*e.g.,* Expr and Val). The Iris typing judgment is mostly standard and can be derived from the use of meta variables — τ ranges over Iris types, P and Q range over Iris propositions, and t and u range over Iris terms of any type. The fragment of Iris we consider includes *higher-order impredicative quantification* ($\exists\, x : \tau.\ P$ and $\forall\, x : \tau.\ P$), the *later modality* ($\triangleright$) for abstract step-indexing, and *guarded fixpoints* ($\mu\, x : \tau.\ t$). Readers unfamiliar with Iris can ignore the *persistence modality* $\Box$ — we use it to ensure that all our definitions obey the laws of intuitionistic logic, even though full Iris is a substructural logic. The connective $s \rightsquigarrow \varphi$ for *saved predicates* [Jung et al. 2016] is used for modeling semantic stamping contexts. We describe saved predicates in Sec. 5.2.3.

We write $P \vdash_{\mathrm{I}} Q$ when P entails Q in Iris. A selection of rules of the fragment of Iris we consider is displayed in Fig. 8. Since we restrict ourselves to a fragment of Iris (with just saved predicates, and not its full support for ghost state), we additionally get the rule (Impl-$\triangleright$), which does not hold in full Iris. We need this rule for proving the semantic typing lemmas corresponding to the contravariant subtyping rules (Typ-<:-Typ) and ($\forall$-<:-$\forall$).

5.2.1 Expression Weakest Preconditions. Support for reasoning about programs (using weakest preconditions or Hoare triples) is not hard-wired into Iris. Instead, using Iris one can define custom reasoning principles for the program language in question [Krebbers et al. 2017a]. Since gDOT is a pure language, we define a custom notion of pure weakest preconditions for stamped gDOT expressions (in this section, we only consider stamped gDOT, so we omit syntax coloring):

$$\mathsf{wp}\ e\ \{\varphi\} \triangleq \begin{cases} \varphi(e) & \text{if } e \in \mathsf{Val} \\ (\exists\, e'.\, e \longrightarrow_{\mathrm{t}} e') \wedge (\forall\, e'.\, e \longrightarrow_{\mathrm{t}} e' \Rrightarrow \triangleright\, \mathsf{wp}\ e'\ \{\varphi\}) & \text{otherwise} \end{cases}$$

Intuitively, $\mathsf{wp}\ e\ \{\varphi\}$ asserts that e is safe, and any resulting value v of e satisfies $\varphi(v)$. We write $\mathsf{wp}\ e\ \{v.\ P\}$ as shorthand for $\mathsf{wp}\ e\ \{\lambda\, v.\ P\}$. Like the standard Iris weakest precondition for stateful languages, the above definition is formalized using the Iris guarded fixpoint operator $\mu x : \tau.\, t$. Weakest preconditions enjoy the following rules:

$$\phi(v) \vdash_{\mathrm{I}} \mathsf{wp}\ v\ \{\phi\}\varphi \qquad \text{(Wp-Val)}$$
$$(e_1 \longrightarrow_{\mathrm{t}} e_2) \wedge \triangleright\, \mathsf{wp}\ e_2\ \{\varphi\} \vdash_{\mathrm{I}} \mathsf{wp}\ e_1\ \{\varphi\} \qquad \text{(Wp-Step)}$$

The later modality ($\triangleright$) in the (Wp-Step) enables eliminating a later in all hypotheses each time we take a program step. This is crucial for proving partial program correctness using (Löb) induction, which allows proving P while assuming the induction hypothesis $\triangleright P$ — using (Wp-Step) and ($\triangleright$-Mono) one can turn the induction hypothesis $\triangleright P$ into just P.

5.2.2 Path Weakest Preconditions. While weakest preconditions for expressions ensure partial correctness, weakest preconditions for paths ensure total correctness (*i.e.,* normalization):

$$\mathsf{wp_P}\ p\ \{\varphi\} \triangleq \begin{cases} \varphi(p) & \text{if } p \in \mathsf{Val} \\ \exists\, v_q, q'.\ \mathsf{wp_P}\ q\ \{v.\ v = v_q\} \wedge v_q.\mathsf{a} \searrow q' \wedge \mathsf{wp_P}\ q'\ \{\varphi\} & \text{if } p = q.\mathsf{a} \end{cases}$$

Intuitively, $\mathsf{wp}_{\mathsf{P}}\ p\ \{\varphi\}$ asserts that p normalizes to a value v satisfying $\varphi(v)$. We write $\mathsf{wp}_{\mathsf{P}}\ p\ \{v.\ P\}$ as shorthand for $\mathsf{wp}_{\mathsf{P}}\ p\ \{\lambda\, v.\ P\}$. Similarly to Iris's notion of *total* weakest preconditions, this definition does not include a later modality ($\triangleright$), and is formalized as a least fixpoint.

5.2.3 Saved Predicates. To model semantic stamping contexts in Sec. 5.3, which map stamps to semantic types, we make use of Iris's support for saved predicates [Jung et al. 2016]. Saved predicates appear in the logic through the connective $s \rightsquigarrow \varphi$, where s is an identifier, and φ is an Iris predicate. They enjoy the rule (Saved-Pred-Agree): if an identifier maps to two predicates, they are equal. This equality appears under a later modality ($\triangleright$) because saved predicates can refer to themselves, and such self-references must be guarded through a later modality ($\triangleright$) to be sound [Jung et al. 2018b, Sec. 3.3]. We do not show the rules for introducing saved predicates, since they require a bigger fragment of Iris than we describe in this paper. However, as we will discuss in Sec. 5.3.3, when proving that a closed program is safe, we can introduce any number of $s \rightsquigarrow \varphi$ connectives.

