The Essence of Dependent Object Types

Nada Amin*, Samuel Grütter*, Martin Odersky*, Tiark Rompf †, and Sandro Stucki*

*EPFL, Switzerland: {first.last}@epfl.ch
†Purdue University, USA: {first}@purdue.edu

Abstract. Focusing on path dependent types, the paper develops foundations for Scala from first principles. Starting from a simple calculus $D_{<:}$ of dependent functions, it adds records, intersections and recursion to arrive at DOT, a calculus for dependent object types. The paper shows an encoding of System F with subtyping in $D_{<:}$ and demonstrates the expressiveness of DOT by modeling a range of Scala constructs in it.

Keywords: Calculus, Dependent Types, Scala

1 Introduction

While hiking together in the French alps in 2013, Martin Odersky tried to explain to Phil Wadler why languages like Scala had foundations that were not directly related via the Curry-Howard isomorphism to logic. This did not go over well. As you would expect, Phil strongly disapproved. He argued that anything that was useful should have a grounding in logic. In this paper, we try to approach this goal.

We will develop a foundation for Scala from first principles. Scala is a functional language that expresses central aspects of modules as first-class terms and types. It identifies modules with *objects* and signatures with *traits*. For instance, here is a trait `KeyGen` that defines an abstract type `Key` and a way to retrieve a key from a string.

```
trait Keys {
  type Key
  def key(data: String): Key
}
```

A concrete implementation of `Keys` could be

```
object HashKeys extends Keys {
  type Key = Int
  def key(s: String) = s.hashCode
}
```

Here is a function which applies a given key generator to every element of a list of strings.

```
def mapKeys(k: Keys, ss: List[String]): List[k.Key] = ss.map(k.key)
```

The function returns a list of elements of type `k.Key`. This is a *path-dependent* type, which depends on the variable name `k`. In general a path in Scala consists of a variable name potentially followed by field selections, but in this paper we consider only simple paths consisting of a single variable reference. Path-dependent types give a limited form of type/term dependency, where types can depend on variables, but not on general terms. This form of dependency already gives considerable power, but at the same time is easier to handle than full term dependency because the equality relation on variables is just syntactic equality.

There are three essential elements of a system for path-dependent types. First, there needs to be a way to define a type as an element of a term, as in the definitions **`type`** `Key` and **`type`** `Key = Int`. We will consider initially only *type tags*, values that carry just one type and nothing else. Second, there needs to be a way to recover the tagged type from such a term, as in `p.Key`. Third, there needs to be a way to define and apply functions whose types depend on their parameters.

These three elements are formalized in System $D_{<:}$, a simple calculus for path-dependent types that has been discovered recently in an effort to reconstruct and mechanize foundational type system proposals for Scala bottom-up from simpler systems [**?**]. One core aspect of our approach to modeling path-dependent types is that instead of general substitutions we only have variable/-variable renamings. This ensures that paths always map to paths and keeps the treatment simple. On the other hand, with substitutions not available, we need another way to go from a type designator like `k.Key` to the underlying type it represents. The mechanism used here is subtyping. Types of type tags define lower and upper bound types. An abstract type such as **`type`** `Key` has the minimal type $\bot$ and the maximal type $\top$ as bounds. A type alias such as **`type`** `Key = T` is has `T` as both its lower and upper bound. A type designator is then a subtype of its upper bound and a supertype of its lower bound.

The resulting calculus is already quite expressive. It emerges as a generalization of System $F_{<:}$ [**?**], mapping type lambdas to term lambdas and type parameters to type tags. We proceed to develop System $D_{<:}$ into a richer calculus supporting objects as first class modules. This is done in three incremental steps. The first step adds records and intersections, the second adds type labels, and the third adds recursion. The final calculus, *DOT*, is a foundation for path-dependent object types. Many programming language concepts, including parameterized types and polymorphic functions, modules and functors, classes and algebraic data types can be modeled on this foundation via simple encodings.

A word on etymology: The term "System D" is associated in English and French with "thinking on your feet" or a "quick hack". In the French origin of the word, the letter 'D' stands for "se débrouiller", which means "to manage" or "to get things done". The meaning of the verb "débrouiller" alone is much nicer. It means: "Create order for things that are in confusion". What better motto for the foundations of a programming language?

Even if the name "System D" suggests quick thinking, in reality the development was anything but quick. Work on DOT started in 2007, following earlier

work on higher-level formalizations of Scala features [?,?]. Preliminary versions of a calculus with path-dependent types were published in FOOL 2012 [?] and OOPSLA 2014 [?]. Soundness results for versions of System $D_{<:}$ and DOT similar to the ones presented here, but based on big-step operational semantics, were recently established with mechanized proofs [?].

Our work shares many goals with recent work on ML modules. In this context, path-dependent types go back at least to SML [?], with foundational work on transparent bindings by Harper and Lillibridge [?] and Leroy [?]. MixML [?] drops the stratification requirement between module and term language and enables modules as first class values. More recent work models key aspects of modules as extensions of System F [?,?]. The latest development, 1ML [?], unifies the ML module and core languages through an elaboration to System F_ω. Our work differs from these in its integration of subtyping and an overriding focus on keeping the formalisms minimal.

The rest of this paper is structured as follows. Section 2 describes System $D_{<:}$. Section 3 shows how it can encode $F_{<:}$. Section 4 extends $D_{<:}$ to DOT. Section 5 studies expressiveness of DOT and shows how it relates to Scala.

