Towards Strong Normalization for Dependent Object Types (DOT)*

Fei Wang[1] **and Tiark Rompf**[2]

1 **Purdue University, West Lafayette, USA**
`wang603@purdue.edu`

2 **Purdue University, West Lafayette, USA**
`firstname@purdue.edu`

Abstract

The Dependent Object Types (DOT) family of calculi has been proposed as a new theoretic foundation for Scala and similar languages, unifying functional programming, object oriented programming and ML-style module systems. Following the recent type soundness proof for DOT, the present paper aims to establish stronger metatheoretic properties. The main result is a fully mechanized proof of strong normalization for $D_{<:}$, a variant of DOT that excludes recursive functions and recursive types. We further discuss techniques and challenges for adding recursive types while maintaining strong normalization, and demonstrate that certain variants of recursive self types can be integrated successfully.

1998 ACM Subject Classification D.3.3 [Programming Languages]: Language Constructs and Features

Keywords and phrases Scala, DOT, strong normalization, logical relations, recursive types

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.27

Supplementary Material ECOOP Artifact Evaluation approved artifact available at `http://dx.doi.org/10.4230/DARTS.3.2.5`

1 Introduction

The Dependent Object Types (DOT) calculus [8, 47, 7] aims to be a uniform foundation for modern expressive languages that combine features from traditional object-oriented languages, functional languages, and ML-style module systems.

After many years of false starts, a recent breakthrough in the study of DOT's metatheory established the key property of type soundness [47], which states that any well-typed program either diverges or evaluates to a (properly typed) value. Thus, type soundness guarantees the absence of runtime errors, as captured by the slogan "well-typed programs don't go wrong".

In this paper, we investigate another key metatheoretic property: strong normalization, which states that any well-typed program evaluates to a (properly typed) value. Thus, strong normalization implies type soundness, but in addition to excluding runtime errors, it excludes the option of divergence: all well-typed programs must terminate. Standard proof methods for type soundness do not scale to termination results, and hence, more involved proof techniques are needed. It is also clear that strongly-normalizing languages cannot be Turing-complete. Hence, some restrictions on the language are necessary to ensure termination.

* This research was supported by NSF through awards 1553471 and 1564207.

A key contribution of this paper is to show that the one important restriction needed in DOT is to prevent the *creation* of recursive type values. In particular, we can still include DOT's flavor of recursive self types without giving up on strong normalization. This result is surprising, because adding traditional recursive types to simply-typed λ-calculus or System F leads to Turing-completeness.

Why does strong normalization matter? It is well known from previous work that type soundness of Turing-complete DOT versions hinges on the termination of *path expressions* p that are used in path-dependent types $p.\text{Type}$. In fact, Scala has documented soundness bugs related to path expressions such as `lazy vals` which are *not* guaranteed to terminate [47]. Hence, studying termination properties of DOT-like calculi in a formal setting is a stepping-stone for future type system extensions of DOT, for example towards higher-kinded types and type lambdas [38].

This paper is structured around its individual contributions:

- We review System $D_{<:}$, its relation to $F_{<:}$ and to DOT and Scala, as well as the previous type soundness result (Section 2).
- We present our strong normalization proof for $D_{<:}$ in full detail. The proof method follows the standard Girard-Tait approach based on logical relations [31, 54]. The key challenge in adapting proof techniques from $F_{<:}$ and similar systems lies in the handling of bounded first-class type values (Section 3).
- We scale our proof from $D_{<:}$ towards DOT. We adapt the proof method to include intersection types, which are used in DOT to model type refinement, and we clarify the boundary between strongly normalizing and Turing-complete systems, where the key challenge lies in handling DOT's recursive self types. We first show that, consistent with our expectations from similar systems, recursive *type values* are enough to encode fixpoint combinators and lead to a Turing-complete language. But surprisingly, with only non-recursive type *values*, we can still add recursive self types to the calculus and maintain strong normalization (Section 4).

Our mechanized Coq proofs are available from:
`https://github.com/tiarkrompf/minidot/tree/master/ecoop17`

2 Background: System $D_{<:}$

We base our description on a formal model situated inbetween $F_{<:}$ and full DOT, called System $D_{<:}$ [9]. Like DOT, $D_{<:}$ has abstract type members and path-dependent type selections. But in constrast to full DOT, which represents all values as objects with method and type members, it has separate forms for dependent functions and first-class type values, and it lacks recursive types.

2.1 Syntax and Typing Rules

System $D_{<:}$ is at its core a system of first-class type objects and path-dependent types. Type objects can be seen as single-field records containing an abstract type member. Type selections, or path-dependent types serve to access these abstract type members.

The syntax and typing rules are shown in Figure 2, after reviewing those of System $F_{<:}$ in Figure 1. The type language includes $\bot$ and $\top$, as least and greatest element of the subtyping relation, first-class abstract types (Type $T_1..T_2$), lower-bounded by T_1 and upper-bounded by T_2, type selections on a variable $x.\text{Type}$ (i.e., path-dependent types), where x is a term variable bound to a type object, and finally dependent function types $(x : T) \to T^x$. The

Syntax

$$\begin{array}{lcl} T & ::= & X \mid \top \mid T \to T \mid \forall X <: T.T^X \\ t & ::= & x \mid \lambda x : T.t \mid \Lambda X <: T.t \mid t\ t \mid t\ [T] \\ \Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, X <: T \end{array}$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\Gamma \vdash T <: \top$$

$$\Gamma \vdash X <: X$$

$$\frac{\Gamma \ni X <: U \qquad \Gamma \vdash U <: T}{\Gamma \vdash X <: T}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \ , \ T_1 <: T_2}{\Gamma \vdash (S_1 \to T_1) <: (S_2 \to T_2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S_2 <: S_1 \\ \Gamma, X <: S_2 \vdash T_1^X <: T_2^X\end{array}}{\Gamma \vdash (\forall X <: S_1.T_1^X) <: (\forall X <: S_2.T_2^X)}$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3}$$

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

$$\frac{\Gamma \ni x : T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash (\lambda x : S.t) : (S \to T)}$$

$$\frac{\Gamma \vdash t_1 : (S \to T) \ , \ t_2 : S}{\Gamma \vdash t_1\ t_2 : T}$$

$$\frac{\Gamma, X <: S \vdash t : T^X}{\Gamma \vdash (\Lambda X <: S.t) : (\forall X <: S.T^X)}$$

$$\frac{\Gamma \vdash t_1 : (\forall X <: U.T^X) \ , \ T_2 <: U}{\Gamma \vdash t_1[T_2] : T^{T_2}}$$

$$\frac{\Gamma \vdash t : S \ , \ S <: T}{\Gamma \vdash t : T}$$

Figure 1 System F$_{<:}$: syntax and typing rules. The notation T^X denotes that variable X may occur free in T. Occuring in the same rule, T^U denotes T with all occurrences of X replaced with U. Types are otherwise assumed to be closed with respect to the environment.

notation T^x denotes that term variable x may occur free in T. The term language includes variables x, creation of type objects (Type T), λ-abstractions $\lambda x.t$, and applications $t_1\ t_2$.

The subtyping relation can compare type selections with the bounds of the underlying abstract types, and compare type objects and dependent functions, respectively. Type assignment contains fairly standard cases for dependent abstraction and application.

To relate System D$_{<:}$ to Scala, let us take a step back and consider two ways to define a standard `List` data type:

```
class List[E]          // parametric, functional style
class List { type E }  // modular style, w. type member
```

The first one is the standard parametric version. The second one defines the element type `E` as a type member, which can be referenced using a path-dependent type. To see the difference in use, here are the two respective signatures of a standard `map` function:

```
def map[E,T](xs: List[E])(fn: E => T): List[T] = ...
def map[T]  (xs: List)(fn: xs.E => T): List & { type E = T } = ...
```

Again, the first one is the standard parametric version. The second one uses the path-dependent type `xs.E` to denote the element type of the particular list `xs` passed as argument,

Syntax

$$
\begin{array}{lcl}
T & ::= & \bot \mid \top \mid \text{Type } T..T \mid x.\text{Type} \mid (x : T) \to T^x \\
t & ::= & x \mid \text{Type } T \mid \lambda x.t \mid t\ t \\
\Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}
$$

Subtyping $\boxed{\Gamma \vdash S <: U}$

$$\Gamma \vdash \bot <: T \ (\textsc{SBot}) \qquad \Gamma \vdash T <: \top \ (\textsc{STop})$$

$$\frac{\Gamma \vdash T_1 <: T_2 \qquad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \ (\textsc{STrans})$$

$$\Gamma \vdash x.\text{Type} <: x.\text{Type} \quad (\textsc{SSelX})$$

$$\frac{\Gamma \vdash x : \text{Type } T..\top}{\Gamma \vdash T <: x.\text{Type}} \quad (\textsc{SSel1})$$

$$\frac{\Gamma \vdash x : \text{Type } \bot..T}{\Gamma \vdash x.\text{Type} <: T} \quad (\textsc{SSel2})$$

$$\frac{\Gamma \vdash S_2 <: S_1 \ , \ U_1 <: U_2}{\Gamma \vdash \text{Type } S_1..U_1 <: \text{Type } S_2..U_2} \ (\textsc{STyp})$$

$$\frac{\begin{array}{c}\Gamma \vdash S_2 <: S_1 \\ \Gamma, x : S_2 \vdash U_1^x <: U_2^x\end{array}}{\begin{array}{c}\Gamma \vdash (x : S_1) \to U_1^x <: \\ (x : S_2) \to U_2^x\end{array}} \quad (\textsc{SFun})$$