In our semantic model, we instantiate Iris's generic saved predicate construction with semantic types, *i.e.,* with predicates over stamped gDOT environments and stamped gDOT values:

$$\mathsf{SemType} \triangleq (\mathsf{Var} \to \mathsf{Val}) \to \mathsf{Val} \to \mathsf{iProp}$$

To see how the use of saved predicates relates to the explicit model as a solution of the recursive domain equation Eq. (Domain) at page 5, let us unfold the model of Iris. The type of propositions of (our fragment of) Iris is the solution to the following recursive domain equation:

$$\mathsf{iProp} = \textsc{Auth}(\mathsf{Stamp} \xrightarrow{\text{fin}} \textsc{Ag}(\blacktriangleright \mathsf{SemType})) \to \mathsf{siProp}$$

Here, siProp is the type of step-indexed propositions, and Auth and Ag are Iris's authoritative and agreement cameras used to describe the allowed resource sharing. Most of these details can be ignored; what matters is that SemType, which contains a recursive occurrence of iProp, appears under a later type former ($\blacktriangleright$). As such, the model of Iris with saved predicates is isomorphic (modulo the indirection via stamps) to a direct model of gDOT as described by Eq. (Domain).

5.3 The Semantic Model of gDOT

We now put Iris and stamped gDOT to work by proving type soundness of gDOT (Theorem 5.2). In Sec. 5.3.1 we define a semantic typing judgment $\Gamma \vDash_G e : T$ for stamped gDOT, and in Sec. 5.3.2 we prove that syntactically well-typed stamped gDOT terms are also semantically well-typed:

Theorem 5.4 (Fundamental). *If $\Gamma \vdash_g e : T$, then $\Gamma \vDash_{\overline{\mathcal{V}}[\![g]\!]} e : T$, where $\overline{\mathcal{V}}[\![g]\!]$ is the semantic stamping context corresponding to the syntactic stamping context g.*

Finally, in Sec. 5.3.3 we prove adequacy of semantic typing for stamped gDOT:

Theorem 5.5 (Adequacy). *If $\varepsilon \vDash_G e : T$, then e is safe.*

Putting these theorems together, we obtain type soundness of unstamped gDOT (Theorem 5.2) as a corollary, *i.e.,* if $\varepsilon \vdash e : T$, then e is safe.

Proof of Theorem 5.2. Assume $\varepsilon \vdash e : T$ for an unstamped e. By Theorem 5.3 we get $\varepsilon \vdash_g e' : T$ for a stamped term e' with $e \approx e'$, and a syntactic stamping context g. By Theorem 5.4 we obtain $\varepsilon \vDash_{\overline{\mathcal{V}}[\![g]\!]} e' : T$. By Theorem 5.5 we obtain that e' is safe, which by $e \approx e'$ gives that e is safe. □

Since the rest of this section is concerned with stamped gDOT, we omit syntax coloring.

Auxilary definitions

$$\mathsf{wellMapped}(G) \triangleq \forall s \in \mathrm{dom}(G).\ s \rightsquigarrow G(s)$$

$$s \rightsquigarrow_\sigma \psi \triangleq \exists \varphi.\ (s \rightsquigarrow \varphi) \wedge \triangleright(\psi = \varphi\ (\sigma))$$

Definition interpretation $\boxed{\overline{\mathcal{D}}[\![\_]\!]\_(\_) : \mathsf{Type} \to \mathsf{Env} \to \mathsf{DefList} \to \mathsf{iProp}}$

$$\overline{\mathcal{D}}[\![\top]\!]_\rho(\bar{d}) \triangleq \mathsf{True}$$

$$\overline{\mathcal{D}}[\![S \wedge T]\!]_\rho(\bar{d}) \triangleq \overline{\mathcal{D}}[\![S]\!]_\rho(\bar{d}) \wedge \overline{\mathcal{D}}[\![T]\!]_\rho(\bar{d})$$

$$\overline{\mathcal{D}}[\![\{\mathsf{a} : T\}]\!]_\rho(\bar{d}) \triangleq \exists p.\ \mathsf{lookup}(\mathsf{a}, \bar{d}) = p \wedge \mathsf{wp_P}\ p\ \{\mathcal{V}[\![T]\!]_\rho\}$$

$$\overline{\mathcal{D}}[\![\{\mathsf{A} :: S\,..\,U\}]\!]_\rho(\bar{d}) \triangleq \exists \sigma, s, \psi.\ \mathsf{lookup}(\mathsf{A}, \bar{d}) = (\sigma, s) \wedge (s \rightsquigarrow_\sigma \psi) \wedge \big(\forall v.\ \triangleright\mathcal{V}[\![S]\!]_\rho(v) \Rightarrow \triangleright\Box\psi(v)\big) \wedge \big(\forall v.\ \triangleright\Box\psi(v) \Rightarrow \triangleright\mathcal{V}[\![U]\!]_\rho(v)\big)$$

$$\overline{\mathcal{D}}[\![T]\!]_\rho(\bar{d}) \triangleq \mathsf{False} \qquad (\text{if } T \text{ is not } \top, S \wedge T, \{\mathsf{A} :: S\,..\,U\}, \text{or } \{\mathsf{a} : T\})$$

Value interpretation $\boxed{\mathcal{V}[\![\_]\!]\_(\_) : \mathsf{Type} \to \mathsf{Env} \to \mathsf{Val} \to \mathsf{iProp}}$

$$\mathcal{V}[\![\top]\!]_\rho(v) \triangleq \mathsf{True}$$

$$\mathcal{V}[\![\bot]\!]_\rho(v) \triangleq \mathsf{False}$$

$$\mathcal{V}[\![S \wedge T]\!]_\rho(v) \triangleq \mathcal{V}[\![S]\!]_\rho(v) \wedge \mathcal{V}[\![T]\!]_\rho(v)$$

$$\mathcal{V}[\![S \vee T]\!]_\rho(v) \triangleq \mathcal{V}[\![S]\!]_\rho(v) \vee \mathcal{V}[\![T]\!]_\rho(v)$$

