# 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 `Keys` 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
}
```

* This is a revised version of a paper presented at WF '16. The original publication is available from Springer via http://dx.doi.org/10.1007/978-3-319-30936-1_14.

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 (Rompf and Amin, 2015). 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` 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 $F_{<:}$ (Cardelli et al., 1994), 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 (Odersky et al., 2003; Cremet et al., 2006). Preliminary versions of a calculus with path-dependent types were published in FOOL 2012 (Amin et al., 2012) and OOPSLA 2014 (Amin et al., 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 (Rompf and Amin, 2015).

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. Sections 5 and 6 study the expressiveness of DOT and show how it relates to Scala. Section 7 outlines the implementation of the DOT constructs in a full Scala compiler. Section 8 discusses related work and Section 9 concludes.

## 2 System $D_{<:}$

Figure 1 summarizes our formulation of System $D_{<:}$. Its term language is essentially the 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 Rompf and Amin (2015) 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 (Ariola et al., 1995).

We use Barendregt's Variable Convention throughout. For example, in the third evaluation rule, which un-nests let-bindings, we assume that we can appropriately $\alpha$-rename the variable $y$ which changes scope so that it is not captured in the final term $u$.

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 (Harper et al., 1993), 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 $U$. 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 $x.A$.

**Syntax**

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

**Evaluation** $\boxed{t \longrightarrow t}$

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

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

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

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

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

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

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

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

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

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

$$\Gamma \vdash T <: \top \quad \text{(Top)}$$

$$\Gamma \vdash \bot <: T \quad \text{(Bot)}$$

$$\Gamma \vdash T <: T \quad \text{(Refl)}$$

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

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

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

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

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

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

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

Dependent sums can be encoded using dependent functions, through an encoding similar to that of existential types in System F.

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

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 **unpack** $x, y = t$ **in** $u$ is only allowed if $x$ and $y$ do 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 may come as a surprise: while they are similar to the hygiene conditions imposed on existential types in System $\mathrm{F}/\mathrm{F}_{<:}$, they do not apply to dependent sums in (fully) dependently typed languages. In such languages, the bound names $x$, $y$ can be prevented from leaking into the overall type of an **unpack** statement by substituting the projections $t.1$ and $t.2$ for occurrences of $x$ and $y$ in $U$. The same is true for the return type of the second projection $z.2$, which would be $[x := z.1]T$. Unfortunately, such substitutions are forbidden in $\mathrm{D}_{<:}$ because types may only depend on variables, as opposed to arbitrary terms (like $z.1$ or $t.2$). For the same reason, the typing rule (Let) for **let** expressions features a similar hygiene condition. Finally, note that the above encoding allows for the unrestricted projection of "existential witnesses" via $-.1$, whereas no such operation exists on existential types in System $\mathrm{F}/\mathrm{F}_{<:}$.

## 3 Embedding $\mathrm{F}_{<:}$ in $\mathrm{D}_{<:}$

System $\mathrm{D}_{<:}$ initially emerged as a generalization of $\mathrm{F}_{<:}$, mapping type lambdas to term lambdas and type parameters to type tags, and removing certain restrictions in a big-step evaluator for $\mathrm{F}_{<:}$ (Rompf and Amin, 2015). 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 translation $^*$ from $\mathrm{F}_{<:}$ types and terms to $\mathrm{D}_{<:}$ types and terms be defined as follows.[1]

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

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

[1] The definition of $t^*$ assumes a countable supply of fresh names $x$, $X \notin \mathit{fv}(t)$.

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:

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

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

*Proof.* The proof is by straight-forward induction on $F_{<:}$ subtyping derivations. The only non-trivial case is subtyping of type variables, which follows from (Var) and (Sel-<:).