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

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\textsc{TVar})$$

$$\Gamma \vdash \text{Type } T : \text{Type } T..T \quad (\textsc{TTyp})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2^x}{\Gamma \vdash \lambda x.t : (x : T_1) \to T_2^x} \quad (\textsc{TAbs})$$

$$\frac{\Gamma \vdash t : (x : T_1) \to T_2^x \ , \ y : T_1}{\Gamma \vdash t\ y : T_2^y} \ (\textsc{TDApp})$$

$$\frac{\Gamma \vdash t_1 : (x : T_1) \to T_2 \ , \ t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2} \ (\textsc{TApp})$$

$$\frac{\Gamma \vdash t : T_1 \ , \ T_1 <: T_2}{\Gamma \vdash t : T_2} \quad (\textsc{TSub})$$

Figure 2 System $D_{<:}$: a generalization of $F_{<:}$ with type values and path-dependent types. A type x.Type refers to the type "within" x (i.e. path dependent type). The notation T^x denotes that variable x may occur free in T. Types are otherwise assumed to be closed with respect to the environment.

and uses a refined type `List & { type E = T }` to define the result of `map`. Such refined types are included in DOT, but absent in $D_{<:}$.

It is easy to see how the modular surface syntax directly maps to the formal $D_{<:}$ syntax, if we express fully abstract types `{ type E }` as (Type $\bot..\top$) and concrete type aliases `{ type E = T }` as (Type $T..T$). It is also important to note that the modular style with first-class type objects can directly encode the functional style, which corresponds to bounded parametric polymorphism as in System $F_{<:}$, but with increased expressiveness due to the $\bot$ type and potential lower bounds on type variables.

Runtime Structures

H	$::=$	$\emptyset \mid H, x : v$	Runtime environments
v	$::=$	$\langle H, \lambda x.t\rangle \mid \langle H, \text{Type } T\rangle$	Runtime values
r	$::=$	Timeout | Done (Error | Val v)	Interpreter results

Definitional Interpreter

```
(* Some Coq data types and auxiliary functions elided *)
Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}: option (option vl) :=
  DO n1 ⇐ FUEL n;                                  (* totality: n1<—n−1, TIMEOUT if n=0 *)
  match t with
     | tvar x      ⇒ DONE (lookup x env)      (* variable  x                          *)
     | ttyp T      ⇒ DONE (VAL (vty env T))  (* type value Type T ⇒ ⟨H, Type T⟩       *)
     | tabs x ey   ⇒ DONE (VAL (vabs env x ey)) (* lambda      λx.e_y  ⇒ ⟨H, λx.e_y⟩    *)
     | tapp ef ex ⇒                            (* application e_f e_x                   *)
       DO vf ⇐ eval n1 env ef;
         DO vx ⇐  eval n1 env ex;
         match vf with
           | (vabs env2 x ey) ⇒
             eval n1 ((x,vx)::env2) ey
           | _ ⇒ ERROR
         end
  end.
```

■ **Figure 3** System D$_{<:}$: Operational Semantics.

2.2 Operational Semantics

The operational semantics of D$_{<:}$ follows the standard call-by-value λ-calculus evaluation rules very closely. We can give a formal semantics in many different ways. We follow previous work [9] in using an environment-based functional evaluator, which serves as a *definitional interpreter* in the style of Reynolds [46]. A substitution-free semantics is attractive in the case of DOT, mainly because term substitution requires additional mechanics in the metatheory to properly handle type selections: in the surface syntax, $[v/x](x.\text{Type}) = v.\text{Type}$ is not a legal type. However, one can freely switch between environment-based and substition-based, as well as big-step and small-step semantics following the interderivation techniques of Danvy et al. [20, 21, 2].

Figure 3 shows both the definition of runtime values and the definition of the evaluator. We opt to show the evaluator in actual Coq code. The only case that is different from a call-by-value λ-calculus evaluator is the case that evaluates first-class type expressions Type T to a form of type closure $\langle H, \text{Type } T\rangle$.

The other aspect that is worth noting about our evaluator is that it is a total function, by virtue of inheriting totality from its defining language, Coq. The evaluator takes a fuel value n and distinguishes explicitly between `Timeout`, `Error`, and value results. The `FUEL` operation in the first line desugars to a simple non-zero check:

```
match n with
  | z ⇒ TIMEOUT
  | S n1 ⇒ ...
end
```

The fuel value upper-bounds the number of steps the evaluator may take and can thus serve as induction measure to prove properties about evaluation.

2.3 Previous Work: Type Soundness

To prove type soundness for $D_{<:}$, previous work by Amin and Rompf [9] followed a technique of Siek [50] and Ernst, Ostermann and Cook [25], which consists in using the numeric fuel value as induction measure. Similar techniques have recently been proposed by Owens et al. [43].

▶ **Theorem 1** (Type soundness for $D_{<:}$). *If* `eval` *does not time out, it returns a well-typed value:* [1]

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \qquad \mathit{eval}\ k\ H\ t = \mathit{Done}\ r}{r = \mathit{Val}\ v \qquad H \vdash v : T}$$

Proof. By induction on the fuel value k. Note that $\Gamma \vDash H$ means that Γ is well-formed with H, i.e. the two environments are of the same length and values in H have corresponding types in Γ. ◀

The proof has some complications compared to well-documented proofs for $F_{<:}$, caused by the fact that lower-bounded type members, may lead to transitivity chains $T_1 <: x.T <: T_2$ with a type selection in the middle, whereas in $F_{<:}$, only upper-bounded type variables $X <: T$ can occur. These issues are described in detail in previous work [10, 47, 9].

It is important to note that soundness becomes quite a bit more complicated once recursive types are added in full DOT [47].

2.4 Type Soundness Hinges on Strong Normalization of Paths

The soundness of DOT hinges on the fact that path terms p in type selections p.Type are strongly normalizing. For this reason, current soundness results only cover type selection on variables x.Type. Identifying larger terminating fragments of DOT lays the basis for future extensions towards richer path expressions, and therefore, more general notions of dependent types.

To see why termination of path expressions is important, it is necessary to realize that one cannot, in general, enforce "good bounds" for all types occuring in a given program [10]. This means that for a type (Type $T_1..T_2$), we need to accept that we cannot statically guarantee that $T_1 <: T_2$. The reason is that this property is not preserved by intersection types, which play a key role in DOT to model type refinement. Hence, DOT enforces this property in a syntactic way, by allowing type values to only contain type aliases (Type $T..T$). This means that we only accept that a type has "good bounds" if it is inhabited. A transitivity chain $T_1 <: p.T <: T_2$ is only safe if evaluation of p terminates with a unique value.

Non-termination of path-expressions or evaluation to non-values (through lazy vals, type projections, or `null` values) is a recurring source of soundness bugs in the production Scala language and compiler [11, 47].

3 Strong Normalization

We present our strong normalization proof for $D_{<:}$ in detail. Instead of assuming eval k H t in the premise of Theorem 1, we now want to derive $\exists$ k. eval k H t in the conclusion.

[1] In a slight abuse of notation, we will sometimes use inference rule notation in this paper to state lemmas and theorems. This is just to make the formulas easier to parse and avoid spelling out all $\forall/\exists$ quantifiers.

▶ **Definition 2** (Strong Normalization). Any well-typed term evaluates to a well-typed value:

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H}{\text{eval } k \ H \ t = \text{Done Val } v \qquad H \vdash v : T}$$

We have fixed a deterministic call-by-value evaluation strategy, since it is known from previous work that arbitrary reductions already violate type soundness [9]. In this setting, strong normalization can be taken as a synonym for termination.[2] Under non-deterministic evaluation strategies, one distinguishes between strong and weak normalization: strong normalization requires that all possible evaluations of a given term terminate with its normal form. Weak normalization only requires that every term *has* a normal form, which can be reached through *some* evaluation path.

3.1 The Girard-Tait Proof Method: Starting-Point $F_{<:}$

The standard approach of proving termination is the method of Girard and Tait [31, 54]. For every type T we define its denotation $[\![T]\!]$ as the set of values that inhabit T, with type-specific characterics that carry the key inductive properties of the main proof. The judgement $v : T$ then becomes $v \in [\![T]\!]$. Based on these sets of values, we can also define sets of terms t (paired with runtime environment H):

$$\mathcal{E}[\![T]\!] = \{\langle H, t\rangle \mid \exists k, v.\ \text{eval } k \ H \ t = \text{Done Val } v \wedge v \in [\![T]\!]\}$$

that evaluate to a value of type T in environment H, in a certain number of steps k.