$$\mathcal{V}[\![\forall x : S.\, T]\!]_\rho(v) \triangleq \exists e.\ (v =_\alpha \lambda x.\, e) \wedge \Box\ \big(\forall w.\ \triangleright\mathcal{V}[\![S]\!]_\rho(w) \Rightarrow \triangleright\mathcal{E}[\![T]\!]_{(\rho, x := w)}(e[x := w])\big)$$

$$\mathcal{V}[\![\{\mathsf{a} : T\}]\!]_\rho(v) \triangleq \exists x, \bar{d}.\ (v =_\alpha \nu x.\, \{\bar{d}\}) \wedge \overline{\mathcal{D}}[\![\{\mathsf{a} : T\}]\!]_\rho(\bar{d}[x := v])$$

$$\mathcal{V}[\![\{\mathsf{A} :: S\,..\,U\}]\!]_\rho(v) \triangleq \exists x, \bar{d}.\ (v =_\alpha \nu x.\, \{\bar{d}\}) \wedge \overline{\mathcal{D}}[\![\{\mathsf{A} :: S\,..\,U\}]\!]_\rho(\bar{d}[x := v])$$

$$\mathcal{V}[\![p.\mathsf{A}]\!]_\rho(v) \triangleq \mathsf{wp_P}\ p[\rho]\ \{w.\ \exists \sigma, s, \psi.\ (w.\mathsf{A} \searrow (\sigma, s)) \wedge (s \rightsquigarrow_\sigma \psi) \wedge \triangleright\Box\,\psi(v)\}$$

$$\mathcal{V}[\![p.\mathbf{type}]\!]_\rho(v) \triangleq \mathsf{wp_P}\ p[\rho]\ \{w.\ v =_\alpha w\}$$

$$\mathcal{V}[\![\mu x.\, T]\!]_\rho(v) \triangleq \mathcal{V}[\![T]\!]_{(\rho, x := v)}(v)$$

$$\mathcal{V}[\![\triangleright T]\!]_\rho(v) \triangleq \triangleright\mathcal{V}[\![T]\!]_\rho(v)$$

Expression interpretation $\boxed{\mathcal{E}[\![\_]\!]\_(\_) : \mathsf{Type} \to \mathsf{Env} \to \mathsf{Expr} \to \mathsf{iProp}}$

$$\mathcal{E}[\![T]\!]_\rho(e) \triangleq \mathsf{wp}\ e\ \{\mathcal{V}[\![T]\!]_\rho\}$$

Environment interpretation $\boxed{\mathcal{G}[\![\_]\!](\_) : \mathsf{TyCtx} \to \mathsf{Env} \to \mathsf{iProp}}$

$$\mathcal{G}[\![\varepsilon]\!](\rho) \triangleq \mathsf{True}$$

$$\mathcal{G}[\![\Gamma, x : T]\!](\rho) \triangleq \mathcal{G}[\![\Gamma]\!](\rho_{|\Gamma}) \wedge \mathcal{V}[\![T]\!]_\rho(\rho(x))$$

Semantic typing judgments

$$\Gamma \vDash_{\mathbf{P}} p :^{\mathsf{i}} T \triangleq \Box\ \big(\forall \rho.\ \mathcal{G}[\![\Gamma]\!](\rho) \Rightarrow \triangleright^i\, \mathsf{wp_P}\ p[\rho]\ \{\mathcal{V}[\![T]\!]_\rho\}\big)$$

$$\Gamma \vDash e : T \triangleq \Box\ \big(\forall \rho.\ \mathcal{G}[\![\Gamma]\!](\rho) \Rightarrow \mathcal{E}[\![T]\!]_\rho(e[\rho])\big)$$

$$\Gamma \mid x : V \vDash \{\bar{d}\} : T \triangleq \mathsf{wf}\ \bar{d} \wedge \Box\ \big(\forall \rho, \bar{d}_v.\ \mathsf{wf}\ \bar{d}_v \Rightarrow (\bar{d} \subseteq \bar{d}_v[x := \nu x.\, \{\bar{d}_v\}]) \Rightarrow \mathcal{G}[\![\Gamma, x : V]\!](\rho, x := \nu x.\, \{\bar{d}_v\}) \Rightarrow \overline{\mathcal{D}}[\![T]\!]_\rho(\bar{d}[\rho])\big)$$

$$\Gamma \vDash T_1\ {}^{\mathsf{i}}{<:}^{\mathsf{j}}\ T_2 \triangleq \Box\ \big(\forall \rho, v.\ \mathcal{G}[\![\Gamma]\!](\rho) \Rightarrow \triangleright^i \mathcal{V}[\![T_1]\!]_\rho(v) \Rightarrow \triangleright^j \mathcal{V}[\![T_2]\!]_\rho(v)\big)$$

$$\Gamma \vDash_G e : T \triangleq \Box(\mathsf{wellMapped}(G) \Rightarrow \Gamma \vDash e : T)$$

Fig. 9. The semantic model of gDOT. The entire figure is phrased on *stamped* syntax. Relation wf $\bar{d}$ asserts that $\bar{d}$ contains no duplicate labels. Environment restriction $\rho_{|\Gamma}$ restricts ρ to entries in Γ.

5.3.1 Semantic Typing Judgments. Fig. 9 shows our semantic model for gDOT. Its definition follows the conventional setup of a logical relations model. First we define interpretation relations $\overline{\mathcal{D}}[\![\, T \,]\!]_\rho(\bar{d})$, $\mathcal{V}[\![\, T \,]\!]_\rho(v)$, and $\mathcal{E}[\![\, T \,]\!]_\rho(e)$ that describe the closed definition lists $\bar{d}$, closed values v, and closed expressions e that safety inhabit a type T, under an environment $\rho \in \mathsf{Env} \triangleq \mathsf{Var} \to \mathsf{Val}$ that gives the interpretation of the variables in T. These interpretation relations are defined by structural recursion on types T. Second, we lift these interpretation relations using closing substitutions to the various semantic typing judgments for open terms.