2 System $D_{<:}$

Figure 1 summarizes our formulation of System $D_{<:}$. Its term language is essentially lambda calculus, with one additional form of value: A type tag $\{A = T\}$ is a value that associates a label A with a type T. For the moment we need only a single type label, so A can be regarded as ranging over an alphabet with just one name. This will be generalized later.

Our description differs from [?] in two aspects. First, terms are restricted to ANF form. That is, every intermediate value is abstracted out in a let binding. Second, evaluation is expressed by a small step reduction relation, as opposed to a big-step evaluator. Reduction uses only variable/variable renamings instead of full substitution. Instead of being copied by a substitution step, values stay in their let bindings. This is similar to the techniques used in the call-by-need lambda calculus [?].

The type assignment rules in Figure 1 define a straightforward dependent typing discipline. A lambda abstraction has a dependent function type $\forall(x:S)T$. This is like a dependent product $\Pi(x:S)T$ in LF [?], but with the restriction that the variable x can be instantiated only with other variables, not general terms. Type tags have types of the form $\{A : S..U\}$, they represent types labeled A which are lower-bounded by S and upper-bounded by T. A type tag referring to one specific type is expressed by having the lower and upper bound coincide, as in $\{A : T..T\}$. The type of a variable x referring to a type tag can be recovered with a type projection $x.A$.

The subtyping rules in Figure 1 define a preorder $T <: T$ between types with rules (Refl) and (Trans). They specify $\top$ and $\bot$ as greatest and least types (Top), (Bot), and make a type projection $x.A$ be a supertype of its lower bound (<:-Sel) and a subtype of its upper bound (Sel-<:). Furthermore, the

Syntax

x, y, z	**Variable**	$d ::=$	**Definition**
$v ::=$	**Value**	$\quad \{A = T\}$	type tag
$\quad d$	labelled def.		
$\quad \lambda(x{:}T)t$	lambda	$S, T, U ::=$	**Type**
$s, t, u ::=$	**Term**	$\quad \top$	top type
$\quad x$	variable	$\quad \bot$	bottom type
$\quad v$	value	$\quad \{A : S..T\}$	type declaration
$\quad x\ y$	application	$\quad x.A$	type projection
$\quad \mathbf{let}\ x = t\ \mathbf{in}\ u$	let	$\quad \forall(x{:}S)T$	dependent function

Evaluation $\boxed{t \longrightarrow t}$

$$\begin{array}{rll} \mathbf{let}\ x = v\ \mathbf{in}\ e[x\ y] & \longrightarrow \mathbf{let}\ x = v\ \mathbf{in}\ e[[z := y]t] & \mathbf{if}\ v = \lambda(z{:}T)t \\ \mathbf{let}\ x = y\ \mathbf{in}\ t & \longrightarrow [x := y]t & \\ \mathbf{let}\ x = \mathbf{let}\ y = s\ \mathbf{in}\ t\ \mathbf{in}\ u & \longrightarrow \mathbf{let}\ y = s\ \mathbf{in}\ \mathbf{let}\ x = t\ \mathbf{in}\ u & \\ e[t] & \longrightarrow e[u] & \mathbf{if}\ t \longrightarrow u \end{array}$$

$$\mathbf{where}\ e ::= [\,] \mid \mathbf{let}\ x = [\,]\ \mathbf{in}\ t \mid \mathbf{let}\ x = v\ \mathbf{in}\ e$$

Type Assignment $\boxed{\Gamma \vdash t : T}$

$$\Gamma,\ x : T,\ \Gamma' \vdash x : T \quad (\textsc{Var})$$

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

$$\frac{\Gamma,\ x : T \vdash t : U \quad x \notin fv(T)}{\Gamma \vdash \lambda(x{:}T)t : \forall(x{:}T)U} \quad (\textsc{All-I})$$

$$\frac{\Gamma \vdash x : \forall(z{:}S)T \quad \Gamma \vdash y : S}{\Gamma \vdash x\ y : [z := y]T} \quad (\textsc{All-E})$$

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

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

Subtyping $\boxed{\Gamma \vdash T <: T}$

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

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

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

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

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

Fig. 1. System $D_{<:}$

standard co/contravariant subtyping relationships are introduced between pairs of function types (All-<:-All) and tagged types (Typ-<:-Typ).

System $D_{<:}$ can encode System $F_{<:}$ as we will see in section 3. However, unlike System $F_{<:}$, System $D_{<:}$ does not have type variables. Instead, type definitions, such as $\{A = T\}$, are first-class values of type $\{A : T..T\}$. Combined with dependent functions, these path-dependent types can express the idioms of type variables, such as polymorphism.

For example, take the polymorphic identity function in System $F_{<:}$:

$$\vdash \Lambda(\alpha <: \top).\lambda(x : \alpha).x \quad : \quad \forall(\alpha <: \top).\alpha \to \alpha$$

and in System $D_{<:}$:

$$\vdash \lambda(a : \{A : \bot..\top\}).\lambda(x : a.A).x \quad : \quad \forall(a{:}\{A : \bot..\top\})\forall(x{:}a.A)a.A$$

Like in System $F_{<:}$, we can apply the polymorphic identity function to some type, say T, to get the identity function on T:

$$\vdash \textbf{let } f = \ldots \textbf{ in let } a = \{A = T\} \textbf{ in } f\ a \quad : \quad \forall(x{:}T)T$$

The role of subtyping is essential: (1) the argument a of type $\{A : T..T\}$ can be used for the parameter a of type $\{A : \bot..\top\}$, (2) the dependent result type $\forall(x{:}a.A)a.A$ can be converted to $\forall(x{:}T)T$ because $T <: a.A <: T$.