Standard proofs for a variety of type systems such as System F, $F_{<:}$, and F-bounded can be found in the literature [28, 39]. As we will see, adapting this proof technique for $D_{<:}$ from $F_{<:}$ and similar systems is not entirely trivial. The key challenge lies in handling bounded first-class type values, which are absent in $F_{<:}$. Nevertheless, it is instructive to look at this simpler setting first. The syntax and typing rules for $F_{<:}$ are reviewed in Figure 1.

The semantic interpretation of types, $[\![.]\!]$, can be defined as:

$$\begin{array}{lcl}
[\![\top]\!]_\rho & = & \{v\} \text{ i.e. set of all values} \\
[\![X]\!]_\rho & = & \rho(X) \\
[\![T_1 \to T_2]\!]_\rho & = & \{\langle H, \lambda x.t\rangle \mid \forall v_x \in [\![T_1]\!]_\rho.\ \langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2]\!]_\rho\} \\
[\![\forall X <: T_1.T_2^X]\!]_\rho & = & \{\langle H, \Lambda X.t\rangle \mid \forall D \subseteq [\![T_1]\!]_\rho.\ \langle H,\ t\rangle \in \mathcal{E}[\![T_2^X]\!]_{\rho(X\mapsto D)}\}
\end{array}$$

The definition of $[\![.]\!]$ is well-founded, since $[\![.]\!]$ is only used on proper subterms on the right hand sides, including indirectly through $\mathcal{E}[\![.]\!]$. The denotation of $[\![\top]\!]$ is the set of all values. To handle type variables X, $[\![.]\!]$ is parameterized over a context ρ which maps names to sets of values. Note that ρ and H have different types and they are not interchangable. The definition of $[\![T_1 \to T_2]\!]$ captures the essential statement of the termination theorem as it applies to functions: if there is a function argument value of the right type, then evaluation of the function body will terminate after some number of steps and produce a result value of the right type. Also note that $F_{<:}$ does not usually have a bottom type $\bot$, but one can naturally define $[\![\bot]\!]_\rho = \emptyset$.

Subtyping is inherently tied to a narrowing property for $[\![.]\!]_\rho$, i.e., the ability to replace a binding in ρ with a subtype. However we cannot prove this directly, since $[\![.]\!]_\rho$ is used

[2] Our use of the term "strong normalization" is consistent with that of McAllester et al. [39], who also used a (partial) evaluation function in their proof of strong normalization for System F_2 and F_ω.

recursively in a contravariant position for function arguments. Hence, the case for $\forall$ types has narrowing "built-in" via $\forall D \subseteq [\![T_1]\!]_\rho$.

To complete the termination proof, a key lemma is needed to model the subsumption case, and interpret the subtyping relation in a semantic way ($\vDash$ is a consistency relation; $\Gamma \vDash H \sim \rho$ means that $\Gamma(x) = T$ implies $H(x) \in \rho(x) \subseteq [\![T]\!]_\rho$):

▶ **Lemma 3** (Semantic Widening). *If* $\Gamma \vDash H \sim \rho$ *and* $\Gamma \vdash T_1 <: T_2$*, then* $[\![T_1]\!]_\rho \subseteq [\![T_2]\!]_\rho$

Proof. By induction on the subtyping derivation. ◀

We can interpret Lemma 3 equivalently as a widening or closure property: if $v \in [\![T_1]\!]_\rho$ and $\Gamma \vdash T_1 <: T_2$ then $v \in [\![T_2]\!]_\rho$. Additional lemmas about environment extension and shrinkage (weakening and strenghtening) as well as about type substitution are needed as well. With these helper lemmas, we can complete the desired theorem:

▶ **Theorem 4** (Strong Normalization for $\mathrm{F}_{<:}$). *Any well-typed term evaluates to a well-typed value:*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \sim \rho}{\langle H, t\rangle \in \mathcal{E}[\![T]\!]_\rho}$$

Proof. By induction on the typing derivation. ◀

In particular, Theorem 4 holds for closed terms, in empty environments Γ and H.

It is worth noting that we assume *lenient well-formedness* throughout. All free variables of syntactically valid forms (types or terms) are bound in environments. This assumption is implicit in all definitions, lemmas and theorems, unless a free variable is explicitly mentioned, as in T^x.

3.2 System $\mathrm{D}_{<:}$: Type Values and Bounds

For $\mathrm{D}_{<:}$, we encounter key difficulties when defining $[\![.]\!]$. A first straightforward attempt inspired by $\mathrm{F}_{<:}$ and adapted to path-dependent types might look like this:

$$\begin{array}{lcl}
[\![\top]\!]_\rho & = & \{v\} \\
[\![\bot]\!]_\rho & = & \emptyset \\
[\![\mathrm{Type}\ T_1..T_2]\!]_\rho & = & \{\langle H, \mathrm{Type}\ \ T\rangle \mid [\![T_1]\!]_\rho \subseteq [\![T_2]\!]_\rho\} \\
[\![x.\mathrm{Type}]\!]_\rho & = & \rho(x) \\
[\![(x : T_1) \to T_2^x]\!]_\rho & = & \{\langle H, \lambda x.t\rangle \mid \forall D \subseteq [\![T_1]\!]_\rho.\ \forall v_x \in D. \\
\langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2^x]\!]_{\rho(x\mapsto D)}\} & &
\end{array}$$

However, this can't be right: consider the case where we have a function type, and $D = [\![\mathrm{Type}\ L..U]\!] \subseteq [\![T_1]\!]$. We add D to the environment ρ, but when it is picked up by some x.Type, we end up comparing again to the type of the binding $[\![\mathrm{Type}\ L..U]\!]$, but we need to compare with the upper bound $[\![U]\!]$ instead! This behavior is dictated by the (SSel2) subtyping rule from Figure 2:

$$\frac{\Gamma \vdash x : \mathrm{Type}\ \bot..U}{\Gamma \vdash x.\mathrm{Type} <: U} \qquad (\textsc{SSel2})$$

The semantic widening Lemma 3 needs to map this rule to $[\![x.\mathrm{Type}]\!] \subseteq [\![U]\!]$, and hence show that for any value, $v \in \rho(x)$ implies $v \in [\![U]\!]$. But unfortunately, we have no way to show this, as $\rho(x)$ is mapped to $[\![\text{Type } \bot..U]\!]$. Attempts to extract the bounds syntactically from a given type, directly or indirectly, fail at various stages in the proof. To solve this problem, we extend the definition of $[\![T]\!]$ to cover not only the type T itself but also its bounds. We let $[\![T]\!]^{0}$ denote the values that inhabit T, $[\![T]\!]^{\mathsf{U}}$ the values that inhabit the upper bound of T, $[\![T]\!]^{\mathsf{UU}}$ the upper bound of the upper bound, and so on. In the example above, we can now access $[\![U]\!]^{0}$ via $\rho(x)^{\mathsf{U}} = [\![\text{Type } \bot..U]\!]^{\mathsf{U}}$. However, this is not quite enough. Consider the case in the semantic widening Lemma 3 that interprets the (SSel1) subtyping rule:

$$\frac{\Gamma \vdash x : \text{Type } L..\top}{\Gamma \vdash L <: x.\text{Type}} \qquad (\textsc{SSel1})$$

Lemma 3 needs to map this rule to $[\![L]\!] \subseteq [\![x.\text{Type}]\!]$. Now we would have to show that $v \in [\![L]\!]$ implies $v \in \rho(x)$, so we need to track lower bounds, too. Just like with upper bounds, we identify $[\![L]\!]^{0}$ with $\rho(x)^{\mathsf{L}} = [\![\text{Type } L..\top]\!]^{\mathsf{L}}$. We need an additional property that $\rho(x)^{\mathsf{L}} \subseteq \rho(x)^{\mathsf{U}}$ to complete this case of the lemma. The following definitions make all this more precise:

▶ **Definition 5** (Indexed Value Sets). We index the value sets as $[\![T]\!]^{\mathsf{B}*}$, where $\mathsf{B}*$ is a possibly empty list of bound selectors B that can be either U (upper bound) or L (lower bound). We use 0 to denote the empty list explicitly.

▶ **Definition 6** (Polarity of Bound Selectors). Let pos $\mathsf{B}* =$ true if the number of L in $\mathsf{B}*$ is even, false otherwise. We also write $\mathsf{B}+$ to denote a positive sequence of bound specifiers (pos $=$ true) and $\mathsf{B}-$ a negative one (pos $=$ false).