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

**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 interesting cases are type and term application, which are given in detail in Appendix A.2.

## 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

**Syntax** ...

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

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

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

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

**Type Assignment (definitions)** $\boxed{\Gamma \vdash d : T}$

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

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

**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 $D_{<:}$

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 (Coppo et al., 1979). 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. Note that type tags are no longer terms in their own right: they must now be wrapped in a $\nu$ binder like any other record definition.

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 $\alpha$-renaming).

The rule ({}-I) assigns a recursive type to an object by typing its definition. Note that the definitions are typed without subsumption (which is only defined on terms, and in this final extension, definitions $d$ are syntactically not terms $t$).

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 (Amadio and Cardelli, 1993). 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 meta-theory.

### 4.4 Meta-Theory

This section presents the main type soundness theorems of DOT and outlines their proofs, which have been machine-checked.[2] The central results are the usual small-step preservation and progress theorems.

Soundness results for slightly different versions of $D_{<:}$, DOT, and a number of variations were established previously with mechanized proofs by Rompf and

[2] The full development on paper and in Coq is at `http://wadlerfest.namin.net`.

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 (Rompf and Amin, 2015).

**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* $\vdash t : T$ *and* $t \longrightarrow u$ *then* $\vdash u : T$.

Note that the preservation property is weaker than normally stated in that it demands an empty context. We will strengthen the theorem in the proofs to get a useful induction property.

The central difficulty for coming up with proofs of these theorems was outlined in a previous FOOL workshop paper (Amin et al., 2012): 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 (Rompf and Amin, 2015), 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* $\Gamma \vdash t : T$ *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* ($\text{And}_1$-<:), ($\text{And}_2$-<:) *and* (Trans).

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

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

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

**Definition 5.** *An environment* $\Gamma = \overline{x_i : T_i}$ *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* $\Gamma$ *if it only contains the following tight variants of* (Sel-<:), (<:-Sel) *when* $\Gamma' = \Gamma$*:*

$$\frac{\Gamma' \vdash_! x : \{A{:}T..T\}}{\Gamma' \vdash T <: x.A}\ (<:\text{-Sel-tight}) \qquad \frac{\Gamma' \vdash_! x : \{A{:}T..T\}}{\Gamma' \vdash x.A <: T}\ (\text{Sel-}<:\text{-tight})$$

*For environments that extend* $\Gamma$, *full* (Sel-<:) *and* (<:-Sel) *are permitted. We write* $\Gamma \vdash_\# t : T$ *or* $\Gamma \vdash_\# S <: U$ *if* $\Gamma \vdash t : T$ *or* $\Gamma \vdash S <: U$ *with a derivation that is tight in* $\Gamma$.

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 $\mathbf{Ts}(\Gamma, x, v)$ *of a variable* $x$ *bound in an environment* $\Gamma$ *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* $\mu(x{:}T) \in \mathcal{S}$.
8. $\top \in \mathcal{S}$.

The possible types of a variable are closed under subtyping:

**Lemma 2.** *If* $\Gamma \sim s$, $s(x) = v$, $T \in \mathbf{Ts}(\Gamma, x, v)$ *and* $\Gamma \vdash T <: U$, *then* $U \in \mathbf{Ts}(\Gamma, x, v)$.

The possible types of a variable include each of its typings:

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

These possible types lemmas are proved first for tight subtyping and typing, by induction on the subtyping or typing derivation, and extended to standard subtyping and typing with Lemma 1, since (Sel-<:) and (<:-Sel) are admissible for $\vdash_{\#}$ over $\Gamma$.

With the above lemmas 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 $s(x) = \lambda(x:T')t$ for some $T'$ and $t$ such that $\Gamma \vdash T <: T'$ 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 $\Gamma \vdash [x := y]t : [x := y]T$.*

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 $\Gamma'$ such that $\Gamma, \Gamma' \vdash t' : T$ and $\Gamma, \Gamma' \sim s'$.*

Theorem 3 and Theorem 4 follow from Proposition 1, since the empty environment corresponds to the empty store.

## 5 Encodings

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

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

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 = { if = λ(x: {A: ⊥..⊤})λ(t: x.A)λ(f: x.A)t } } ∧
   { false = { if = 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 ...
```