2.1 Example: Dependent Sums / Σs

Dependent sums can be encoded using dependent functions, in the usual way:

$$\begin{aligned}
\Sigma(x : S)T &\equiv \forall(z{:}\{A : \bot..\top\})\forall(f{:}\forall(x{:}S)\forall(y{:}T)z.A)z.A \\
\textbf{pack } [x, y] \textbf{ as } \Sigma(x : S)T &\equiv \lambda(z{:}\{A : \bot..\top\})\lambda(f{:}\forall(x{:}S)\forall(y{:}T)z.A)f\ x\ y \\
\textbf{unpack } x : S, y : T = t \textbf{ in } u &\equiv \textbf{let } z_1 = t \textbf{ in let } z_2 = \{A = U\} \textbf{ in} \\
&\qquad \textbf{let } z_3 = (\lambda(x{:}S)\lambda(y{:}T)u) \textbf{ in} \\
&\qquad \textbf{let } z_4 = z_1\ z_2 \textbf{ in } z_4\ z_3 \\
z.1 &\equiv \textbf{unpack } x : S, y : T = z \textbf{ in } x \\
z.2 &\equiv \textbf{unpack } x : S, y : T = z \textbf{ in } y
\end{aligned}$$

where U is the type of u. The associated, admissible subtyping and typing rules are easy to derive and can be found in Appendix A.1. Note that

1. unpacking via $\textbf{unpack } x, y = t \textbf{ in } u$ is only allowed if x does not appear free in the type U of u,
2. similarly, the second projection operator $-.2$ may only be used if x does not appear free in T. In such cases, we have $\Sigma(x : S)T = S \times T$, i.e. z is in fact an ordinary pair.

These restrictions are similar to the ones imposed on existential types in System F.

3 Correspondence with $F_{<:}$

System $D_{<:}$ initially emerged as a generalization of $F_{<:}$, mapping type lambdas to term lambdas and type parameters to type tags, and removing certain restrictions in a big-step evaluator for $F_{<:}$ [**?**]. We make this correspondence explicit below.

Pick an injective mapping from type variables X to term variables x_X. In the following, any variable names not written with an X subscript are assumed to be outside the range of that mapping. Let the encoding * from $F_{<:}$ types and terms to $D_{<:}$ types and terms be defined as follows.

$$\begin{aligned} X^* &= x_X.A \\ \top^* &= \top \\ (T \to U)^* &= \forall(x{:}T^*)U^* \\ (\forall(X <: S)T)^* &= \forall(x_X{:}\{A : \bot..S^*\})T^* \end{aligned}$$

$$\begin{aligned} x^* &= x \\ (\lambda(x : T)t)^* &= \lambda(x : T^*)t^* \\ (\Lambda(X <: S)t)^* &= \lambda(x_X : \{A : \bot..S^*\})t^* \\ (t\ u)^* &= \textbf{let}\ x = t^*\ \textbf{in let}\ y = u^*\ \textbf{in}\ x\ y \\ (t[U])^* &= \textbf{let}\ x = t^*\ \textbf{in let}\ y_Y = \{A = U^*\}\ \textbf{in}\ x\ y_Y \end{aligned}$$

Note that there are $D_{<:}$ terms that are not in the image of *. An example is $\lambda(x : \{A{:}\top..\top\})x$. Typing contexts are translated point-wise as follows:

$$\begin{aligned} (X <: T)^* &= x_X : \{A : \bot..T^*\} \\ (x : T)^* &= x : T^* \end{aligned}$$

Theorem 1. *If* $\Gamma \vdash_F S <: T$ *then* $\Gamma^* \vdash_D S^* <: T^*$.

Proof. The proof is by straight-forward induction on (System F) subtyping derivations. The only non-trivial case is (TVar), which follows from (Var) and (Sel-<:).

$$\frac{\Gamma^*,\ x_X : \{A : \bot..T^*\},\ \Gamma'^* \vdash x_X : \{A : \bot..T^*\}}{\Gamma^*,\ x_X : \{A : \bot..T^*\},\ \Gamma'^* \vdash x_X.A <: T^*} \qquad \text{(Sel-<:)}$$

Theorem 2. *If* $\Gamma \vdash_F t : T$ *then* $\Gamma^* \vdash_D t^* : T^*$.

Proof (sketch). The proof is by induction on (System F) typing derivations. The case for subsumption follows immediately from preservation of subtyping. The only remaining non-trivial cases are type and term application, which are given in detail in Appendix A.2.