Before we detail the interpretation relations and semantic typing judgments, we describe how stamps are handled. Using Iris's saved predicates we define $s \rightsquigarrow_\sigma \psi$, which says that stamp s and deferred substitution σ map to semantic type $\psi \in \mathsf{SemType} \triangleq \mathsf{Env} \to \mathsf{Val} \to \mathsf{iProp}$. Instead of threading *semantic stamping contexts* $G \in \mathsf{Stamp} \xrightarrow{\text{fin}} \mathsf{SemType}$ through all semantic typing judgments, we keep them implicit using $\mathsf{wellMapped}(G)$, which asserts that G is reflected through Iris's $\rightsquigarrow$. Only the top-level judgment $\Gamma \vDash_G e : T$ makes the stamping context G explicit.

The value interpretations of the basic types are standard for a logical relations model. The interpretations of $\bot$, $\top$, $\wedge$, $\vee$ and $\triangleright$ use the corresponding logical connectives of Iris.

The value interpretation of function types $\mathcal{V}[\![\, \forall\,(x : S).\, T \,]\!]_\rho(v)$ expresses that v is α-equivalent to a function $\lambda x.\, e$ that maps values w of type S into expressions $e[x := w]$ of type T. The latter is expressed by the expression interpretation $\mathcal{E}[\![\, T \,]\!]_{(\rho, x:=w)}(e[x := w])$, which is defined using weakest preconditions as is standard for logical relations in Iris. However, since gDOT supports dependent functions (where the argument x is in scope in type T), we interpret T in the extended context $\rho, x := w$. Moreover, we use the later modality ($\triangleright$), like in step-indexed logical relations for *equi-recursive* types, where type constructors must be *contractive* rather than *non-expansive* [Appel and McAllester 2001]. This choice provides stronger typing rules, and is for instance the reason why (T-∀-I-Strong) can strip a later ($\triangleright$) from gDOT's typing context.

The value interpretation of record types $\mathcal{V}[\![\, \{\mathsf{a} : U\} \,]\!]_\rho(v)$ and $\mathcal{V}[\![\, \{\mathsf{A} :: L \,..\, U\} \,]\!]_\rho(v)$ expresses that v is α-equivalent to an object $vx.\, \{\bar{d}\}$ with value member a (or type member A) that enjoys the right property. The latter is expressed using the interpretation $\overline{\mathcal{D}}[\![\, \_ \,]\!]_\rho(\bar{d})$ for definition lists $\bar{d}$. We highlight the interpretation of type members $\overline{\mathcal{D}}[\![\, \{\mathsf{A} :: S \,..\, U\} \,]\!]_\rho(\bar{d})$. Since values only store semantic types ψ that are guarded through saved predicates in Iris, we only refer to ψ under the later modality ($\triangleright$). This definition prevents bad bounds by ensuring that ψ respects its bounds S and U: for instance, $\mathcal{V}[\![\, \{\mathsf{A} :: \top \,..\, \bot\} \,]\!]_\rho(v)$ is false. Therefore, objects with bad bounds cannot be typed in the empty context, and unsound subtyping evidence cannot be constructed.

The value interpretation of abstract types $\mathcal{V}[\![\, p.\mathsf{A} \,]\!]_\rho(v)$ describes that p normalizes to object w, which in field $w.\mathsf{A}$ holds stamp s that refers to semantic type ψ. Since (paths in) types are able to contain variables, this definition substitutes ρ in p, and then uses the path weakest precondition to reason about the resulting object w. It asserts that v, under the later modality ($\triangleright$), satisfies the semantic type ψ. Similarly, the interpretation of singleton types $\mathcal{V}[\![\, p\,.\mathbf{type} \,]\!]_\rho(v)$ normalizes p to w, and checks that v and w coincide.

The value interpretation of μ-types $\mathcal{V}[\![\, \mu(x : T) \,]\!]_\rho(v)$ interprets T in the extended environment $\rho, x := v$, matching the informal semantics.

Using the interpretation relations we define the semantic typing judgments. For instance, the semantic expression typing judgment $\Gamma \vDash e : T$ asserts that e runs safely in any environment ρ matching Γ, and results in a value satisfying $\mathcal{V}[\![\, T \,]\!]$. The semantic path typing judgment $\Gamma \vDash_{\mathbf{p}} p :^{\mathsf{i}} T$ is defined using path weakest preconditions, so it asserts that path p normalizes to a value satisfying $\mathcal{V}[\![\, T \,]\!]$. Unlike expression typing, path typing thus does not allow its subject to diverge (unlike in pDOT, paths in gDOT cannot loop; see Sec. 8 for further discussion).

5.3.2 Semantic Typing Lemmas. After having defined the semantic typing judgments, we can prove the *semantic typing lemmas*. Basically, for each typing rule, we replace $\vdash_g$ with $\vDash$, and prove the result as a lemma. In fact, while designing gDOT, what we did was exactly the opposite — we first proved the semantic typing lemmas before turning gDOT into a syntactic type system.

The proofs of the bulk of the semantic type lemmas are fairly straightforward — the majority of the work was in devising the right interpretations of types. Two typing rules with interesting proofs are (Sel-<:), and (T-{}-I). Rule (T-{}-I) is interesting because its proof relies on (Löb) induction: to prove that the object $v \triangleq \nu x.\, \{\bar{d}\}$ satisfies $\mathcal{V}[\![\, \mu x.\, T \,]\!]_\rho(v)$, we can assume it satisfies $\triangleright(\mathcal{V}[\![\, \mu x.\, T \,]\!]_\rho(v))$. The proof of rule (Sel-<:) is interesting because it uses Iris's proof rule (Saved-Pred-Agree) for saved propositions. This proof also explains the $\triangleright$ in the rule (Sel-<:) — it appears in rule (Sel-<:) because it appears in (Saved-Pred-Agree). In general, the proofs of the semantic typing rules led to the insight that types and function bodies only contain information later, hence introduction and elimination rules only require information under a later type former.