This example shows 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 to 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(self: {A; isEmpty: bool.Boolean; head: self.A; tail: sci.List∧{A
     <: self.A}})
 nil: sci.List∧{A = ⊥} =
  let result = ν(self) {
   A = ⊥; isEmpty = bool.true; head = self.head; tail = self.tail }
  in result
 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 result = ν(self) {
    A = x.A; isEmpty = bool.false; head = hd; tail = tl }
   in result
}: { μ(sci: {
 List <: μ(self: {A; head: self.A; tail: sci.List∧{A <: self.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 `result` corresponds to the given result type `sci.List∧{A <: x.A}`, the typechecker proceeeds as follows. First step:

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

Second step:

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

Combining both steps with (And-I) gives

`result: sci.List∧{A <: x.A}`

The explicit naming of the `result` was necessary so that we could "unwrap" the recursive type on it, apply subsumption and then "rewrap" using (Rec-I) and (And-I). In Scala, this sequence of steps can be an automatic conversion on the value; no separate let-binding is needed.

How close is this encoding to actual Scala? Here's legal Scala code which closely mimicks the previous list encoding in DOT.

```
object scala_collection_immutable { sci =>
  trait List { self =>
    type A
    def isEmpty: Boolean
    def head: self.A
    def tail: List{type A <: self.A}
  }

  def nil: sci.List{type A = Nothing} = new List{ self =>
    type A = Nothing
    def isEmpty = true
    def head: A = self.head
    def tail: List{type A = Nothing} = self.tail
  }

  def cons(x: {type A})(hd: x.A)(tl: sci.List{type A <: x.A})
    : sci.List{type A <: x.A} = new List{ self =>
    type A = x.A
    def isEmpty = false
    def head = hd
```

```
    def tail = tl
  }
}
```

The main difference between the DOT and Scala versions concerns recursive types. Scala does not support recusive types directly, but recursion is employed indirectly in the definition of traits and objects. Consequently, `List` in now a trait instead of a type definition, and `scala_collection_immutable` is an object instead of a let-bound variable. Otherwise, there are only minor syntactic differences.

## 6 Dependent Types in Scala

The previous section showed how generic types can be encoded as path-dependent types in DOT. So it established that DOT is sufficiently expressive to encode standard abstraction patterns we expect in programming languages today. In this section we show by means of an example[3] that path-dependent types are themselves a useful programming concept.

The example models command line options using heteregoenous maps. Say, you want to represent the settings given in the bash command line

```
ls -l --sort time --width=120
```

There are two settings: `sort` is associated with the string `"time"` and `width` is associated with the integer `120`. We'd like to store these settings in a map in a typesafe way. This map is necessarily heterogenous: if the key is `"sort"` it should accept and return a string as value, whereas for the `"width"` key it should accept and return integer values.

Here is the interface of a `HMap` trait for heterogenous maps in Scala:

```
trait Key { type Value }

trait HMap {
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value): HMap
}
```

Type dependencies are expressed with the `Key` type which contains a single type field `Value` to indicate the type of the value associated with that key. `HMap` contains two methods, `get` and `add` with result types that depend on the passed `key` argument.

To represent command line settings as `HMap` keys, we define a class `Setting` that extends `Key` and that defines a `str` field:

```
class Setting(val str: String) extends Key
```

Settings for `sort` and `width` define the setting name and the type of the values associated with the setting.

[3] The example is based on a talk by Jon Pretty (2015).

```
val sort = new Setting("sort") { type Value = String }
val width = new Setting("width") { type Value = Int }
```

The settings in the command line above can now be represented as:

```
val params = HMap.empty
  .add(width)(120)
  .add(sort)("time")
```

The system keeps track of the type associated with a setting, so the following two accesses of the `params` map both typecheck:

```
val x: Option[Int] = params.get(width)
val y: Option[String] = params.get(sort)
```

Here is a minimally complete implementation of `HMap` that makes this work:

```
trait HMap { self =>
  def get(key: Key): Option[key.Value]
  def add(key: Key)(value: key.Value) = new HMap {
    def get(k: Key) =
      if (k == key) Some(value.asInstanceOf[k.Value])
      else self.get(k)
  }
}

object HMap {
  def empty = new HMap { def get(k: Key) = None }
}
```

Note the `asInstanceOf` cast in the definition of `get`. This is necessary because the type checker cannot conclude that `k == key` implies `k.Value == key.Value`, i.e. that equality is extensional. Equality `(==)` is user-defined in Scala, so extensionality needs to be ensured in the definitions of the actual key types and values.

If we compare this to Haskell's `HMap` (van der Ploeg, 2013) we notice several differences. First, Haskell, lacking dependent method types, expresses a dependent method type with an additional polymorphic parameter. Second, the type of keys in Haskell's implementation is fixed, whereas in Scala's approach it can be augmented with new information through subclassing, as was seen in the case of `Setting`. Third, in Haskell key creation is a monadic operation, so all code manipulating `HMap` has to be placed in a monad. By contrast, the Scala code is Monad-free (and also side-effect-free).

## 7 Implementation

The DOT calculus is at the core of the new *dotty* compiler for Scala (Odersky et al., 2013), which has been under development since 2013. *dotty* is currently

used as an experimental platform to develop future versions of Scala. One of the first changes, influenced by DOT, is the introduction of true intersection types.