Syntax ...

a, b, c	**Term member**	$d ::= \ldots$	**Definition**
A, B, C	**Type member**	$\{a = t\}$	field def.
$v ::=$	**Value**	$d \wedge d'$	aggregate def.
$\nu(x{:}T)d$	object	$S, T, U ::= \ldots$	**Type**
$\lambda(x{:}T)t$	lambda	$\{a : T\}$	field declaration
$s, t, u ::= \ldots$	**Term**	$S \wedge T$	intersection
$x.a$	selection	$\mu(x{:}T)$	recursive type

Evaluation ... $\boxed{t \longrightarrow t'}$

$$\textbf{let } x = v \textbf{ in } e[x.a] \longrightarrow \textbf{let } x = v \textbf{ in } e[t] \quad \textbf{if } v = \nu(x{:}T)\ldots\{a = t\}\ldots$$

Type Assignment ... $\boxed{\Gamma \vdash t : T}$ $\boxed{\Gamma \vdash d : T}$

$$\frac{\Gamma,\ x : T \vdash d : T}{\Gamma \vdash \nu(x{:}T)d : \mu(x{:}T)}\ (\{\}\text{-I}) \qquad \frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T}\ (\textsc{Fld}\text{-E})$$

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

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

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T\}}\ (\textsc{Fld}\text{-I}) \qquad \frac{\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \quad dom(d_1) \cap dom(d_2) = \emptyset}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2}\ (\textsc{AndDef}\text{-I})$$

Subtyping ... $\boxed{\Gamma \vdash T <: T}$

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

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

Fig. 2. DOT Extensions to $D_<$:

4 DOT

Figure 2 presents three extensions needed to turn $D_{<:}$ into a basis for a full programming language. The first extension adds records, the second adds type labels, and the third adds recursion. For reasons of space, all three extensions are combined in one figure.

4.1 Records

We model records by single field values that can be combined through intersections. A single-field record is of the form $\{a = t\}$ where a is a term label and t is a term. Records d_1, d_2 can be combined using the intersection $d_1 \wedge d_2$. The selection $x.a$ returns the term associated with label a in the record referred to by x. The evaluation rule for field selection is:

$$\textbf{let } x = v \textbf{ in } e[x.a] \longrightarrow \textbf{let } x = v \textbf{ in } e[t] \quad \textbf{if } v = \ldots \{a = t\} \ldots$$

It's worth noting that records are "call-by-name", that is, they associate labels with terms, not values. This choice was made because it sidesteps the issue how record fields should be initialized. The choice does not limit expressiveness, as a fully evaluated record can always be obtained by using let bindings to pre-evaluate field values before they are combined in a record.

New forms of types are single field types $\{a : T\}$ and intersection types $T \wedge U$. Subtyping rules for fields and intersection types are as expected. The typing rules (Fld-I) and (Fld-E) introduce and eliminate field types in the standard way. The typing rule (AndDef-I) types record combinations with intersection types. To ensure that combinations are consistent, it requires that the labels of the combined records are disjoint.

There is also the general introduction rule (And-I) well known from lambda calculus with intersections [**?**]. In the system with non-recursive records, that rule is redundant, because it can be obtained from subsumption and ($<:$-And). As demonstrated in Section 5, the rule does add expressiveness once recursion is added.

4.2 Type labels

Records with multiple type fields are supported simply by extending the alphabet of type labels from a single value to arbitrary names. In the following we let letters A, B, C range over type labels.

4.3 Recursion

The final extension introduces objects and recursive types. An object $\nu(x : T)d$ represents a record with definitions d that can refer to each other via the variable x. Its type is the recursive type $\mu(x : T)$. Following the path-dependent approach, a recursive type recurses over a term variable x, since type variables are absent

from DOT. Definitions d are now dropped as a separate kind of value, because they can always be expressed by wrapping them in a ν binder.

The evaluation rule for records is generalized to the one in Figure 2. There, when selecting $x.a$, the selected reference x and the self-reference in the object $\nu(x{:}T)d$ are required to coincide (this can always be achieved by α-renaming).

The introduction and elimination rules (Rec-I) and (Rec-E) for recursive types are as simple as they can possibly be. There is no subtyping rule between recursive types [**?**]. Instead, recursive types of variables are unwrapped with (Rec-E) and re-introduced with (Rec-I).

The choice of leaving out subtyping between recursive types represents a small loss in expressiveness but a significant gain in simplifying the metatheory.

4.4 Meta-Theory

Soundness results for versions of $D_{<:}$, DOT, and a number of variations were recently established with mechanized proofs by Rompf and Amin. These soundness results are phrased with respect to big-step evaluators. The type systems are slightly more expressive than the ones presented here in that they are not restricted to terms in ANF, and in including a subtyping rule on recursive types. This particular rule poses lots of challenges, which are described in detail in the technical report [**?**].

The meta theory for the version shown in this paper is so far only developed on paper. The central results are the usual small-step preservation and progress theorems, adapted to the ANF setting.

Definition 1. *An* answer n *is defined by the production*

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

Theorem 3. *(Progress) If* $\vdash t : T$ *then* t *is an answer or there is a term* u *such that* $t \longrightarrow u$.

Theorem 4. *(Preservation) If* $\Gamma \vdash t : T$ *and* $t \longrightarrow u$ *then* $\Gamma \vdash u : T$.

The central difficulty for coming up with proofs of these theorems was outlined in a previous FOOL workshop paper [**?**]: It is possible to arrive at type definitions with unsatisfiable bounds. For instance a type label A might have lower bound $\top$ and upper bound $\bot$. By subtyping rules (Sel-<:), (<:-Sel) and transitivity of subtyping one obtains $\top <: \bot$, which makes every type a subtype of every other type. So a single "bad" definition causes the whole subtyping relation to collapse to one point, just as an inconsistent theory can prove every proposition. With full type intersection and full recursion, it is impractical to rule out such bad bounds *a priori*. Instead, a soundness proof has to make use of the fact that environments corresponding to an actual execution trace correspond to types of concrete values. In a concrete object value any type definition is of the form $A = T$, so lower and upper bounds are the same, and bad bounds are excluded. Similar reasoning applied in the big-step proofs [**?**],

where the semantics has a natural distinction between static terms and run-time values. Here, we need to establish this distinction first.