As expected for a semantic model based on logical relation, putting together the semantic typing lemmas, we prove the *fundamental property* (Theorem 5.4), *i.e.,* if $\Gamma \vdash_g e : T$, then $\Gamma \vDash_{\overline{\mathcal{V}}[\![g]\!]} e : T$.

Proof of Theorem 5.4. The theorem is proved by mutual induction on all typing judgments that make up our type system. Each case of the proof follows directly from the semantic typing lemmas that have been proved for each syntactic typing rule. □

5.3.3 Adequacy of semantic typing. We outline the proof of the adequacy theorem of our semantic typing judgment (Theorem 5.5), *i.e.,* if $\vDash_G e : T$, then e is safe.

Proof of Theorem 5.5. To use the judgment $\vDash_G e : T$, we must prove its premise $\mathsf{wellMapped}(G)$ by initializing the saved predicates.[8] This gives $\mathsf{wp}\ e\ \{\mathcal{V}[\![\, T \,]\!]_\rho\}$, which by adequacy of pure weakest preconditions, shows that e is safe. The adequacy proof of pure weakest preconditions resembles the adequacy proof of stateful weakest preconditions in Iris [Krebbers et al. 2017a]. □

6 EXPRESSIVITY EVALUATION

We show that, despite gDOT's guardedness restrictions, we can encode both existing and new examples from the literature. All examples presented in this section, and additional ones, including all examples in Sec. 1-5 of the WadlerFest DOT paper [Amin et al. 2016], are mechanized in Coq. In Sec. 6.1 we describe the syntactic typing of an encoding of covariant lists, a highly recursive benchmark from the DOT literature. In Sec. 6.2 and Sec. 6.3 we show that gDOT enforces data abstraction: we semantically type two programs whose safety relies on class invariants, including our motivating example from the introduction (Sec. 1.1).

6.1 Covariant Lists

As it is standard in the DOT literature [Amin et al. 2016; Rapoport and Lhoták 2019; Rompf and Amin 2016], we encode the Scala type `List[T]` of lists, together with its core methods. Our encoding, which is shown in Fig. 10, is mostly standard in DOT (except for the shaded parts, to which we return in a moment), but we summarize a few features of this encoding. Object *lists* defines an abstract type of lists List, together with constructors nil and cons. In turn, the type of lists defines a type member A representing the type of elements, together with accessor methods. The definition of *lists*.List is highly recursive: it uses self variables *lists* and *list* to refer to both itself and its own type member *list*.A. Since DOT lacks exceptions, here and in later examples we let failing methods invoke an infinite loop **diverge** with type $\bot$, like in other DOT papers [Amin et al. 2016; Rapoport

[8] This proof uses a bigger Iris fragment than shown in Sec. 5.2; we omit details in the paper.

```
let bools = ... in let lists = v lists. {
  List   >: ⊥ = μ list.
         {A >: ⊥ <: ⊤; isEmpty : ⊤ → bools.Bool; head : ⊤ → list.A; tail : ⊤ → lists.List ∧ {A <: list.A}}
  nil    : ▷ lists.List ∧ {A = ⊥}
         = v _. {A = ⊥; isEmpty = λ_. bools.true; head = λ_. diverge; tail = λ_. diverge}
  cons   : ∀ (x : {S <: ⊤}). x.S → (lists.List ∧ {A <: x.S}) → lists.List ∧ {A <: x.S}
         = λ x hd tl. v _. {A = x.S; isEmpty = λ_. bools.false; head = λ_. hd; tail = λ_. tl}
} in ...
```

Fig. 10. Covariant lists in gDOT using (elided for space) Church-encoded Booleans [Amin et al. 2016].

```
posSemT ∈ SemType  ≜ λ ρ, v. ∃ n : ℤ. v = n ∧ n > 0
  let positives    = v positives. {
    Pos            >: ⊥ <: Int = posSemT
    mkPos          : Int → positives.Pos = λ m. if m > 0 then m else diverge
    div            : Int → positives.Pos → Int = λ m n. m/(coerce n)
  } in ...
```

Fig. 11. A module for positive numbers and safe division using abstract types.

and Lhoták 2019; Rompf and Amin 2016]. We encode the Scala type `List[T]` as *lists*.List ∧ {A <: *T*}. Similarly to lists in Scala, this encoding is *covariant*, *i.e.,* if T_1 `<:` T_2 then `List[`$T1$`] <: List[`T_2`]`.

Type checking the body of cons relies on gDOT's rules (μ-<:) and (<:-μ) for subtyping of recursive types [Rompf and Amin 2016]. Since most other DOT variants [Amin et al. 2016; Rapoport et al. 2017; Rapoport and Lhoták 2019] do not support those rules, they require instead modifying the source code and inserting a spurious let-binding, to then use rules analogous to (P-μ-I) and (P-μ-E).

The only unusual guardedness restriction of gDOT is the use of a later in front of the type of nil. Since nil is defined as a value member, we cannot use coercions; and since the type of self variable *lists* is guarded during construction, we cannot derive bounds for *lists*.List but only for ▷ *lists*.List. This restriction could be avoided by thunking nil (*i.e.,* making it a method). In the present encoding, the later can be removed at the client side via a **coerce**.

6.2 Positive Numbers

A syntactic type system (like gDOT's) cannot recognize all semantically safe programs because safety may rely on functional correctness or the language's support for data abstraction. Similarly to the RustBelt model of Rust [Jung et al. 2018a], our semantic model of gDOT enables proving the safety of such examples by dropping down to the definition of semantic typing in Iris. In this and the next section, we discuss two such examples.