Unlike *nsc*, the current reference compiler for Scala, *dotty* does not maintain parameterized types and type applications. Instead, parameterized types in Scala sources are immediately mapped to types with abstract type members, and type applications are mapped to type refinements, very similar to what is shown in Section 5.

An important benefit of this reductionist approach is that it gives for the first time a satisfactory explanation of *wildcard types*, which arise when modeling type parameter variance. If we combine generics and subtyping, we inevitably face the problem that we want to express a generic type where the type argument is an unknown type that can range over a set of possible types. The prototypical case is where the argument ranges over all subtypes or supertypes of some type bound, as in `List[_ <: Fruit]` (or `List<? extends Fruit>` in Java).

Such partially undetermined types come up when we want to express variance. We would like to have that `List[Apple]` is a subtype of `List[Fruit]` since `Apple` is a subtype of `Fruit`. An equivalent way to state this is to say that the type `List[Fruit]` includes lists where the elements are of an arbitrary subtype of `Fruit`. The wildcard type `List[_ <: Fruit]` expresses that notion directly. *Definition-site variance* can be regarded as a user-friendly notation that expands into *use-site variance* expressions using wildcards.

The problem is how to model a wildcard type such as `List[_ <: Fruit]` in a formal type system. The original proposal of wildcard types by Igarashi and Viroli (Igarashi and Viroli, 2002) interpreted them as existential types. However, the implementation of the feature in Java uses subtyping rules that are not explicable using just existential types. A tentative formalization exists with WildFJ (Torgersen et al., 2004), but the issues look quite complicated. Tate, Leung and Learner have explored some possible explanations (Tate et al., 2011); however their treatment raises about as many questions as it answers.

Say, `C` is a class with type parameter `A`, which can be declared nonvariant, covariant, or contravariant. The scheme used in dotty is to model `C` as a class with a single type member

```
class C { type A }
```

A type application `C[T]` is then expressed as `C & { type A = T }` if `A` is nonvariant, as `C & { type A <: T }` if `C` is covariant and as `C & { type A >: T }` if `C` is contravariant.

Wildcard type applications are mapped as they are written. For instance, `C[_ <: T]` would be expressed as `C & { type A <: T }`. So the previously thorny issue how to model wildcards has an almost trivially simple solution in this approach.

## 8 Related Work

The DOT calculus represents the latest development in a long-standing effort to formalize the core features of the Scala programming language. Previous formalizations include the $\nu$Obj calculus (Odersky et al., 2003), Featherweight Scala (Cremet et al., 2006) and Scalina (Moors et al., 2008). The $\nu$Obj calculus features a rich type language, including distinct notions of singleton types, type selections, record types, class types and compound types. However, this richness makes $\nu$Obj rather unwieldy in practice and somewhat unsuitable as a core calculus. Furthermore, subtyping in $\nu$Obj lacks unique upper or lower bounds, and mixin composition is not commutative. Type checking in $\nu$Obj was shown to be undecidable, prompting the development of Featherweight Scala (Cremet et al., 2006) as an attempt at a calculus with decidable type checking. Scalina (Moors et al., 2008) was proposed as a formal underpinning for introducing higher-kinded types in Scala. Among these three systems, only the $\nu$Obj has been proven sound, and its proof has not been machine-verified.

A common theme among earlier systems is their complexity, which makes them hard to reason about and extend with new features. The goal of DOT is to provide a more streamlined calculus that could serve as a basis for mechanized proofs of various extensions. To this end, DOT focuses on the essence of path-dependent types and reduces complexity as far as having only variables as paths. Despite and due to such restrictions, DOT is more general than previous models as it does not assume a class-based language or any other particular choice of implementation reuse. A preliminary version of DOT was formalized using a small-step operational semantics with a store (Amin et al., 2012) without giving a soundness proof. The first mechanized soundness proof was established for a more restricted variant ($\mu$DOT) using a closure-based big-step semantics (Amin et al., 2014). Soundness results with mechanized proofs for versions of System $D_{<:}$ and DOT similar to the ones presented here (but still based on big-step semantics) were only recently established (Rompf and Amin, 2015).