We first define a *precise typing relation* $\Gamma \vdash_! t : T$ as follows:

Definition 2. *$\Gamma \vdash_! t : T$ if $\Gamma \vdash t$ and the following two conditions hold.*

1. *If t is a value, the typing derivation of t ends in* (All-I) *or* (-I).
2. *If t is a variable, the typing derivation of t consists only of* (Var), (Rec-E) *and* (Sub) *steps and every* (Sub) *step makes use of only the subtyping rules* (And-<:) *and* (Trans).

Definition 3. *A* store *s is a sequence of bindings $x = v$, with ϵ representing the empty store.*

Definition 4. *The combination $s \mid t$ combines a store s and a term t. It is defined as follows:*

$$\begin{array}{rcl} x = v, s \mid t & \equiv & \textbf{let } x = v \textbf{ in } (s \mid t) \\ \epsilon \mid t & \equiv & t \end{array}$$

Definition 5. *An environment $G = \overline{x_i : T_i}$ corresponds to a store $s = \overline{x_i = v_i}$, written $\Gamma \sim s$, if $\Gamma \vdash_! v_i : T_i$*

A precise typing relation over an environment that corresponds to a store can only derive type declarations where lower and upper bounds coincide. We make use of this in the following definition.

Definition 6. *A typing or subtyping derivation is* tight *in environment Γ if it only contains the following tight variants of (Sel-<:), (<:-Sel) for variables x bound in Γ:*

$$\frac{\Gamma \vdash_! x : \{A{:}T..T\}}{\Gamma \vdash T <: x.A}\ (<:\text{-Sel-tight}) \qquad \frac{\Gamma \vdash_! x : \{A{:}T..T\}}{\Gamma \vdash x.A <: T}\ (\text{Sel-}<:\text{-tight})$$

We write $\Gamma \vdash_\# t : T$ if $\Gamma \vdash t : T$ with a derivation that is tight in Γ.

A core part of the proof shows that the full (Sel-<:) and (<:-Sel) rules are admissible wrt $\vdash_\#$ if the underlying environment corresponds to a store.

Lemma 1. *If $\Gamma \sim s$ and $s(x) = \nu(x : T)d$ and $\Gamma \vdash_\# x : \{A : S..U\}$ then $\Gamma \vdash_\# x.A <: U$ and $\Gamma \vdash_\# S <: x.A$.*

The next step of the proof is to characterize the possible types of a variable bound in a store s in an environment that corresponds to s.

Definition 7. *The* possible types $\mathbf{Ts}(G, x, v)$ *of a variable x bound in an environment Γ and corresponding to a value v is the smallest set $\mathcal{S}$ such that:*

1. *If $v = \nu(x{:}T)d$ then $T \in \mathcal{S}$.*
2. *If $v = \nu(x{:}T)d$ and $\{a = t\} \in d$ and $\Gamma \vdash t : T'$ then $\{a{:}T'\} \in \mathcal{S}$.*

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

One can show that the possible types of a variable are closed under subtyping:

Lemma 2. *If* $\Gamma \sim s$ *and* $\Gamma \vdash x : T$ *and* $s(x) = v$ *then* $\mathbf{Ts}(G, x, v)$ *is closed wrt* $\Gamma \vdash \_ <: \_$.

The proof establishes by induction on the formation rules (1) – (7) that $\mathbf{Ts}(G, x, v)$ is closed wrt $\Gamma \vdash_\# \_ <: \_$. The property extends to the standard subtyping $\Gamma \vdash \_ <: \_$ with Lemma 1 since (Sel-<:) and (<:-Sel) are admissible for $\vdash_\#$ over Γ.

By an induction over typing derivations one can now relate $\mathbf{Ts}(G, x, v)$ and typings of x:

Lemma 3. *If* $\Gamma \sim s$ *and* $\Gamma \vdash x : T$ *then* $T \in \mathbf{Ts}(G, x, s(x))$.

With the possible types lemma it is easy to establish a standard canonical forms lemma.

Lemma 4. *(Canonical Forms) If* $\Gamma \sim s$, *then*

1. *If* $\Gamma \vdash x : \forall(x : T)U$ *then* $s(x) = \lambda(x : T')t$ *where* $\Gamma \vdash T <: T'$ *and* $\Gamma, x : T' \vdash t : U$.
2. *If* $\Gamma \vdash x : \{a:T\}$ *then* $s(x) = \nu(x : S)d$ *for some* S, d, t *such that* $\Gamma \vdash d : S$, $\{a = t\} \in d$, $\Gamma \vdash t : T$.

As usual, we also need a substitution lemma, the proof of which is by a standard mutual induction on typing and subtyping derivations.

Lemma 5. (Substitution) *If* $\Gamma, x : S \vdash t : T$ *and* $\Gamma \vdash y : [x := y]S$ *then* $G \vdash [x := y]t : [x := y]T$.

With (Canonical Forms) and (Substitution) one can then establish the following combination of progress and preservation theorems by an induction on typing derivations.

Proposition 1. *Assume* $\Gamma \vdash t : T$ *and* $\Gamma \sim s$. *Then either*

- t *is an answer, or*
- *There exist a store* s' *and a term* t' *such that* $s \mid t \longrightarrow s' \mid t'$ *and for any such* s', t' *there exists an environment* Γ' *such that* $\Gamma, \Gamma' \vdash t' : T$ *and* $\Gamma, \Gamma' \sim s'$.

