Syntactically Restricting Bounded Polymorphism for Decidable Subtyping

Julian Mackay[1 [0000-0003-3098-3901]], Alex Potanin[1 [0000-0002-4242-2725]], Jonathan Aldrich[2 [0000-0003-0631-5591]], and Lindsay Groves[1 [0000-0002-9179-3602]]

[1] Victoria University of Wellington, Wellington, New Zealand
`{julian, alex, lindsay}@ecs.vuw.ac.nz`
[2] Carnegie Mellon University, Pittsburgh, USA
`jonathan.aldrich@cs.cmu.edu`

Abstract. Subtyping of Bounded Polymorphism has long been known to be undecidable when coupled with contra-variance. While decidable forms of bounded polymorphism exist, they all sacrifice either useful properties such as contra-variance (Kernel $F_{<:}$), or useful metatheoretic properties ($F^{\top}_{<:}$). In this paper we show how, by syntactically separating contra-variance from the recursive aspects of subtyping in System $F_{<:}$, decidable subtyping can be ensured while also allowing for both contra-variant subtyping of certain instances of bounded polymorphism, and many of System $F_{<:}$'s desirable metatheoretic properties. We then show that this approach can be applied to the related polymorphism present in $D_{<:}$, a minimal calculus that models core features of the Scala type system.

Keywords: Polymorphism · Language Design · Functional Languages · Object Oriented Languages

1 Introduction

Bounded polymorphism (or bounded quantification) is a powerful and widely used language construct that introduces a form of abstraction for types. Where functions provide an abstraction of behaviour for values, bounded polymorphism provides an abstraction of behaviour for types. A motivating example is an ordering for numbers, comparing two numbers, and returning −1 if the first is smaller than the second, 0 if the two numbers are equal, and 1 if the first is larger than the second. Below we provide such the signature for `ord` using no particular language syntax.

```
def ord : [A <: Number] A -> A -> Integer
```

The type `A` is unimportant except in that A is some subtype of `Number` (the upper bound on `A`). Ideally we would like `ord` to be defined abstractly for any value that could be considered a `Number`, and not have to write a separate function for

`Integer`, `Natural`, and `Real`. `ord` is quantified over type `A` which is bounded by `Number`.

Bounded polymorphism has been adopted by many different languages, and is not exclusive to any specific paradigm. Haskell is an instance of bounded polymorphism in a functional setting. In Haskell, bounds take the form of type classes that values must conform to [17]. In an object oriented language, Java Generics provide a form of bounded polymorphism for both method and class definitions. Scala exists in both the function and object oriented paradigms, and includes generics similar to that of Java (only more flexible), but adds abstract type members on top, further complicating matters.

Unfortunately, several forms of bounded polymorphism have been shown to exhibit undecidable subtyping. To the surprise of many at the time, Pierce [13] demonstrated that subtyping in System $F_{<:}$, a typed λ-calculus with subtyping and bounded polymorphism, was undecidable by a reduction to the halting problem. More recently, and to perhaps less surprise, subtyping of Java Generics was also shown to be undecidable [8]. Hu and Lhoták [9] showed subtyping of $D_{<:}$, a minimal calculus, capturing parts of the Scala type system, was undecidable by a reduction to an undecidable fragment of System $F_{<:}$. Mackay et al. [10] developed two decidable variants on Wyvern, a programming language closely related to Scala. Mackay et al. focused on recursive types in Scala, but touched on bounded polymorphism.

If subtyping in languages with relatively wide usage is undecidable, then one might ask the question: how important is decidable subtyping in practice? Unfortunately, undecidability means that type checking of certain programs will not terminate, and will potentially crash without any error message indicating the problem. In writing a compiler, one fix to this problem might be to enforce a maximal depth on proof search, or to simply timeout during type checking. These are unsatisfying solutions, as not only might they create some false negatives, but they also won't be able to provide the programmer much guidance on debugging their program. Thus, while presumably rare, the potential problems are severe.