▶ **Definition 7** (Indexed Value Set Inclusion). An indexed value set D_1 is smaller or equal than D_2, written $D_1 \sqsubseteq D_2$, iff

$$\forall \mathsf{B}+.\ D_1^{\mathsf{B}+} \subseteq D_2^{\mathsf{B}+} \qquad \wedge \qquad \forall \mathsf{B}-.\ D_2^{\mathsf{B}-} \subseteq D_1^{\mathsf{B}-}$$

To add some intuition to this definition, consider the case where $T_1 <: T_2 <: T_3 <: T_4$. Then Type $T_2..T_3 <:$ Type $T_1..T_4$, based on our subtyping rule. Regard D_2 as $[\![\text{Type } T_1..T_4]\!]$, and D_1 as $[\![\text{Type } T_2..T_3]\!]$. Then intuitively, $D_1 \sqsubseteq D_2$, which makes sense when we see that $D_1^{\mathsf{U}} \subseteq D_2^{\mathsf{U}}$, and $D_2^{\mathsf{L}} \subseteq D_1^{\mathsf{L}}$, because $D_1^{\mathsf{U}} = [\![T_3]\!]$ and $D_2^{\mathsf{U}} = [\![T_4]\!]$, and $D_1^{\mathsf{L}} = [\![T_2]\!]$ and $D_2^{\mathsf{L}} = [\![T_1]\!]$.

▶ **Definition 8** (Good bounds). An indexed value set D has "good bounds", written GoodBounds D, iff for all $\mathsf{A}*$ such that $D^{\mathsf{A}*} \neq \emptyset$ we have:

$$\forall \mathsf{B}+.\ D^{\mathsf{A}*\mathsf{L}\mathsf{B}+} \subseteq D^{\mathsf{A}*\mathsf{U}\mathsf{B}+} \qquad \wedge \qquad \forall \mathsf{B}-.\ D^{\mathsf{A}*\mathsf{U}\mathsf{B}-} \subseteq D^{\mathsf{A}*\mathsf{L}\mathsf{B}-}$$

To add some intuition to this definition, consider $D^{\mathsf{A}*\mathsf{L}}$ as D_1 and $D^{\mathsf{A}*\mathsf{U}}$ as D_2. Intuitively, $D^{\mathsf{A}*\mathsf{L}} \sqsubseteq D^{\mathsf{A}*\mathsf{U}}$. Then applying definition 7 to D_1 and D_2 gives us definition 8.

The switching of polarity is necessary to account for contravariance in lower-bound comparisions, in accordance with the (Styp) subtyping rule. Note that the definition of "good bounds" is lenient with respect to empty sets, which correspond to uninhabited types.

With these auxiliary definitions at hand, we can define the value type relation $[\![\cdot]\!]$ for $\text{D}_{<:}$:

▶ **Definition 9** (Value Type Relation).

$$
\begin{array}{lll}
[\![\top]\!]^{\mathsf{B+}}_\rho & = & \{v\} \\
[\![\top]\!]^{\mathsf{B-}}_\rho & = & \emptyset \\
[\![\bot]\!]^{\mathsf{B+}}_\rho & = & \emptyset \\
[\![\bot]\!]^{\mathsf{B-}}_\rho & = & \{v\} \\
\\
[\![\text{Type } T_1..T_2]\!]^{\mathsf{0}}_\rho & = & \{\langle H, \text{Type } \ T\rangle \mid [\![T_1]\!]_\rho \sqsubseteq [\![T_2]\!]_\rho\} \\
[\![\text{Type } T_1..T_2]\!]^{\mathsf{UB*}}_\rho & = & [\![T_2]\!]^{\mathsf{B*}}_\rho \\
[\![\text{Type } T_1..T_2]\!]^{\mathsf{LB*}}_\rho & = & [\![T_1]\!]^{\mathsf{B*}}_\rho \\
\\
[\![x.\text{Type}]\!]^{\mathsf{B*}}_\rho & = & \rho(x)^{\mathsf{UB*}} \\
\\
[\![(x : T_1) \to T_2^x]\!]^{\mathsf{0}}_\rho & = & \{\langle H, \lambda x.t\rangle \mid \forall D. D \sqsubseteq [\![T_1]\!]_\rho \wedge \text{GoodBounds } D \Rightarrow \\
& & \quad \forall v_x \in D^{\mathsf{0}}.\ \langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2^x]\!]^{\mathsf{0}}_{\rho(x\mapsto D)}\} \\
[\![(x : T_1) \to T_2^x]\!]^{(\mathsf{B+}\ \neq\ \mathsf{0})}_\rho & = & \{v\} \\
[\![(x : T_1) \to T_2^x]\!]^{(\mathsf{B-})}_\rho & = & \emptyset \\
\\
\mathcal{E}[\![T]\!]^{\mathsf{B*}}_\rho & = & \{\langle H, t\rangle \mid \exists k, v.\ \text{eval } k\ H\ t = \text{Done Val } v \wedge v \in [\![T]\!]^{\mathsf{B*}}_\rho\}
\end{array}
$$

The interpretation of $\top$ includes all values, and the upper bound of $\top$, and in fact all positive deeper bounds are again equal to $\top$. Its negative bounds are not inhabited: they correspond to the definition of type $\bot$. All positive bounds of $\bot$ are empty, and thus equal to $\bot$ itself. The lower bound of $\bot$, and all other negative bounds, are equal to $\top$. The interpretation of Type $T_1..T_2$ requires the bounds T_1 and T_2 to be properly ordered, and can extract the corresponding bound for selectors $\mathsf{UB*}$ and $\mathsf{LB*}$. Note that to keep the definition well-founded, no restrains are given for the relationship between T and T_1, T_2, and none are needed. This somewhat surprising scheme works essentially due to a type erasure property (types are not required to be represented at runtime). We will see an alternative model in Section 4. Type selections $x.T$ are mapped to the upper bound of the type stored in the context, in accordance with subtyping rule (SSEL2). Function types are interpreted as expected for the base type, and have lower bound $\bot$ and upper bound $\top$. This is to ensure that every type has *some* bounds.

The definition of $\mathcal{E}[\![.]\!]$ is as before. If within some steps k, a term t evaluates to some value v in an evironment H, and v belongs to the set of values that inhabits type T with context ρ (i.e. $v \in [\![T]\!]^{\mathsf{0}}_\rho$), then the pair $\langle H, t\rangle$ is a member of the logical relation $\mathcal{E}[\![T]\!]^{\mathsf{0}}_\rho$. Bound selectors other than $\mathsf{0}$ are analagous.

We prove a couple of straightforward structural lemmas, which we will use at various later points:

▶ **Lemma 10** (Weakening/Strengthening). *The value type relation is invariant under extending and shrinking the context:*

$$\frac{x \notin FV(T)}{[\![T]\!]^{\mathsf{B*}}_\rho = [\![T]\!]^{\mathsf{B*}}_{\rho(x\mapsto D)}}$$

Proof. By induction on the size of T. ◀

▶ **Lemma 11** (Substitution). *The value type relation is invariant under substitution of bound variables that map to equivalent type sets:*

$$\frac{\rho(x) = \rho(y)}{[\![T^x]\!]^{\mathsf{B*}}_\rho = [\![T^y]\!]^{\mathsf{B*}}_\rho}$$

Proof. By induction on the size of T. ◀

▶ **Definition 12** (Consistent Environments). A type environment Γ, a value environment H, and a value typing context ρ are consistent, written, $\Gamma \vDash H \sim \rho$, iff they contain exactly the same bindings and the following proposition holds:

$$\frac{\Gamma(x) = T}{H(x) \in \rho(x)^0 \ \wedge\ \rho(x) \sqsubseteq [\![T]\!]_\rho \ \wedge\ \text{GoodBounds}\ \rho(x)}$$

We also use the notation $\Gamma \vDash \rho$ when we do not need to refer to a specific value environment H, but assume that a suitable one exists. The strong similarity between consistent environments and the definition of $[\![(x : T_1) \to T_2^x]\!]^0_\rho$ is no coincidence. We need to maintain this correspondence, so that when the environment is extended with new bindings for a $\lambda x.y$ term, the consistency of the involved (type, value, and value typing) environments is retained. We formulate this capability as an auxiliary structural lemma:

▶ **Lemma 13** (Extending Consistent Environments).

$$\frac{\Gamma \vDash H \sim \rho \qquad v \in D^0 \qquad D \sqsubseteq [\![T]\!]_{\rho(x \mapsto D)} \qquad \text{GoodBounds}\ D}{(\Gamma, x : T) \vDash (H, x : v) \sim (\rho, x : D)}$$

Proof. By straightforward case distinction on the target index y. If $y = x$, i.e. y refers to the newly added T, v, and D in the three respective environments, then the provided premises are just right for the goal. If $y \neq x$, i.e. y refers to older respective entries, then necessary evidence can be obtained from $\Gamma \vDash H \sim \rho$, with the help of Lemma 10. ◀

3.3 Good Bounds

We are now ready to prove our first semantically meaningful lemma:

▶ **Lemma 14** (Good Bounds). *In a consistent environment, all types have good bounds:*

$$\frac{\Gamma \vDash \rho}{\text{GoodBounds}\ [\![T]\!]_\rho}$$

Proof. By induction on T. The cases for $\top$, $\bot$, and for function types are solved by contradiction, since either the type itself or the lower bound in question is not inhabited. The case for type selections $x.\text{Type}$ uses the consistent environment rule, which states that all value sets D in ρ have the *GoodBounds* property. Case for type values Type $T_1..T_2$ requires a case distinction on the bound selectors $\mathsf{B}*$. If $\mathsf{B}*$ is $\mathsf{0}$, the result follows immediately from the definition of $[\![.]\!]$. If $\mathsf{B}*$ is $\mathsf{L} :: \mathsf{B}'*$ or $\mathsf{U} :: \mathsf{B}'*$, the result follows from the inductive hypothesis, either for the type of the lower bound T_1 or the type of the upper bound T_2, respectively. ◀

3.4 Semantic Subtyping

As already discussed in Section 3.1 for $F_{<:}$, we need a key lemma that provides a semantic interpretation of the syntactic subtyping relation. This semantic widening or subsumption lemma for $D_{<:}$ is slightly different from the one for $F_{<:}$ (Lemma 3). First, because it is defined on indexed value sets and on the corresponding ordering relation $\sqsubseteq$ instead of plain sets and set inclusion $\subseteq$, and therefore needs to take the switch of direction for negative bounds selectors into account. Second, in $D_{<:}$ the subtyping rules for type selections $x.\text{Type}$, (Ssel1) and (Ssel2), depend on the type assignment relation, which again depends on subtyping via the subsumption rule (Tsub). Hence, we need to prove two statements in a mutual induction.