Theorem 3 and Theorem 4 follow from Proposition 1.

5 Correspondence with Scala

In this section, we explore the expressiveness of DOT by studying program fragments that are typable in it. We start with an encoding of booleans, which is in spirit similar to the standard Church encoding, except that it also creates `Boolean` as a new nominal type. We use the abbreviation

$$IFT \equiv \texttt{\{ if: } \forall\texttt{(x: \{A: } \bot..\top\texttt{\})}\forall\texttt{(t: x.A)}\forall\texttt{(f: x.A): x.A \}}$$

A first step defines the type `boolImpl.Boolean` as an alias of its implementation, a record with a single member **`if`**.

```
let boolImpl =
  ν (b: { Boolean: IFT..IFT } ∧ { true: IFT } ∧ { false: IFT })
    { Boolean = IFT } ∧
    { true = λ(x: {A: ⊥..⊤})λ(t: x.A)λ(f: x.A)t } ∧
    { false = λ(x: {A: ⊥..⊤})λ(t: x.A)λ(f: x.A)f }
in ...
```

In a second step, the implementation of Boolean gets wrapped in a function that produces an abstract type. This establishes `bool.Boolean` as a nominal type, which is different from its implementation. The type is upper bounded by IFT, which means that the conditional **`if`** is still accessible as a member.

```
let bool =
  let boolWrapper =
    λ(x: μ(b: {Boolean: ⊥..IFT} ∧ {true: b.Boolean} ∧ { false: b.Boolean
        })) x
  in boolWrapper boolImpl
in ...
```

The previous example showed that that direct mappings into DOT can be tedious at times. The following shorthands help.

5.1 Abbreviations

– We group multiple intersected definitions or declarations in one pair of braces, replacing ∧ with `;` or a newline. E.g.

```
{ A = T; a = t } ≡ { A = T } ∧ { a = t }
{ A: S..T; a: T } ≡ { A: S..T } ∧ { a: T }
```

– We allow terms in applications and selections, using the expansions

```
t u ≡ let x = t in x u
x u ≡ let y = u in x y
t.a ≡ let x = t in x.a
```

– We expand type ascriptions to applications:

```
t: T ≡ (λ(x: T)x)t
```

– We abbreviate ν(x: T)d to ν(x)d if the type of definitions d is given explicitly.
– We abbreviate type bounds by expanding A <: T to A:⊥..T, A >:S to A: S..⊤, A = T to A: T..T, and A to A:⊥..⊤.

5.2 Example: A Covariant Lists Package

As a more involved example we show how can define a parameterized abstract data type as a class hierarchy. A simplified version of the standard covariant List type can be defined in Scala as follows:

```
package scala.collection.immutable
trait List[+A] {
 def isEmpty: Boolean; def head: A; def tail: List[A]
}
object List {
 def nil: List[Nothing] = new List[Nothing] {
  def isEmpty = true; def head = head; def tail = tail /* infinite loops */
 }
 def cons[A](hd: A, tl: List[A]) = new List[A] {
  def isEmpty = false; def head = hd; def tail = tl
 }
}
```

This defines List as a covariant type with head and tail members. The nil object defines an empty list of element type Nothing, which is Scala's bottom type. Its members are both implemented as infinite loops (lacking exceptions that's the only way to produce a bottom type). The cons function produces a non-empty list.

In the DOT encoding of this example, the element type A becomes an abstract type member of List, which is now itself a type member of the object denoting the package. From the outside, List is only upper-bounded, so that it becomes nominal: the only way to construct a List is through the package field members nil and cons. Also, note that the covariance of the element type A is reflected because the tail of the List only requires the upper bound of the element type to hold.

```
let scala_collection_immutable = ν(sci) {
 List = μ(thisList: {A; head: thisList.A; tail: sci.List∧{A <: thisList.A}})
 nil: sci.List∧{A = ⊥} =
  let thisList = ν(thisList) {
   A = ⊥; isEmpty = bool.true; head = thisList.head; tail = thisList.tail }
  in thisList
 cons: ∀(x: {A})∀(hd: x.A)∀(tl: sci.List∧{A <: x.A})sci.List∧{A <: x.A} =
  λ(x: {A})λ(hd: x.A)λ(tl: sci.List∧{A <: x.A})
   let thisList = ν(thisList) {
    A = x.A; isEmpty = bool.false; head = hd; tail = tl }
   in thisList
}: { μ(sci: {
```

```
  List <: μ(thisList: {A; head: thisList.A; tail: sci.List∧{A <:
       thisList.A}})
  nil: sci.List∧{A = ⊥}
  cons: ∀(x: {A})∀(hd: x.A)∀(tl: sci.List∧{A <: x.A})sci.List∧{A <: x.A}
})}
in ...
```

As an example of a typing derivation in this code fragment consider the right hand side of the `cons` method. To show that `thisList` corresponds to the given result type `sci.List∧{A <: thisList.A}`, the typechecker proceeeds as follows. First step:

by typing of rhs
`thisList: ρ(thisList: { A = x.A, hd: x.A, tl: sci.List∧{A = x.A}})`
by (Rec-E)
`thisList: { A = x.A, hd: x.A, tl: sci.List∧{A = x.A}}`
by (Sub), *since* thisList.A <: x.A
`thisList: { A = x.A, hd: thisList.A, tl: sci.List∧{A = thisList.A}}`
by (Sub), *since* A: x.A..x.A <: A
`thisList: { A, hd: thisList.A, tl: sci.List∧{ A = thisList.A}}`
by (Rec-I)
`thisList: ρ(thisList: { A, hd: thisList.A, tl: sci.List∧{ A = thisList.A}})`
by (Sub) *via* (<:-Sel) *on* sci.List
`thisList: sci.List`