Not all forms of bounded polymorphism are undecidable, and there have been attempts at identifying fragments of bounded polymorphism that are both decidable and expressive. With regard to System $F_{<:}$, the most notable instances of these are perhaps Kernel $F_{<:}$ and $F_{<:}^{\top}$ (technically Kernel $F_{<:}$ existed prior to questions of decidability). All restrictions sacrifice some aspect of the language, and exclude some category of program from the language. Both Kernel $F_{<:}$ and $F_{<:}^{\top}$ exclude useful behaviour, or in the case of $F_{<:}^{\top}$ introduce undesirable properties to the language (this will be addressed in Section 2).

In this paper we show how simple syntactic restrictions can allow for decidable forms of bounded polymorphism that are easy to type check, allow for informative error messages, all while retaining many of the useful properties of typing in System $F_{<:}$: subtype transitivity, type safety, and minimal typing. We then demonstrate that this approach can be extended to the related calculus $D_{<:}$. The novelty of our approach lies in its simplicity. Simple syntactic restrictions allow for relatively simple extensions to existing type checkers, and can help keep metatheory simple. Simplicity of metatheory is particularly useful in the context of D$_{<:}$, a type system that arises from a family of type systems that are notoriously nuanced in their theoretical foundations [15].

2 The Undecidability of Bounded Polymorphism in System F$_{<:}$

Bounded polymorphism was formalized in System F$_{<:}$ by Cardelli [4], and shown to be undecidable by Pierce [13]. System F$_{<:}$ introduces bounded polymorphism to the simply typed λ-calculus by way of a universally quantified syntactic form with the following typing rule.

$$\frac{\Gamma, (\alpha \leqslant \tau_1) \vdash t \ : \ \tau_2}{\Gamma \vdash \Lambda(\alpha \leqslant \tau_1).t \ : \ \forall(\alpha \leqslant \tau_1).\tau_2}$$

That is, term t, with type τ_2, is quantified over some type, represented by α, whose upper bound is τ_1. The undecidability of subtyping in System F$_{<:}$ was demonstrated by a reduction of subtyping to the halting problem. The reduction relies on the contra-variance in the subtyping rule for bounded polymorphism given below.

$$\frac{\Gamma \vdash \tau_2 \ <: \ \tau_1 \qquad \Gamma, (\alpha \leqslant \tau_2) \vdash \tau_1' \ <: \ \tau_2'}{\Gamma \vdash \forall(\alpha \leqslant \tau_1).\tau_1' \ <: \ \forall(\alpha \leqslant \tau_2).\tau_2'} \quad (\textsc{S-All})$$

As can be seen above, subtyping of bounded polymorphism in System F$_{<:}$ allows for contra-variance on the polymorphic type bound. Kernel F$_{<:}$, a variant of System F$_{<:}$, has been shown to be decidable in its subtyping [14]. Kernel F$_{<:}$ removes the contra-variance of the S-All rule above, and instead enforces invariance on the bound.

$$\frac{\Gamma, (\alpha \leqslant \tau) \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash \forall(\alpha \leqslant \tau).\tau_1 \ <: \ \forall(\alpha \leqslant \tau).\tau_2} \quad (\textsc{S-All-Kernel})$$

While decidable, S-All-Kernel is unsatisfying as it excludes desirable behaviour. Ideally, we would like the `ord` function, from Section 1, to be usable in positions that require a more specific type such as `Integer`. Suppose we want to parameterize an `Integer` sorting algorithm on not just the list, but the ordering too.

```
def sort (compare : [A <: Integer] A -> A -> Integer,
          l : List[Integer]) : List[Integer]
```

We would like to be able to call the above `sort` function with `ord`.

```
assert(sort(ord, [1, 8, 2, -10]) == [-10, 1, 2, 8])
```

Castagna and Pierce [5] attempted to introduce such variance in a safe way by proposing F$^{\top}_{<:}$ with the following subtyping rule for bounded polymorphism.