Our work shares many goals with recent work on ML modules. In this context, path-dependent types go back at least to SML (Macqueen, 1986), with foundational work on transparent bindings by Harper and Lillibridge (1994) and Leroy (1994). Unlike DOT, their systems are stratified, i.e. definitions can only depend on earlier definitions in the same type. This ensures well-formedness of definitions by construction, but rules out recursive definitions. Also, type bounds are not considered. MixML (Dreyer and Rossberg, 2008) drops the stratification requirement between module and term language and enables modules as first class values as well as mixin-style recursive definitions. More recent work models key aspects of modules as extensions of System F (Montagu and Rémy, 2009; Rossberg et al., 2014). The latest development, 1ML (Rossberg, 2015), unifies the ML module and core languages through an elaboration to System $F_\omega$. Our work differs from these in its integration of subtyping and recursive definitions, as well as an overriding focus on keeping the formalism minimal.

In object-oriented programming languages, Ernst first proposed path-dependent types for family polymorphism (2001) and virtual classes (2003) in gbeta. Calculi

for virtual classes include *vc* (Ernst et al., 2006) and Tribe (Clarke et al., 2007). Virtual classes abstract not only over types, but also over classes and inheritance – requiring additional restrictions or type machinery to control interactions between statically unknown class hierarchies. In contrast, the core DOT calculus does not model inheritance by design.

Abstract type tags $\{A\colon S..T\}$ provide a form of first-class subtyping constraints in DOT. As illustrated in Section 3, this subsumes the type of bounded polymorphism found in e.g. $F_{<:}$. However, unlike $F_{<:}$, DOT allows abstract types to be lower-bounded, and even to assume absurd bounds, such as $\{A\colon \top..\bot\}$. This makes the type system more expressive, but complicates the meta theory and renders reduction under abstraction unsafe in general (see Section 4). In DOT, these issues are solved by adopting a weak-reduction strategy and ensuring type safety only in run-time typing contexts (i.e. those that correspond to a store). Cretin, Scherer and Rémy propose an alternative approach in their $F_{cc}$ and $F_{th}$ systems (Cretin and Rémy, 2014; Scherer and Rémy, 2015). They consider a more general notion of *coercion constraints* which come in two flavors: consistent constraints, under which reduction is safe, and (potentially) inconsistent constraints, under which reduction is forbidden. Their approach thus regains a more flexible reduction strategy at the expense of a considerably more complex type system.

## 9 Conclusion and Future Work

We have developed DOT, a minimal formal foundation for Scala. The calculus departs from standard practice in that it places type names related by subtyping constraints and path-dependent types at the center and regards type parameterization as a secondary feature that can be expressed through encodings.

Developing a sound meta-theory for DOT has been difficult because the issues were at first poorly understood. In retrospect, the main difficulty came from the fact that the combination of upper and lower type bounds[4] allows programmers to define their own subtyping theory and that theory might well be inconsistent. An inconsistent subtyping theory arises from *bad bounds*, where the lower bound of a type is not a subtype of its upper bound. Through transitivity of subtyping, bad bounds can turn every type into a subtype of every other type, which makes the progress part of type soundness fail. Because of recursion and intersection types, it's impossible to stratify the system so that bad bounds are ruled out *a priori*.

We solve the issue by carefully distinguishing values that can arise at run-time from other terms. Run-time values carrying types are always constructed using a $\nu$-term and its typing rule makes sure that every type member is defined as an alias of a concrete type. So run-time values cannot have bad bounds. The tricky aspect of the meta theory was how to exploit this intuition in the formal proofs.

[4] Having both kind of bounds is essential in DOT to model type aliases; if we would replace bounds by aliases we would run into the same problems.

The intuitions gained by DOT feed into Scala in several ways. For one, Scala is set to adopt constructs studied in DOT, such as full intersection types. Furthermore, the *dotty* Scala compiler has an internal type representation that generally resembles DOT and in particular encodes type parameterization as membership.

We also plan to study several extensions of DOT with the aim of bridging the gaps between the calculus and the programming language. Areas to be studied include formalizations of type parameters, variance, traits and classes, as well as inheritance. We expect that most of these extensions will be formalized through encodings into the base calculus. Their soundness will likely be established by proving type preservation of the encodings instead of adding to the meta-theory of DOT itself.

## Acknowledgments

Adriaan Moors, Donna Malayeri and Geoffrey Washburn have contributed to previous versions of DOT. We thank the anonymous reviewers of Wadlerfest for helpful comments on this paper. For insightful discussions we thank Amal Ahmed, Derek Dreyer, Erik Ernst, Matthias Felleisen, Paolo Giarrusso, Scott Kilpatrick, Grzegorz Kossakowski, Alexander Kuklev, Viktor Kuncak, Jon Pretty, Didier Rémy, Lukas Rytz, Miles Sabin, Ilya Sergey, Jeremy Siek, Josh Suereth and Phil Wadler. Contributors to the Dotty project include Dmitry Petrashko, Guillaume Martres, Vladimir Nikolaev, Ondřej Lhoták, Vera Salvisberg and Jason Zaugg. This research was supported by the European Research Council (ERC) under grant 587327 DOPPLER and the Swiss National Science Foundation under grant "Foundations of Scala".