Second step:

```
thisList: { A = x.A, hd: x.A, tl: sci.List∧{A = x.A}}
     by (Sub)
thisList: {A <: x.A}
```

Combining both steps with (And-I) gives

```
thisList: sci.List∧{A <: x.A}
```

DOT Rules

A Additional Rules and Detailed Proofs

A.1 Typing of Dependent Sums / Σs

There are one subtyping rule and four typing rules (all admissible in $D_{<:}$) associated with the encoding of dependent sums given in Section 2.1.

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

$$\frac{\Gamma \vdash x : S \qquad \Gamma \vdash y : T \qquad x \notin fv(S)}{\Gamma \vdash \mathbf{pack}\ [x, y]\ \mathbf{as}\ \Sigma(x : S)T : \Sigma(x : S)T} \qquad (\Sigma\text{-I})$$

Syntax

x, y, z	**Variable**	$v ::=$	**Value**
a, b, c	**Term member**	$\nu(x{:}T)d$	object
A, B, C	**Type member**	$\lambda(x{:}T)t$	lambda
$S, T, U ::=$	**Type**	$s, t, u ::=$	**Term**
$\top$	top type	x	variable
$\bot$	bot type	v	value
$\{a : T\}$	field declaration	$x.a$	selection
$\{A : S..T\}$	type declaration	$x\ y$	application
$x.A$	type projection	**let** $x = t$ **in** u	let
$S \wedge T$	intersection	$d ::=$	**Definition**
$\mu(x{:}T)$	recursive type	$\{a = t\}$	field def.
$\forall(x{:}S)T$	dependent function	$\{A = T\}$	type def.
		$d \wedge d'$	aggregate def.

Fig. 3. DOT: Syntax

Evaluation $\boxed{t \longrightarrow t'}$

$$
\begin{array}{rll}
e[t] & \longrightarrow e[t'] & \textbf{if } t \longrightarrow t' \\
\textbf{let } x = v \textbf{ in } e[x\ y] & \longrightarrow \textbf{let } x = v \textbf{ in } e[[z := y]t] & \textbf{if } v = \lambda(z{:}T)t \\
\textbf{let } x = v \textbf{ in } e[x.a] & \longrightarrow \textbf{let } x = v \textbf{ in } e[t] & \textbf{if } v = \nu(x{:}T) \ldots \{a = t\} \ldots \\
\textbf{let } x = y \textbf{ in } t & \longrightarrow [x := y]t & \\
\textbf{let } x = \textbf{let } y = s \textbf{ in } t \textbf{ in } u & \longrightarrow \textbf{let } y = s \textbf{ in let } x = t \textbf{ in } u & \\
 & \textbf{where } e ::= [\,] \mid \textbf{let } x = [\,] \textbf{ in } t \mid \textbf{let } x = v \textbf{ in } e &
\end{array}
$$

Fig. 4. DOT: Evaluation

Type Assignment $\boxed{\Gamma \vdash t : T}$

$$\Gamma,\ x : T,\ \Gamma' \vdash x : T \quad \text{(Var)}$$

$$\frac{\Gamma,\ x : T \vdash t : U}{\Gamma \vdash \lambda(x{:}T)t : \forall(x{:}T)U} \quad \text{(All-I)}$$

$$\frac{\Gamma \vdash x : \forall(z{:}S)T \quad \Gamma \vdash y : S}{\Gamma \vdash x\ y : [z := y]T} \quad \text{(All-E)}$$

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

$$\frac{\Gamma \vdash x : \{a : T\}}{\Gamma \vdash x.a : T} \quad (\{\}\text{-E})$$

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

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

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

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

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

Definition Type Assignment $\boxed{\Gamma \vdash d : T}$

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash \{a = t\} : \{a : T\}} \quad \text{(Fld-I)}$$

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

$$\frac{\Gamma \vdash d_1 : T_1 \quad \Gamma \vdash d_2 : T_2 \quad dom(d_1), dom(d_2)\ \text{disjoint}}{\Gamma \vdash d_1 \wedge d_2 : T_1 \wedge T_2} \quad \text{(AndDef-I)}$$

Fig. 5. DOT: Type Assignment

Subtyping $\boxed{\Gamma \vdash T <: T}$

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

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

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

$$\Gamma \vdash T \wedge U <: T \quad (\textsc{And}_2\text{-}<:) \qquad \frac{\Gamma \vdash x : \{A : S..T\}}{\Gamma \vdash S <: x.A}\ (<:\text{-}\textsc{Sel})$$

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

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

Fig. 6. DOT: Subtyping

$$\frac{\Gamma \vdash t : \Sigma(x : S)T \qquad \Gamma,\ x : S,\ y : T \vdash u : U \qquad x, y \notin fv(U)}{\Gamma \vdash \textbf{unpack}\ x : S, y : T = t\ \textbf{in}\ u : U}\ (\Sigma\text{-E})$$

$$\frac{\Gamma \vdash z : \Sigma(x : S)T}{\Gamma \vdash z.1 : S}\ (\Sigma\text{-}\textsc{Proj}_1)$$

$$\frac{\Gamma \vdash z : \Sigma(x : S)T \qquad x \notin fv(T)}{\Gamma \vdash z.2 : T}\ (\Sigma\text{-}\textsc{Proj}_2)$$

The proofs of admissibility are left as an (easy) exercise to the reader.