▶ **Lemma 15** (Semantic Widening).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash T_1 <: T_2}{[\![T_1]\!]_\rho \sqsubseteq [\![T_2]\!]_\rho}$$

▶ **Lemma 16** (Inversion of Variable Typing).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash x : T}{H(x) \in \rho(x)^0 \qquad \rho(x) \sqsubseteq [\![T]\!]_\rho \qquad \text{GoodBounds } \rho(x)}$$

Proof. By simultaneous induction on the subtyping and type assignment relations. Cases (Styp) and (Strans) are solved directly by the inductive hypothesis. Cases (Ssel1) and (Ssel2) are solved by a combination of the inductive hypothesis for type assignment and the resulting properties for the value set D. For case (Sfun), the case for the parameter type is solved by the inductive hypothesis. To use the inductive hypothesis for the result type, the results for the function argument and the consistent environments premise have to be extended using Lemma 10 and Lemma 13. The remaining subtyping cases (Sbot), (Stop), and (SselX), are immediate. Case (Tvar) follows from the consistent environments property. Case (Tsub) follows by induction on both type assignment and subtyping. ◀

3.5 Inversion of Function Typing

When we know that a value v is of a function type, we need to be able to extract more knowledge from this value. In particular, we need to be able to derive that the value is an actual function closure, and that, given a proper argument value, the evaluation of the function body will terminate at the correct type. After all, this is the main design of our value type relationship definition. The inversion lemmas below make this knowledge explicit.

▶ **Lemma 17** (Non-Dependent Function Inversion).

$$\frac{v \in [\![(x : T_1) \to T_2]\!]^0_\rho \qquad \text{GoodBounds } [\![T_1]\!]_\rho}{v = \langle H', \lambda x.t\rangle \qquad \forall v_x \in [\![T_1]\!]^0_\rho.\ \langle H'(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2]\!]^0_\rho}$$

Proof. The main challenge of the proof is to create a value set D, such that $D \sqsubseteq [\![T_1]\!]_\rho \wedge$ GoodBounds D, even though this D is never referred to by T_2. Thankfully we can just use the identity set (i.e. $[\![T_1]\!]_\rho$) for this case, with the help of the "good bounds" premise. Strengthening (Lemma 10) shrinks the internal context $\rho(x \mapsto [\![T_1]\!]_\rho)$ back to ρ, since x is not free in T_2. ◀

Lemma 17 deals with non-dependent function application in case (Tapp), where the resulting types do not have any free variables. We also need the next lemma, to deal with dependent function application in case (Tdapp).

▶ **Lemma 18** (Dependent Function Inversion).

$$\frac{v \in [\![(x : T_1) \to T_2^x]\!]^0_\rho \qquad \rho(z) \sqsubseteq [\![T_1]\!]_\rho \qquad \text{GoodBounds } \rho(z)}{v = \langle H', \lambda x.t\rangle \qquad \forall v_x \in \rho(z)^0.\ \langle H'(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2^z]\!]^0_\rho}$$

Proof. Here, ρ already contains a matching value set D at position z. Via substitution (Lemma 11), we can switch between names x and z as required. The rest of the proof is straightforward. ◀

Not all premises needed for Lemma 17 and Lemma 18 are directly available from the consistent environment premise in Theorem 19, but they can be obtained indirectly. Lemma 16 is used to connect consistent environments with the premises needed for dependent application, and the good bounds premise for non-dependent application follows from Lemma 14.

3.6 The Main Strong Normalization Proof

Our main strong normalization theorem states that a correctly typed term, under consistent environment, will always evaluate to a value of the same type.

▶ **Theorem 19** (Strong Normalization for D$_{<:}$). *Any well-typed term evaluates to a well-typed value:*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \sim \rho}{\langle H, t\rangle \in \mathcal{E}[\![T]\!]^0_\rho}$$

Proof. By induction on the typing derivation. Case (TTYP) is immediate. Case (TVAR) follows from the consistent environment premise. Case (TAPP) is solved by the inductive hypothesis, Lemma 17, and using the resulting evidence. The good bounds premise for Lemma 17 follows from the good bounds Lemma 14 and consistent environments. Case (TDAPP) is solved by the inductive hypothesis, Lemma 18, and Lemma 16. Both of the two application cases need extra calculations to sum up a sufficient amout of evaluation fuel k in the resulting $\mathcal{E}[\![.]\!]$ evidence. Case (TABS) uses the environment extension Lemma 13 and "stores" the inductive hypothesis inside the returned $[\![(x : T_1) \to T_2^x]\!]_\rho$ evidence, where it can be picked up by an application case later. Case (TSUB) follows from the inductive hypothesis and Lemma 15. ◀

4 Scaling up to DOT

Having proved strong normalization for D$_{<:}$, we would like to add more language features. Particular missing features from DOT are supports for records or objects with multiple members, and recursive types.

4.1 Intersection Types

DOT uses intersection types $T_1 \wedge T_2$ to model objects with multiple methods and type members, such as (Type $A = ...$) $\wedge$ (Type $B = ...$). Unfortunately, intersection types are not readily supported by our proof. To see why, consider first the usual introduction rule for intersection types

$$\frac{\Gamma \vdash t : T_1 \;,\; \Gamma \vdash t : T_2}{\Gamma \vdash t : T_1 \wedge T_2} \qquad \text{(TAND)}$$

and again the definition of $[\![.]\!]$ for type values:

$$[\![\text{Type } T_1..T_2]\!]^0_\rho \quad = \quad \{\langle H, \text{Type } \; T\rangle \mid [\![T_1]\!]_\rho \sqsubseteq [\![T_2]\!]_\rho\}$$

Since this definition does not relate T to T_1 and T_2 in any way, we may assign two types with conflicting bounds to a given value in (TAND), even though each type may have good bounds individually. Hence, the intersection of two such types will have bad bounds.

The straightforward idea would be to require in the definition of $[\![\text{Type } T_1..T_2]\!]^0_\rho$ that T is inbetween T_1 and T_2, but unfortunately, this would make $[\![.]\!]$ no longer well-founded.

It is well known that simply-typed λ-calculus with intersection types corresponds exactly to the strongly normalizing λ-terms [29]. Hence, we should be able to support them in $\mathrm{D}_{<:}$, too, without breaking strong normalization. However, additional mechanisms are needed to carry the evidence that from rule (TTYP)

$$\Gamma \vdash \text{Type } T : \text{Type } T..T \tag{TTYP}$$

only type aliases can be created, which by definition cannot have conflicting bounds.

4.2 Recursion

DOT also supports recursive functions and recursive self types. In contrast to traditional iso- or equi-recursive types, the self-reference is a term variable instead of a type variable:

$$T ::= .. \mid \mu(x : T^x)$$

Recursive Type Values May Diverge

By intention, DOT is a full Turing-complete language. But it is interesting to study the boundary between strongly normalizing and Turing-complete systems. What is the minimum required change to achieve Turing-completeness? Consistent with our expectations from traditional models of recursive types, we demonstrate that recursive *type values* are enough to encode diverging computation. If we replace the current introduction rule for type values

$$\Gamma \vdash \text{Type } T : \text{Type } T..T \tag{TTYP}$$

with a recursive one

$$\Gamma \vdash \{x \Rightarrow \text{Type } T^x\} : \mu(x : \text{Type } T^x..T^x) \tag{TTYPREC}$$

and assume standard syntactic sugar for `let` bindings, then we can write the following term:

`let` $x = \{x \Rightarrow \text{Type } (x.\text{Type } \to \bot)\}$ `in`
`let` $g = \lambda(f : x.\text{Type}).\ f\ f$ `in`
$g\ g$

This term is well-typed and diverges. Hence, we have a counterexample to strong normalization.