As a warm-up, and to demonstrate the use of semantic types, we show that object *positives* in Fig. 11 is semantically well-typed. This object defines a type member Pos of positive numbers, and safe methods to create (mkPos), and consume them (div). The method mkPos represents a "smart constructor": it returns the input number *m* if positive, and fails by looping otherwise. The method div makes use of an *unsafe* division operator, which gets stuck in the operational semantics when called with the argument 0.[9] Since safety of the division operator relies on functional correctness

[9]Whereas we used Church encoded Booleans in Sec. 6.1, the version of gDOT that we mechanized in Coq in fact has primitive support for Booleans and integers, which we use in this section.

```
assert c ≜ if c then 0 else diverge;
None ≜ {isEmpty : true.type; ...}
Some ≜ μ some. {
  A          >: ⊥ <: ⊤
  isEmpty    : false.type   // The only value in singleton type false.type is false.
  pmatch     : ∀(x : {U <: ⊤}). x.U → (some.A → x.U) → x.U
  get        : ▷ some.A
}
let options : {Option <: None ∨ Some} = ... in
let pcore = v pcore. {
  types = v types. {
    Type              >: ⊥ = ⊤
    TypeTop           >: ⊥ = types.Type
    newTypeTop        : ⊤ → types.TypeTop = λ_. v_. {}
    TypeRef           >: ⊥ <: types.Type ∧ {symb : pcore.symbols.Symbol}
                      = types.Type ∧ {symb : (pcore.symbols.Symbol ∧ {tpe : Some})}
    newTypeRef        : pcore.symbols.Symbol → types.TypeRef
                      = λs. { assert(¬(coerce s).tpe.isEmpty); v_. {symb = s} }
    typeFromTypeRef   : types.TypeRef → types.Type =
                      = λt. { coerce² (coerce (coerce t).symb).tpe.get }
  } // symbols is unchanged
} in ...
```

Fig. 12. The (simplified) fragment of Dotty from Fig. 1 in gDOT (**assert**, **None**, and **Some** are abbreviations).

(*i.e.,* the argument being non-zero), it cannot be typed in the syntactic type system of (g)DOT. In turn, object *positives* cannot be typed syntactically.

Yet, we can prove that *positives* is *semantically typed*. While the class invariant of Pos cannot be expressed through a syntactic type, we can express it as the logical predicate *posSemT* in stamped gDOT.[10] To prove semantic typing of *positives*, we unfold the semantic typing judgment of our gDOT model and perform a manual proof in Iris. Let us walk through that proof. First, we need to prove that Pos respects its type bounds ⊥ and `Int`. This holds trivially because positive integers are integers. Next, we should prove semantic typing of mkPos. For that, we need to show that if the conditional succeeds, the argument satisfies *posSemT*. Finally, we should prove semantic typing of div. Since its argument *n* has abstract type *positives*.Pos, it satisfies semantic type *posSemT*. Note that since type members are modeled using saved predicates in Iris, method div uses a coercion, allowing us to strip a later when acquiring the semantic type *posSemT* of type member *positives*.Pos.

Thanks to the Iris framework, we do not have to deal with explicit step-indexing. All proofs are carried out using Iris's support for abstract step-indexing. Moreover, to streamline semantic typing proofs such as the above, we have generalized semantic typing judgments to semantic types in our Coq mechanization, which enable us to reuse our typing lemmas both here and in Sec. 6.3.

6.3 Mutual Information Hiding

We now return to our motivating example from the introduction (Sec. 1.1). As discussed in Sec. 4, we have shown in Coq that the gDOT version (Fig. 2) of the Scala code (Fig. 1) is syntactically well-typed. However, method `typeFromTypeRefUnsafe` (which is present in the Scala version, but not

[10] The definition of Pos is encoded using a stamp and an identity deferred substitution.

in the DOT version), cannot be shown to be syntactically well-typed in any DOT calculus — it uses an unsafe cast whose safety crucially relies on (g)DOT's support for data abstraction. Using gDOT's semantic model we show that this example is in fact semantically well-typed. This demonstrates the flexibility of semantic typing, and shows that gDOT enforces the data abstraction that mutual information hiding should provide.

Method `typeFromTypeRefUnsafe` in Fig. 1 retrieves an actual `types.Type` by invoking the `get` method on `t.symb.tpe`, which has type `Option[types.Type]`. In Scala, invoking `Option`[T]'s method `get` on `None` will trigger an exception. However, since gDOT (like all DOT calculi) lacks exceptions, and to show gDOT's support for data abstraction, we model (unlike the Scala standard library) `get` as a function from `Some`[T] to T, where `Some`[T] is a subtype of `Option`[T] that has a `get` function. Hence, to call `get`, we first must unsafely cast `tpe` via `tpe.asInstanceOf[Some[types.Type]]` to get a value of type `Some[types.Type]`. Although this cast cannot be typed syntactically, it is safe due to the `assert` in constructor `TypeRef`. In turn, safety of this `assert` relies on `Option`'s class invariant: `isEmpty` only returns `false` on instances of the `Some` constructor.

We encode the Scala example from Fig. 1 in gDOT as shown in Fig. 12. As usual in gDOT, we use coercions when unfolding abstract types to ensure guardedness. More importantly, we express the class invariants of types *options*.Option and pcore.types.TypeRef by defining them to stricter types than in Scala. In particular, the upper bound of Option formalizes the informal invariants of *options*'s public API using union and singleton types. An instance of Option is then either an instance of **None**, exposing an isEmpty method that returns **true**, or an instance of **Some**, exposing an isEmpty method that returns **false** and a get method that returns the contained value.

Thanks to gDOT's support for mutual information hiding, one can also expose class invariants locally, and hide them by using subsumption. This is used for TypeRef, whose class invariant guarantees that symb.tpe has type **Some** containing method get, but this is hidden outside types. The more precise definition of TypeRef makes typeFromTypeRef syntactically well-typed (hence the change of name), but makes newTypeRef syntactically ill-typed, so we prove it well-typed *semantically*. In this proof, we take the result v of (**coerce** s).tpe, show it has type **None** $\vee$ **Some**, and reason by cases on this union type. If v has type **None**, newTypeRef diverges and is thus safe. If v has type **Some**, the return value of newTypeRef will have the correct type TypeRef. This concludes our informal proof sketch, which we have mechanized in Coq. Parts of the typing derivation are constructed syntactically; in those parts, we needed various distributivity rules, including rule (Distr-$\wedge$-$\vee$) to distribute intersections over union and turn intersection (**None** $\vee$ **Some**) $\wedge$ {A :: $\bot$ .. *pcore*.types.Type} into a union type (see Sec. 4.4).