A.2 Proof of Theorem 2

We want to show that the translation $-^*$ preserves typing, i.e. if $\Gamma \vdash_F t : T$ then $\Gamma^* \vdash_D t^* : T^*$.

We start by stating and proving two helper lemmas. Write $[x.A := U]T$ for the capture-avoiding substitution of $x.A$ for U in T.

Lemma 6. *Substitution of type variables for types commutes with translation, i.e.*

$$([X := U]T)^* = [x_X.A := U^*]T^*$$

Proof. The proof is by straight-forward induction on the structure of T.

Lemma 7. *The following subtyping rules are admissible in $D_<$:*

$$\frac{\Gamma \vdash x : \{A : U..U\}}{\Gamma \vdash [x.A := U]T <: T} \qquad \frac{\Gamma \vdash x : \{A : U..U\}}{\Gamma \vdash T <: [x.A := U]T}$$

Proof. The proof is by induction on the structure of T.

Proof (Theorem 2). Proof is by induction on (System F) typing derivations. The case for subsumption follows immediately from preservation of subtyping. The only remaining non-trivial cases are type and term application.

Case (App). We need to show that

$$\frac{\Gamma^* \vdash s^* : \forall(z{:}S^*)T^* \qquad \Gamma^* \vdash t^* : S^*}{\Gamma^* \vdash \textbf{let } x = s^* \textbf{ in let } y = t^* \textbf{ in } x\ y : T^*}$$

is admissible. First note that z is not in $fv(T^*)$, hence $[z := y]T^* = T^*$ for all y. In particular, using (Var) and (All-E), we have

$$\frac{\Gamma' \vdash x : \forall(z{:}S^*)T^* \qquad \Gamma' \vdash y : S^*}{\Gamma' \vdash x\ y : T^*}\ (\text{All-E})$$

where $\Gamma' = \Gamma^*, x : \forall(z : S^*)T^*, y : S^*$. Applying the induction hypothesis, context weakening and (Let), we obtain

$$\frac{\Gamma^*, x : \forall(z{:}S^*)T^* \vdash t^* : S^* \qquad \Gamma' \vdash x\ z : T^*}{\Gamma^*, x : \forall(z{:}S^*)T^* \vdash \textbf{let } y = t^* \textbf{ in } x\ y : T^*}\ (\text{Let})$$

The result follows by applying the induction hypothesis once more:

$$\frac{\Gamma^* \vdash s^* : \forall(z{:}S)^*T^* \qquad \Gamma^*, x : \forall(z{:}S)^*T^* \vdash \textbf{let } y = t^* \textbf{ in } x\ y : T^*}{\Gamma^* \vdash \textbf{let } x = s^* \textbf{ in let } y = t^* \textbf{ in } x\ y : T^*}\ (\text{Let})$$

Case (TApp). By preservation of subtyping, we have $\Gamma^* \vdash U^* <: S^*$, hence it suffices to show that

$$\frac{\Gamma^* \vdash t^* : \forall(x_X{:}\{A : \bot..S^*\})T^* \qquad \Gamma^* \vdash U^* <: S^*}{\Gamma^* \vdash \textbf{let } x = t^* \textbf{ in let } y_Y = \{A = U^*\} \textbf{ in } x\ y_Y : ([X := U]T)^*}$$

is admissible. By context weakening, (Typ-<:-Typ), (Bot) and (Sub), we have

$$\frac{\Gamma' \vdash y_Y : \{A : U^*..U^*\} \qquad \dfrac{\Gamma' \vdash \bot <: U^* \qquad \Gamma' \vdash U^* <: S^*}{\Gamma' \vdash y_Y : \{A : \bot..S^*\}}\ (\text{Typ-<:-Typ})}{\Gamma' \vdash y_Y : \{A : \bot..S^*\}}\ (\text{Sub})$$

where $\Gamma' = \Gamma^*, x : \forall(x_X : \{A : \bot..S^*\})T^*, y_Y : \{A : U^*..U^*\}$. By (Var) and (All-E), we have

$$\frac{\Gamma' \vdash x : \forall(x_X : \{A : \bot..S^*\})T^* \qquad \Gamma' \vdash y_Y : \{A : \bot..S^*\}}{\Gamma' \vdash x\ y_Y : [x_X := y_Y]T^*}\ (\textsc{All-E})$$

Next, we observe that, by the first Lemma 6, type substitution commutes with translation, i.e.

$$\begin{aligned}([X := U]T)^* &= [x_X.A := U^*]T^* \\ &= [y_Y.A := U^*][x_X := y_Y]T^*\end{aligned}$$

By the Lemma 7, (Var) and (Sub) we thus have

$$\frac{\Gamma' \vdash x y_Y : [x :=_{]} X y_Y T^* \qquad \dfrac{\Gamma' \vdash y_Y : \{A : U^*..U^*\}}{\Gamma' \vdash [x_X := y_Y]T^* <: ([X := U]T)^*}\ (\text{Lemma 7})}{\Gamma' \vdash x\ y_Y : ([X := U]T)^*}\ (\textsc{Sub})$$

The result follows from applying (Let) twice. Note that, being fresh, neither x nor y_Y appear free in $([X := U]T)^*$, hence the hygiene condition of (Let) holds.