$$\frac{\Gamma \vdash \tau_2 \ <: \ \tau_1 \qquad \Gamma, (\alpha \leqslant \top) \vdash \tau_1' \ <: \ \tau_2'}{\Gamma \vdash \forall(\alpha \leqslant \tau_1).\tau_1' \ <: \ \forall(\alpha \leqslant \tau_2).\tau_2'} \quad (\textsc{S-All}^{\top})$$

$$
\begin{array}{llll}
\tau ::= & & & \\
\top & \textit{top} & \tau \to \tau & \textit{arrow} \\
\alpha & \textit{variable} & \forall(\alpha \leqslant \tau).\tau & \textit{all}
\end{array}
$$

Fig. 1. System $\mathrm{F}_{<:}$ Type Syntax

Unfortunately, while decidable, $\mathrm{F}^{\top}_{<:}$ sacrifices minimal typing. That is, it is possible to write a term in $\mathrm{F}^{\top}_{<:}$ that can be typed with two different, and unrelated types [6]. A lack of minimal typing means that the typing algorithm for $\mathrm{F}^{\top}_{<:}$ is not complete.

3 Separating Recursion and Contra-variance in System $\mathbf{F}_{<:}$

In this section we present a variant of System $\mathrm{F}_{<:}$ that introduces a syntactic restriction on bounded polymorphism to achieve decidable subtyping. We start by introducing the type syntax of System $\mathrm{F}_{<:}$ in Figure 1. Since we are only concerned with subtyping, and not typing, we only present the type syntax. The term syntax and typing rules can be found in the accompanying technical report. Further, throughout the rest of this paper, we refer to several different definitions of subtyping and typing. To distinguish between these differences, we annotate the the judgment. We have already mentioned three different subtyping definitions, and differentiate them here

- Subtyping for System $\mathrm{F}_{<:}$ as defined by Cardelli et al. [4] is indicated as $\Gamma \vdash \tau_1 \;<:\; \tau_2$.
- Subtyping for Kernel $\mathrm{F}_{<:}$ is indicated as $\Gamma \vdash \tau_1 \;<:^{K}\; \tau_2$.
- Subtyping for $\mathrm{F}^{\top}_{<:}$ is indicated as $\Gamma \vdash \tau_1 \;<:^{\top}\; \tau_2$.

A type in System $\mathrm{F}_{<:}$ is either the top type ($\top$), a bounded type variable (α), an arrow type ($\tau \to \tau$), or a universally quantified type ($\forall(\alpha \leqslant \tau).\tau$) i.e. bounded polymorphism. Note: in the literature, polymorphism can mean several different language features, however in this paper, unless stated otherwise, we use it as short hand to refer to bounded polymorphism of the form in System $\mathrm{F}_{<:}$.

In Figure 2 we define the subtyping of $\mathrm{F}^{N}_{<:}$, a normal form of subtyping in System $\mathrm{F}_{<:}$, defined by Pierce [13]. Subtyping is bounded above by $\top$ (S^{N}-Top) and explicitly reflexive in the case of type variables (S^{N}-Rfl). A type supertypes a type variable if it supertypes its upper bound (S^{N}-Var). Subtyping of arrow types is contra-variant with respect to its argument type, and covariant with respect to its return type (S^{N}-Arr). Finally, subtyping of bounded polymorphism is contra-variant with respect to the type bounds, and covariant with respect to the type bodies.

Achieving a decidable variant of System $\mathrm{F}_{<:}$ follows a simple idea: we restrict contra-variance of type bounds to only types that do not themselves contain

$$\Gamma \vdash \tau \ <:^{N} \ \top \quad (\mathrm{S}^{N}\text{-}\textsc{Top}) \qquad\qquad \Gamma \vdash \alpha \ <:^{N} \ \alpha \quad (\mathrm{S}^{N}\text{-}\textsc{Rfl})$$