## References

Amadio, R.M., Cardelli, L.: Subtyping recursive types. ACM Trans. Program. Lang. Syst. 15(4), 575–631 (1993)

Amin, N., Moors, A., Odersky, M.: Dependent object types. In: FOOL (2012)

Amin, N., Rompf, T., Odersky, M.: Foundations of path-dependent types. In: OOPSLA (2014)

Ariola, Z.M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: A call-by-need lambda calculus. In: POPL (1995)

Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An extension of system F with subtyping. Inf. Comput. 109(1/2), 4–56 (1994)

Clarke, D., Drossopoulou, S., Noble, J., Wrigstad, T.: Tribe: a simple virtual class calculus. In: AOSD (2007)

Coppo, M., Dezani-Ciancaglini, M., Sallé, P.: Functional characterization of some semantic equalities inside lambda-calculus. In: Automata, Languages and Programming, 6th Colloquium (1979)

Cremet, V., Garillot, F., Lenglet, S., Odersky, M.: A core calculus for Scala type checking. In: MFCS (2006)

Cretin, J., Rémy, D.: System F with coercion constraints. In: CSL-LICS (2014)

Dreyer, D., Rossberg, A.: Mixin' up the ML module system. In: ICFP (2008)
Ernst, E.: Family polymorphism. In: ECOOP (2001)
Ernst, E.: Higher-order hierarchies. In: ECOOP (2003)
Ernst, E., Ostermann, K., Cook, W.R.: A virtual class calculus. In: POPL (2006)
Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. ACM 40(1), 143–184 (Jan 1993)
Harper, R., Lillibridge, M.: A type-theoretic approach to higher-order modules with sharing. In: POPL (1994)
Igarashi, A., Viroli, M.: On variance-based subtyping for parametric types. In: ECOOP (2002)
Leroy, X.: Manifest types, modules and separate compilation. In: POPL (1994)
Macqueen, D.: Using dependent types to express modular structure. In: POPL (1986)
Montagu, B., Rémy, D.: Modeling abstract types in modules with open existential types. In: POPL (2009)
Moors, A., Piessens, F., Odersky, M.: Safe type-level abstraction in Scala. In: FOOL (2008)
Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A nominal theory of objects with dependent types. In: ECOOP (2003)
Odersky, M., Petrashko, D., Martres, G., others.: The dotty project (2013), `https://github.com/lampepfl/dotty`
van der Ploeg, A.: The HMap package. `https://hackage.haskell.org/package/HMap` (2013)
Pretty, J.: Minimizing the slippery surface of failure. Talk at Scala World (2015), `https://www.youtube.com/watch?v=26UHdZUsKkE`
Rompf, T., Amin, N.: From F to DOT: Type soundness proofs with definitional interpreters. Tech. rep., Purdue University (2015), `http://arxiv.org/abs/1510.05216`
Rossberg, A.: 1ML - core and modules united (f-ing first-class modules). In: ICFP (2015)
Rossberg, A., Russo, C.V., Dreyer, D.: F-ing modules. J. Funct. Program. 24(5), 529–607 (2014)
Scherer, G., Rémy, D.: Full reduction in the face of absurdity. In: ESOP (2015)
Tate, R., Leung, A., Lerner, S.: Taming wildcards in java's type system. In: PLDI (2011)
Torgersen, M., Ernst, E., Hansen, C.P.: WildFJ. In: FOOL (2004)

# A Additional Rules and Detailed Proofs

## A.1 Typing of Dependent Sums / $\Sigma$s

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.

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

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

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

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

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

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 of types, 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 ($F_{<:}$) typing derivations. The case for subsumption follows immediately from preservation of subtyping (Theorem 1). The only remaining non-trivial cases are type and term application.

*Term application case.* We need to show that

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

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

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

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

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

The result follows by applying the induction hypothesis once more:

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

*Type application case.* By preservation of subtyping, we have $\Gamma^* \vdash U^* <: S^*$, hence it suffices to show that

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

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

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

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 Lemma 6

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

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

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

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.