7 COQ MECHANIZATION

We mechanized gDOT and its semantic soundness proof in Coq, using the Iris framework and the MoSeL tactic language [Krebbers et al. 2018, 2017b]. We mechanized binding through de Bruijn indexes and parallel substitution, using the Autosubst 1 library [Schäfer et al. 2015]. While DOT binding does not fit perfectly with Autosubst 1 (unlike Autosubst 2 [Stark et al. 2019]), we were able to use Autosubst 1 by defining substitution by hand, while reusing Autosubst 1's tactics for deciding binding lemmas. Due to Autosubst 1's limitations, path substitution is defined separately. The only axioms we use are functional extensionality (needed by Autosubst), and uniqueness of identity proofs (to simplify dealing with dependent pattern matching).

Overall, our gDOT mechanization consists of 17.566 lines of Coq code. It includes some language-generic components (3.079 lines). The mechanization of the gDOT semantic model (4.742 lines) consists of the definition of the gDOT language and operational semantics (2.095 lines), and the logical relation, semantic typing lemmas and adequacy theorem (2.647 lines). The mechanization of the gDOT syntactic type system consists of definitions and proofs about the syntactic type system,

derived typing rules, stamping (including Theorem 5.3), and the fundamental theorem (5.552 lines). Finally, we mechanized all examples (4.193 lines).

8 RELATED WORK

pDOT. The variant of the DOT calculus that is closest to gDOT is pDOT, introduced by Rapoport and Lhoták [2019]. The rule (D-Val-New) of gDOT is an "alternative design" they considered for pDOT [Rapoport and Lhoták 2019, Sec. 4.2.2]. Unlike Scala, pDOT considers paths as normal forms. Instead, gDOT lets paths reduce, but ensures they have a normal form.

Normalization for $D_{<:}$. We were inspired by Wang and Rompf [2017], who use logical relations to prove normalization of a DOT subset including $D_{<:}$. They prove normalization (in a way that implies type safety), which they argue is important for paths. However, for proving type safety of Scala (which in itself is not normalizing), it is sufficient to prove normalization of paths only. Indeed, our path typing judgment implies normalization through the use of total weakest preconditions in Iris (see Sec. 5.2.2). Moreover, their model imposes guardedness restrictions on μ-types instead of abstract types. Those guardedness restrictions are more severe than gDOT's — they only allow for a weak elimination rule for μ-types, reminiscent of System F-style weak existentials. While their results also imply type safety for the language they study, it is unclear how to adapt their technique to prove type safety of a Turing-complete (*i.e.,* non-normalizing) variant of the language.

Coinductive Type Systems. Brandt and Henglein [1998] define subtyping for recursive types using a coinductive formulation of subtyping, which resembles our typing rule for object creation, and our use of Löb induction. That is, to prove a judgment J (such as type equality or subtyping), they allow using J as an assumption, but forbid using J immediately. One might suspect that a coinductive formulation of DOT, and of rule (T-{}-I) in particular, might allow making (D-Typ-Abs) sound without using later. However, DOT's μ-types (and Amin et al.'s refinements [2012]) differ from standard recursive types, and resemble more closely recursively defined signatures [Crary et al. 1999], Cedille's ι-types [Fu and Stump 2014], and dependent intersections [Kopylov 2003].

Logical relations for Predicative Type Members. Logical relation models are available for other type systems with features similar to Scala type members, such as ML modules [Crary 2017] and type theory. However, such type systems avoid the challenges we face because they feature a universe hierarchy and predicative/stratified type members/Σ-types: if a value v contain a type in a certain universe, the type of v lives in a larger universe [Harper and Mitchell 1993], eschewing the need for stratification via step-indexing.

Virtual Classes and Impredicative Type Members. Type members in DOT and Scala eschew the sort of universe hierarchy described in the previous paragraph: we say they feature *impredicative* type members. Impredicative type members also feature in other type systems with path-dependent types or virtual classes [Clarke et al. 2007; Ernst et al. 2006].

9 FUTURE WORK

Annotation inference and type checking. DOT calculi are not meant to be programmed in directly; type checking of DOT is conjectured to be undecidable, like $D_{<:}$ [Hu and Lhoták 2020], and gDOT additionally requires inserting later ($\triangleright$) and coercion (**coerce**) annotations. Future work could investigate inference of these annotations, either directly [Severi 2019], or by translating from a Scala subset with decidable typechecking [Cremet et al. 2006] into gDOT or a suitable variant.

Expressivity. The programs we prove safe are decorated by no-op coercions (**coerce**). We conjecture that removing these coercions preserves safety, but we leave a proof for future work.

Amin et al. [2016] prove that all $F_{<:}$ programs can be translated into DOT, but their proof does not easily extend to gDOT. However, we have been able to translate many given examples by hand by adding a sufficient number of ▹ and **coerce** annotations, so we conjecture that there exists a whole-program encoding of $F_{<:}$ programs into gDOT.

Additional features. We are investigating support for higher-kinded types, by modeling type arguments as values. The latest work in this direction [Stucki 2016] ran into strong challenges and a counterexample to soundness (luckily, not affecting Scala). We conjecture that our techniques scale directly to this form of higher kinds, and that gDOT's existing guardedness restrictions already rule out this counterexample.

ACKNOWLEDGMENTS

We thank Sandro Stucki, Tiark Rompf, Nada Amin, Marianna Rapoport, Samuel Grütter, Ondřej Lhoták, Andreas Rossberg and Dimitrios Vytiniotis for helpful discussions on DOT, and Derek Dreyer for first directing the first author to Iris. Amin Timany was supported by the Flemish research fund (FWO) and Robbert Krebbers was supported by the Netherlands Organisation for Scientific Research (NWO), project number 016.Veni.192.259. This work was also supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation and by the ModuRes Sapere Aude Advanced Grant from The Danish Council for Independent Research for the Natural Sciences (FNU).