Recursive Self Types Don't

But surprisingly, with only non-recursive type *values* via rule (TTYP), we can still add recursive self types to the calculus and maintain strong normalization. The full DOT calculus [47] includes the following introduction and elimination rules:

```
 (* Only showing the evaluation rule for unpack terms *)
 Fixpoint eval(n: nat)(env: venv)(t: tm){struct n}: option (option vl) :=
   DO n1 ⇐ FUEL n;              (* totality: TIMEOUT if not enough fuel *)
   match t with
     ...                        (* same as in Figure 2 *)
     | tunpack ex x ey ⇒        (* unpack e_x as x in e_y *)
         DO vx ⇐ eval n1 env ex;
           eval n1 ((x,vx)::env) ey
   end.
```

Figure 4 Operational Semantics of `unpack` terms.

$$\frac{\Gamma \vdash x : T^x}{x : \mu(z : T^z) \in \Gamma} \qquad (\textsc{TvarPack})$$

$$\frac{x : \mu(z : T^z) \in \Gamma}{\Gamma \vdash x : T^x} \qquad (\textsc{TvarUnpack})$$

As well as a subtyping rule for recursive types:

$$\frac{\Gamma, x : T_1 \vdash T_1^x <: T_2^x}{\Gamma \vdash \mu(x : T_1) <: \mu(x : T_2)} \qquad (\textsc{Srec})$$

In this paper, we settle for a slightly weaker model, with an explicitly scoped `unpack` construct and the following typing rule (Tunpack) instead of (TvarUnpack) above:

$$\frac{\Gamma \vdash e_1 : \mu(z : T^z) \qquad \Gamma, x : T^x \vdash e_2 : U}{\Gamma \vdash \texttt{unpack}\ e_1\ \texttt{as}\ x\ \texttt{in}\ e_2 : U} \qquad (\textsc{Tunpack})$$

The `unpack` term is newly introduced. Its operational semantics is that of a standard `let` construct, implemented in the definitional interpreter as shown in Figure 4. We will come back to discuss difficulties in the proof with rule (TvarUnpack) in Section 4.4.

F-Bounded Quantification

Can we still do anything useful with recursive self types if the creation of proper recursive type values is prohibited? Even in this setting, recursive self types enable a certain degree of F-bounded quantification [16], as the following example shows.

Using Scala syntax, and assuming that we extend our calculus with support for records with multiple named members as in DOT, we can define a type of points with cartesian coordinates:

```
type Point = { val x: Int; val y: Int }
```

We further define a type of comparable points:

```
type CmpPoint = { val x: Int; val y: Int; def cmp(other: Point): Boolean }
```

Values of type `CmpPoint` are straightforward to create, and the comparison operation only needs to look at `x` and `y`, which are already present in type `Point`. Assuming any standard interpretation of record subtyping, `CmpPoint` is a subtype of `Point`. Hence, due to contravariance, `CmpPoint` is a subtype of `{ def cmp(o: CmpPoint): Boolean }`. In other words, `CmpPoint` values are comparable to each other, but the comparison can only treat them as `Points`—in particular, `cmp` cannot call `cmp` on another `CmpPoint`, which could potentially lead to cycles.

With recursive self types, we can abstract over types that are comparable to themselves:

```
type SelfComparable = { m =>
  type BoxedType <: { def cmp(other: m.BoxedType): Boolean }
}
```

This type is legal to define using rule (TTYP), since there is no recursive reference to `SelfComparable`, but we could not create a type value that holds a direct equivalent of `BoxedType`. However, we can create a type value that holds `CmpPoint`, and assign it type `SelfComparable` via up cast:

```
val p = { type BoxedType = CmpPoint }
val m = p: SelfComparable // up-cast
```

The definition of `BoxedType` in `SelfComparable` looks dangerously close to the diverging case shown above, and it will in fact lead to a form of self application, when a given `CmpPoint` is compared to itself. The crucial difference is that `BoxedType` is lower-bounded by type $\bot$, as opposed to being a type alias in the case above. It cannot be a precise type, because we explicitly want to widen the argument type of `cmp` from `Point` to `CmpPoint`. Due to this imprecise lower bound, we cannot assing type `m.BoxedType` to any value "from the outside".

Given this abstraction it is straightforward to define functions that operate on self-comparable data types in a generic way.

4.3 Extended Proof Method

For both intersection types and recursive self types, the required invariants rely in crucial ways on transporting properties from the creation site of type objects to their use sites – in particular the fact that only type aliases $\langle H, \text{Type } T\rangle$ can be created (with type (Type $T..T$)), and that these cannot be recursive.

This was also a key insight in the soundness proof for DOT, but it is not directly reflected in the termination proof from Section 3, which is based on tracking the GoodBounds property as part of an environment predicate.

Our revised proof method is based on the idea that we can pair each $\langle H, \text{Type } T\rangle$ value with the semantic interpretation of the type member $[\![T]\!]$. So $[\![T]\!]$ in general is no longer a set of values, but a set of $(v, [\![.]\!])$ pairs. On the first glance, this looks tricky because value sets become recursive:

$$[\![\cdot]\!] = \{(v, [\![\cdot]\!])\}$$

However we can employ a fairly straightforward indexing scheme to make this definition well-founded:

$$\begin{aligned} [\![\cdot]\!]^0 &= \{v\} \\ [\![\cdot]\!]^{n+1} &= \{(v, [\![\cdot]\!]^n)\} \end{aligned}$$

We can now define value sets as the intersection of all finite approximations:

$$[\![T]\!] = \bigcap_n [\![T]\!]^n$$

As it turns out, we no longer need the previuos `L/U` bound selectors, and the (Type $T_1..T_2$) case can ensure that the *actual* type member of an object is inbetween the given bounds. This also enables support for intersection types.

The value type relation in this model is defined as follows, where D is a value set $[\![.]\!]$ and D^n the approximation at a particular index. We write $(v, D) \in [\![T]\!]_\rho$ to mean $\forall n.\ (v, D^n) \in [\![T]\!]_\rho^{n+1}$. The environment ρ maps names to non-indexed value sets.

▶ **Definition 20** (Value Type Relation with $\wedge$ and μ).

$$
\begin{array}{lcl}
[\![T]\!]^0_\rho & = & \{v\} \\
[\![\top]\!]^{n+1}_\rho & = & \{v, D^n\} \\
[\![\bot]\!]^{n+1}_\rho & = & \{\} \\
[\![\text{Type } T_1..T_2]\!]^{n+1}_\rho & = & \{\langle H, \text{Type } \ T\rangle, D^n \mid [\![T_1]\!]^n_\rho \subseteq D^n \subseteq [\![T_2]\!]^n_\rho\} \\
[\![x.\text{Type}]\!]^{n+1}_\rho & = & \rho(x)^{n+1} \\
[\![(x : T_1) \to T_2^x]\!]^{n+1}_\rho & = & \{\langle H, \lambda x.t\rangle, D^n \mid \forall v_x, D_x.\ (v_x, D_x) \in [\![T_1]\!]_\rho \Rightarrow \\
 & & \langle H(x \mapsto v_x),\ t\rangle \in \mathcal{E}[\![T_2^x]\!]_{\rho(x \mapsto D_x)}\} \\
[\![\mu(x : T^x)]\!]^{n+1}_\rho & = & \{v, D^n \mid (v, D^n) \in [\![T^x]\!]^{n+1}_{\rho(x\mapsto D)}\} \\
[\![T_1 \wedge T_2]\!]^{n+1}_\rho & = & [\![T_1]\!]^{n+1}_\rho \cap [\![T_2]\!]^{n+1}_\rho \\
\mathcal{E}[\![T]\!]_\rho & = & \{\langle H, t\rangle \mid \exists k, v, D.\ \text{eval } k\ H\ t = \text{Done Val } v \wedge (v, D) \in [\![T]\!]_\rho\}
\end{array}
$$

Compared to Section 3, the proof structure in this model remains largely identical, with some simplifications. For example, we no longer need a "good bounds" lemma, and it also becomes more tractable to integrate the function inversion lemmas into the main proof (explicit functional inversion lemmas are no longer needed). We list the following definitions and lemmas/theorems to highlight the main differences to Section 3. The individual proofs are largely analogous.