REFERENCES

Amal Ahmed. 2004. *Semantics of Types for Mutable State.* Ph.D. Dissertation. Princeton University.

Pierre America and Jan J. M. M. Rutten. 1989. Solving reflexive domain equations in a category of complete metric spaces. *JCSS* 39, 3 (1989), 343–375.

Nada Amin. 2016. *Dependent Object Types.* Ph.D. Dissertation. EPFL.

Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. In *WadlerFest (LNCS)*, Vol. 9600. 249–272.

Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *FOOL*.

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *TOPLAS* 23, 5 (2001), 657–683.

Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *POPL*. 109–122.

Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. 2011. Step-indexed Kripke models over recursive worlds. In *POPL*. 119–132.

Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. 2010. The category-theoretic solution of recursive metric-space equations. *TCS* 411, 47 (2010), 4102–4122.

Michael Brandt and Fritz Henglein. 1998. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticae* 33, 4 (1998), 309–338.

Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. 2007. Tribe: a simple virtual class calculus. In *AOSD*, Vol. 208. 121–134.

Karl Crary. 2017. Modules, abstraction, and parametric polymorphism. In *POPL*. 100–113.

Karl Crary, Robert Harper, and Sidd Puri. 1999. What is a recursive module?. In *PLDI*. 50–63.

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A core calculus for Scala type checking. In *MFCS (LNCS)*, Vol. 4162. 1–23.

Erik Ernst, Klaus Ostermann, and William R. Cook. 2006. A virtual class calculus. In *POPL*. 270–282.

Peng Fu and Aaron Stump. 2014. Self types for dependently typed lambda encodings. In *RTA-TLCA (LNCS)*, Vol. 8560. 224–239.

Paolo G. Giarrusso. 2019. Can we prove that type constructors are "distributive"? Github issue, https://web.archive.org/web/20200304175526/https://github.com/lampepfl/dotty-feature-requests/issues/51, archived on 04 March 2020.

Robert Harper and Mark Lillibridge. 1994. A type-theoretic approach to higher-order modules with sharing. In *POPL*. 123–137.

Robert Harper and John C. Mitchell. 1993. On the type structure of Standard ML. *TOPLAS* 15, 2 (1993), 211–252.

Jason Z. S. Hu and Ondřej Lhoták. 2020. Undecidability of $D_{<:}$ and its decidable fragments. *PACMPL* 4, POPL (2020), 9:1–9:30.

DeLesley S. Hutchins. 2010. Pure subtype systems. In *POPL*. 287–298.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018a. RustBelt: Securing the foundations of the Rust programming language. *PACMPL* 2, POPL (2018), 66:1–66:34.

Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. 2016. Higher-order ghost state. In *ICFP*. 256–269.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018b. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *JFP* 28 (2018), e20.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *POPL*. 637–650.

Ifaz Kabir and Ondřej Lhoták. 2018. κDOT: Scaling DOT with mutation and constructors. In *SCALA@ICFP*. 40–50.

Alexei Kopylov. 2003. Dependent intersection: A new way of defining records in type theory. In *LICS*. 86–95.

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: A general, extensible modal framework for interactive proofs in separation logic. *PACMPL* 2, ICFP (2018), 77:1–77:30.

Robbert Krebbers, Ralf Jung, Ales Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. 2017a. The essence of higher-order concurrent separation logic. In *ESOP (LNCS)*, Vol. 10201. 696–723.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017b. Interactive proofs in higher-order concurrent separation logic. In *POPL*. 205–217.

Robin Milner. 1978. A theory of type polymorphism in programming. *JCSS* 17, 3 (1978), 348–375.

Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS*. 255–266.

Abel Nieto. 2017. Towards algorithmic typing for DOT (short paper). In *SCALA@SPLASH*. 2–7.

Martin Odersky. 2016. DOT with higher-kinded types. Github discussion, https://web.archive.org/web/20200304175613/https://gist.github.com/odersky/36aee4b7fe6716d1016ed37051caae95, archived on 04 March 2020.

Martin Odersky, Guillaume Martres, and Dmitry Petrashko. 2016. Implementing higher-kinded types in Dotty. In *SCALA@SPLASH*. 51–60.

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *MSCS* 11, 4 (2001), 511–540.

Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A simple soundness proof for dependent object types. *PACMPL* 1, OOPSLA (2017), 46:1–46:27.

Marianna Rapoport and Ondřej Lhoták. 2016. *Mutable WadlerFest DOT*. Technical Report. University of Waterloo. http://arxiv.org/abs/1611.07610

Marianna Rapoport and Ondřej Lhoták. 2019. A path to DOT: formalizing fully path-dependent types. *PACMPL* 3, OOPSLA (2019), 145:1–145:29.

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

Steven Schäfer, Tobias Tebbi, and Gert Smolka. 2015. Autosubst: reasoning with de Bruijn terms and parallel substitutions. In *ITP (LNCS)*, Vol. 9236. 359–374.

Paula Severi. 2019. A light modality for recursion. *LMCS* 15, 1 (2019).

Kathrin Stark, Steven Schäfer, and Jonas Kaiser. 2019. Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions. In *CPP*. 166–180.

Sandro Stucki. 2016. DOT with higher-kinded types — A sketch. Github discussion, https://web.archive.org/web/20200304175148/https://gist.github.com/sstucki/3fa46d2c4ce6f54dc61c3d33fc898098, archived on 04 March 2020.

Fei Wang and Tiark Rompf. 2017. Towards strong normalization for dependent object types (DOT). In *ECOOP (LIPIcs)*, Vol. 74. 27:1–27:25.

Yanpeng Yang and Bruno C. d. S. Oliveira. 2017. Unifying typing and subtyping. *PACMPL* 1, OOPSLA (2017), 47:1–47:26.