▶ **Definition 21** (Consistent Environments Rec). A type environment Γ, a value environment H, and a value typing context ρ are consistent, written, $\Gamma \vDash H \sim \rho$, iff they contain exactly the same bindings and the following proposition holds:

$$\frac{\Gamma(x) = T}{(H(x), \rho(x)) \in [\![T]\!]_\rho}$$

▶ **Lemma 22** (Extending Consistent Environments Rec).

$$\frac{\Gamma \vDash H \sim \rho \qquad (v, D) \in [\![T]\!]_{\rho(x\mapsto D)}}{(\Gamma, x : T) \vDash (H, x : v) \sim (\rho, x : D)}$$

▶ **Lemma 23** (Semantic Widening Rec).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash T_1 <: T_2}{[\![T_1]\!]_\rho \subseteq [\![T_2]\!]_\rho}$$

▶ **Lemma 24** (Inversion of Variable Typing Rec).

$$\frac{\Gamma \vDash H \sim \rho \qquad \Gamma \vdash x : T}{(H(x), \rho(x)) \in [\![T]\!]_\rho}$$

▶ **Theorem 25** (Strong Normalization Rec). *Any well-typed term evaluates to a well-typed value:*

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vDash H \sim \rho}{\langle H, t\rangle \in \mathcal{E}[\![T]\!]_\rho}$$

4.4 Limitations on Unpacking Recursive Types

As already mentioned in Section 4.2, our current proof relies on unpacking recursive self types in explicitly scoped contexts, via rule (TUNPACK). The full DOT formalism [47, 9], however, includes an unpacking rule that is symmetric to the (TVARPACK) rule:

$$\frac{x : \mu(z : T^z) \in \Gamma}{\Gamma \vdash x : T^x} \qquad \text{(TVARUNPACK)}$$

Note that since the subtyping rules for type selections (SSEL1),(SSEL2) are defined in terms of variable type assignment $\Gamma \vdash x : (\text{Type } L..U)$, these may pack and (especially!) unpack recursive types as well.

Extending our strong normalization proof to include rule (TVARUNPACK) has proven difficult, for the following reason. The given definition of $[\![\mu(x : T^x)]\!]$ contains an implicit existential on the right hand side, which we can make explicit as follows:

$$[\![\mu(x : T^x)]\!]^{n+1}_\rho = \{v, d \mid \exists D.\ d = D^n \wedge (v, D) \in [\![T^x]\!]_{\rho(x \mapsto D)}\}$$

In the (TVARUNPACK) case of the main theorem, we have

$$\forall n.\ (H(x), \rho(x)^n) \in [\![\mu(x : T^x)]\!]^{n+1}_\rho$$

and we need to show

$$\forall n.\ (H(x), \rho(x)^n) \in [\![T^x]\!]^{n+1}_\rho.$$

Equivalently, with $H(x) = v$ and $\rho(x) = E$ we have

$$\forall n.\ \exists D.\ E^n = D^n \wedge \forall k.\ (v, D^k) \in [\![T^y]\!]^{k+1}_{\rho(y \mapsto D)}$$

and we need to show

$$\forall n.\ (v, E^n) \in [\![T^y]\!]^{n+1}_{\rho(y \mapsto E)}.$$

This is problematic, as we may have a *different* D for each n. Taking a more global view, we know that this can never actually be the case, as recursive types are only ever assigned by rule (TVARPACK), which uses the *same* $D = \rho(x)$ for each n. However, the given definition of $[\![.]\!]$ is unable to carry forward this piece of evidence, and it seems very hard to impose a corresponding constraint within the current indexed definition of $[\![.]\!]^{n+1} = \{(v, [\![.]\!]^n)\}$. Monotonicity properties such as those often used in step-indexed logical relations [4, 3] are not sufficient. Since we do not have the number of execution steps available as an input, the function case $[\![(x : T_1) \to T_2^x]\!]$ requires access to $[\![T_1]\!]$ at higher indexes than its own, and therefore precludes establishing any useful upper bound on n.

We leave support for (TVARUNPACK) in our mechanized proof as future work, along with more diverse models of recursive types, which would further increase expresssiveness, while remaining strongly normalizing. An obvious candidate among those would be, for example, an extension with strictly positive recursive type values, similar to the model that underlies inductive definitions in Coq [30].

5 Related Work

Semantic Models

There is a vast body of work on semantics and proof techniques, including Plotkin's structural operational semantics [45], Kahn's Natural Semantics [35], and Reynold's Definitional Interpreters [46].

The use of step counters in natural semantics to distinguish between divergence and errors goes back to at least Gunter and Rémy's partial proof semantics [32] and has recently been advocated in the context of compiler verification [43].

Strong Normalization

The standard proof method for strong normalization is based on logical relations and goes back to Girard and Tait [31, 54]. Strong normalization proofs for $F_{<:}$ and related calculi we presented by McAllester et al. [39] and by Ghelli [28]. Step-indexed logical relations extend the general proof method to turing complete languages. While they cannot, of course, be used to derive termination results in this case, this method can be used to show type soundness and other properties in the presence of recursive types, mutable state, and other relevant language features [12, 3, 5]. Terminating calculi that include recursion facilities have been studied for example by Stump et al. [53]. Their work on termination casts provides a type and effect system for termination. A possibly diverging term `t` can be cast to terminating type, if there is evidence for `Terminates t`, which is a primitive type form. Casinghino et al. [17] combine proofs and programs in a dependently typed language, where the logical subset is proven to be strongly normalizing via plain Girard-Tait logical relations, and step-indexed logical relations are used in the computational fragment to enable full recursion.

Subtyping and Dependent Types

Subtyping has been combined with logically consistent (and thus strongly normalizing) dependent type systems, albeit without polymorphism [13], motivated by applications in the context of logical frameworks. Pure subtype systems [34] unify not only types and terms, but also type assignment and subtyping. Being still fairly recent work, the metatheory of such pure subtype systems does not appear to be fully developed yet. In the context of intersection types, it is well known that the typable terms in simply-typed λ-calculus with intersection types are exactly the strongly normalizing λ-terms. A rather elegant proof is due to Ghilezan [29].

Recursive Self Types

System S by Fu and Stump [26] also considers a form of self-types and strong normalization. The motivation is to establish lambda encodings as a practical foundation for datatypes, i.e., enable type theories without primitive datatypes such as those in Coq and Agda. In particular, the self-type construct in System S is used to support dependent elimination with lambda encodings, including induction principles. Strong normalization was established by erasure to a version of System F_ω with positive recursive types.

Comparing System S with self types in DOT, it appears that rules (selfGen) and (selfInst) in System S are analogous to (TvarPack) and (TvarUnpack) in DOT. Our (Tunpack) rule introduces an additional `unpack` term construct, which appears less elegent. The key difference with System S seems to be that their rules deal with arbitrary terms, while the rules in DOT only deal with variables. Thus, the self-types in System S appear to be more general, but on the other hand System S has no notion of subtyping.

CDLE (the Calculus of Dependent Lambda Eliminations) [52] is a continuation of this idea and goal, and added lifting types to the calculus in order to support large eliminations. The key proven results for CDLE are type soundness and logical consistency, i.e., that no terms can inhabit contradictory types (`false`). The CDLE calculus has been implemented as a system called Cedille. Cedille is implemented in Agda, however with Agda's positivity checker turned off to allow for higher-order encodings.

Scala Foundations

Much work has been done on grounding Scala's type system in theory. Early efforts included νObj [42], Featherweight Scala [19] and Scalina [40].

None of them lead to mechanized metatheoretical results, especially soundness. DOT [8] was proposed as a simpler and more foundational core calculus, focusing on path-dependent types but disregarding classes, mixin linearization and similar questions. The original DOT formulation [8] had actual preservation issues.

The μDOT calculus [10] is the first calculus in the line with a mechanized soundness result.

Soundness for full DOT has been established more recently [47, 7], and recent work [9] has connected DOT with well-studied calculi such as $F_{<:}$ through $D_{<:}$ and related systems. The various DOT results are described in full detail in Amin's PhD thesis [6].

ML Module Systems

1ML [48] unifies the ML module and core languages through an elaboration to System F_ω based on earlier such work [49]. Compared to DOT and $D_{<:}$, the formalism treats recursive modules in a less general way and it only models fully abstract vs fully concrete types, not bounded abstract types.

In good ML tradition, 1ML supports Hindler-Milner style type inference, with only small restrictions. Path-dependent types in ML modules go back at least to SML [37], with foundational work on translucent signatures by Harper and Lillibridge [33] and Leroy [36]. MixML [22] drops the stratification requirement and enables modules as first-class values.

Related Languages

Other calculi related to DOT's path-dependent types include the family polymorphism of Ernst [23], Virtual Classes [25, 24, 41, 27], and ownership type systems like Tribe [18, 15].

Like System $D_{<:}$, pure type systems [14] unify term and type abstraction. Extensions of System $F_{<:}$ related to DOT include intersection types and bounded polymorphism [44] and higher-order subtyping [51, 1].

6 Conclusions

Following the recent type soundness proof for DOT, the present paper establishes stronger metatheoretic properties. The main result is a fully mechanized proof of strong normalization for $D_{<:}$, a variant of DOT that excludes recursive functions and recursive types. We further showed that certain variants of DOT's recursive self types can be integrated successfully while keeping the calculus strongly normalizing. This result is surprising, as traditional recursive types are known to make a language Turing-complete.

Acknowledgements. The authors thank Nada Amin, Martin Odersky, Sandro Stucki, and the anonymous ECOOP reviewers.

References

1 Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 18:797–822, 10 2008.

2 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP*, 2003.

3 Amal J. Ahmed. *Semantics of types for mutable state.* PhD thesis, Princeton University, 2004.

4 Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

5 Amal J. Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *ESOP*, 2006.

6 Nada Amin. *Dependent Object Types.* PhD thesis, EPFL, August 2016.

7 Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. The essence of dependent object types. In *A List of Successes That Can Change the World*, volume 9600 of *Lecture Notes in Computer Science*, pages 249–272. Springer, 2016.

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

9 Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *POPL*, 2017.

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

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

12 Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

13 David Aspinall and Adriana Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1):273–309, 2001.

14 H. P. Barendregt. Handbook of logic in computer science. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types. Oxford University Press, 1992.

15 Nicholas R. Cameron, James Noble, and Tobias Wrigstad. Tribal ownership. In *OOPSLA*, 2010.

16 Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280. ACM, 1989.

17 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, pages 33–46. ACM, 2014.

18 Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: a simple virtual class calculus. In *AOSD*, 2007.

19 Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *MFCS*, 2006.

20 Olivier Danvy and Jacob Johannsen. Inter-deriving semantic artifacts for object-oriented programming. *J. Comput. Syst. Sci.*, 76(5):302–323, 2010.

21 Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. On inter-deriving small-step and big-step semantics: A case study for storeless call-by-need evaluation. *Theor. Comput. Sci.*, 435:21–42, 2012.

22 Derek Dreyer and Andreas Rossberg. Mixin' up the ML module system. In *ICFP*, 2008.

23 Erik Ernst. Family polymorphism. In *ECOOP*, 2001.

24 Erik Ernst. Higher-order hierarchies. In *ECOOP*, 2003.

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

26 Peng Fu and Aaron Stump. Self types for dependently typed lambda encodings. In *RTA-TLCA*, volume 8560 of *Lecture Notes in Computer Science*, pages 224–239. Springer, 2014.

27 Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Dependent classes. In *OOPSLA*, 2007.

28 Giorgio Ghelli. Termination of system f-bounded: A complete proof. *Inf. Comput.*, 139(1):39–56, 1997.

29 Silvia Ghilezan. Strong normalization and typability with intersection types. *Notre Dame Journal of Formal Logic*, 37(1):44–52, 1996.

30 Eduardo Giménez. Structural recursive definitions in type theory. In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408. Springer, 1998.

31 Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

32 C. A. Gunter and D. Rémy. A proof-theoretic assesment of runtime type errors. Technical Report 11261-921230-43TM, AT&T Bell Laboratories, 1993.

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

34 DeLesley S. Hutchins. Pure subtype systems. In *POPL*, 2010.

35 Gilles Kahn. Natural semantics. In *STACS*, 1987.

36 Xavier Leroy. Manifest types, modules and separate compilation. In *POPL*, 1994.

37 David Macqueen. Using dependent types to express modular structure. In *POPL*, 1986.

38 Dmitry Petrashko Martin Odersky, Guillaume Martres. Implementing higher-kinded types in dotty. In *Scala*, 2016.

39 David A. McAllester, J. Kucan, and D. F. Otth. A proof of strong normalization of F_2, F_ω and beyond. *Inf. Comput.*, 121(2):193–200, 1995.

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

41 Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA*, 2004.

42 Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP*, 2003.

43 Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. In *ESOP*, 2016.

44 Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism.* PhD thesis, Carnegie Mellon University, December 1991.

45 Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

46 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

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

48 Andreas Rossberg. 1ML - core and modules united (F-ing first-class modules). In *ICFP*, 2015.

49 Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014.

50 Jeremy Siek. Type safety in three easy lemmas. `http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html`, 2013.

51 Martin Steffen. *Polarized higher-order subtyping.* PhD thesis, University of Erlangen-Nuremberg, 1997.

52 Aaron Stump. The calculus of dependent lambda eliminations. Technical report, The University of Iowa (under submission to JFP), 2016. `http://homepage.cs.uiowa.edu/~astump/papers/cedille-draft.pdf`.

53 Aaron Stump, Vilhelm Sjöberg, and Stephanie Weirich. Termination casts: A flexible approach to termination with general recursion. In *PAR*, volume 43 of *EPTCS*, pages 76–93, 2010.

54 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967.

A Mechanization in Coq

We outline the correspondence between the formalism on paper and its implementation in Coq (`https://github.com/tiarkrompf/minidot/tree/master/ecoop17`).

The Coq package contains the following source files:

- `dsubsup_total.v` – Strong normalization proof for $D_{<:}$, closely matches the presentation in Section 3
- `dsubsup_total_rec.v` – Strong normalization proof for $D_{<:}$ with recursive self types and intersection, Section 4

A.1 Model

A.1.1 Syntax (Figure 2)

`ty`	S, T, U ::=	Type
`TTop`	$\top$	top type
`TBot`	$\bot$	bottom type
`TMem` $S\ U$	Type : $S..U$	type member
`TAll` $S\ U$	$(x : S) : U^x$	(dependent) function type
`TSel` X	x.Type	type selection
`TBind` T	$\{z \Rightarrow T^z\}$	recursive self type
`TAnd` $T\ T$	$T \wedge T$	intersection type
`tm`	t, u ::=	Term
`tvar` x	x	variable reference
`ttyp` T	Type T	type value
`tabs` $T\ t$	$\lambda x : T.t$	function abstraction
`tapp` $t\ t$	$t\ t$	function invocation

For representing variabe names in relation to an environment, we use a reverse de Bruijin convention, so that the name is invariant under environment extension. An environment is a list of right-hand sides (types, values, ...). The older the binding, the more to the right, the smaller its number. The name is uniquely determined by the position in the list as the length of the tail (see indexr function in the artifact).

In addition, for types, we use a locally-nameless de Bruijin convention for variables under dependent types so that it's easy to substitute binders without variable capture. A variable x bound in T^x by a dependent function type $(x : S) \to T^x$ (or type abstraction for $D_{<:}$) is represented by (TVarB i) where i is the de Brujin level, i.e., the number of other binders in scope in between a bound variable occurrence and its binder.

A.1.2 Type System Judgements

`stp` $\Gamma\ S\ U$	$\Gamma \vdash S <: U$	Subtyping
`has_type` $\Gamma\ t\ T$	$\Gamma \vdash t : T$	Typing
`val_type` $H\ v\ T$	$H \vdash v : T$	Runtime Value Typing

As we mention in Section 3, we omit routine well-formedness checks from the rules on paper for readability. In Coq, these correspond to *closed* conditions, which ensure that all the variables in a type are well-bound for the given environment and binding structure. The relation *closed* $k\ |j|\ |H|\ T$ ensures that T is well-bound in a context H, abstract environment J and under at most $\leq k$ binders.

A.2 Strong Normalization Proofs for Plain $D_{<:}$ (Section 3)

A.2.1 Figures and Definitions

- (Figure 2, System $D_{<:}$) — file dsubsup_total.v (tm, ty, stp)
- (Definition 5, Indexed Value Sets) — file dsubsup_total.v (bound, sel)
- (Definition 6, Polarity of Bound Selectors) — file dsubsup_total.v (pos)
- (Definition 7, Indexed Value Set Inclusion) — file dsubsup_total.v (vtsub)
- (Definition 8, Good bounds) — file dsubsup_total.v (good_bounds)
- (Definition 9, Value Type Relation) — file dsubsup_total.v (val_type)
- (Definition 12, Consistent Environments) — file dsubsup_total.v (R_env)

A.2.2 Lemmas

- (Lemma 10, Weakening/Strengthening) corresponds to `Lemma valtp_extend(H)` and `Lemma valtp_shrink(M,H)`.
- (Lemma 11, Substitution) corresponds to `Lemma vtp_subst(1,2,3)`.
- (Lemma 13, Extending Consistent Environments) corresponds to `Lemma wf_env_extend(0)`.
- (Lemma 14, Good Bounds) corresponds to `Lemma valtp_bounds`.
- (Lemma 15, Semantic Widening) corresponds to `Lemma valtp_widen`.
- (Lemma 16, Inversion of Variable Typing) corresponds to `Lemma invert_var`.
- (Lemma 17, Non-Dependent Function Inversion) corresponds to `Lemma invert_abs`.
- (Lemma 18, Dependent Function Inversion) corresponds to `Lemma invert_dabs`.

A.2.3 Theorems

- (Theorem 19, Strong Normalization for $D_{<:}$) corresponds to `Theorem full_total_safety`.

A.3 Intersection and Recursive Types (Section 4)

The core lemmas and definitions are analogous to the ones in Section 3 as shown above. The definition of value sets as the intersection of all finite approximations

$$[\![T]\!] = \bigcap_n [\![T]\!]^n$$

translates to Coq as follows, extending our definition of value sets as characteristic functions (`vl -> Prop`) to accommodate the indexing scheme. We use universal quantification ($\forall n$) to represent unbounded intersection:

```
Fixpoint vset n :=
  match n with
    | 0 => vl -> Prop
    | S n => vl -> vset n -> Prop
end.
Definition vseta := forall n, vset n.
```

Note that `val_type n` in the Coq file corresponds to $[\![.]\!]^{n+1}$ in the text. Lemma `valtp_to_vseta` adjusts the index back.