$$\frac{\begin{array}{c}(\alpha \ \leqslant \ \tau') \ \in \ \Gamma \\ \Gamma \vdash \tau' \ <:^{N} \ \tau\end{array}}{\Gamma \vdash \alpha \ <:^{N} \ \tau} \ (\mathrm{S}^{N}\text{-}\textsc{Var}) \qquad\qquad \frac{\begin{array}{c}\Gamma \vdash \tau_2 \ <:^{N} \ \tau_1 \\ \Gamma \vdash \tau_1' \ <:^{N} \ \tau_2'\end{array}}{\Gamma \vdash \tau_1 \to \tau_1' \ <:^{N} \ \tau_2 \to \tau_2'} \ (\mathrm{S}^{N}\text{-}\textsc{Arr})$$

$$\frac{\Gamma \vdash \tau_2 \ <:^{N} \ \tau_1 \qquad \Gamma, (\alpha \leqslant \tau_2) \vdash \tau_1' \ <:^{N} \ \tau_2'}{\Gamma \vdash \forall(\alpha \leqslant \tau_1).\tau_1' \ <:^{N} \ \forall(\alpha \leqslant \tau_2).\tau_2'} \ (\mathrm{S}^{N}\text{-}\textsc{All})$$

Fig. 2. System $\mathrm{F}_{<:}$ Subtyping

$\tau ::=$	$\mathbf{F}^{R}_{<:}$ **Type**	$\alpha ::=$	**Type Variable**
$\top$	*top*	υ	*unrestricted*
α	*variable*	γ	*restricted*
$\tau \to \tau$	*arrow*	$\rho ::=$	**Restricted Type**
$\forall(\gamma \leqslant \rho).\tau$	*restricted all*	$\top$	*top*
$\forall(\upsilon \leqslant \tau).\tau$	*all*	γ	*variable*
		$\rho \to \rho$	*arrow*

Fig. 3. $\mathrm{F}^{R}_{<:}$ Type Syntax

bounded polymorphism. In Figure 3 we introduce a separated variant for the syntax of System $\mathrm{F}_{<:}$ called $\mathrm{F}^{R}_{<:}$. In $\mathrm{F}^{R}_{<:}$, types containing no bounded polymorphism are identified by ρ. Their only difference from more general types is a lack of bounded polymorphism. A restricted type, ρ, is either $\top$, a restricted type variable γ, or an arrow type. We keep the generalized form of type variables, α, for convenience. We now define a restricted subtyping relation using the rule set in Figure 4

The subtyping of $\mathrm{F}^{R}_{<:}$ defined in Figure 4 replaces the S^{N}-All rule with two rules: S^{R}-All-Kernel and S^{R}-All. S^{R}-All-Kernel is exactly the rule for subtyping of bounded polymorphism found in Kernel $\mathrm{F}_{<:}$, that is, for $\forall(\alpha \leqslant \tau_1).\tau_1'$ to subtype $\forall(\alpha \leqslant \tau_2).\tau_2'$, τ_1 and τ_2 must be syntactically equivalent. Contra-variance is allowed only in cases where the type bounds are of the form $\gamma \leqslant \rho$, and thus do not themselves include bounded polymorphism. This is captured by the rule S^{R}-All.

The result of this restriction is that subtyping may only introduce new instances of bounded polymorphism into the context if they are common to both types.

3.1 Subtype Decidability

In order to prove subtype decidability, we define a finite measure on types under a context ($\mathcal{M}(\Gamma, \tau)$), along with an ordering ($\mathcal{M}(\Gamma_1, \tau_1) < \mathcal{M}(\Gamma_2, \tau_2)$). We

$$\Gamma \vdash \tau \;<:^{R}\; \top \quad (\text{S}^{R}\text{-}\textsc{Top}) \qquad\qquad \Gamma \vdash \alpha \;<:^{R}\; \alpha \quad (\text{S}^{R}\text{-}\textsc{Rfl})$$

$$\frac{\begin{array}{c}(\alpha \leqslant \tau') \in \Gamma \\ \Gamma \vdash \tau' \;<:^{R} \tau\end{array}}{\Gamma \vdash \alpha \;<:^{R}\; \tau} \;(\text{S}^{R}\text{-}\textsc{Var}) \qquad\qquad \frac{\begin{array}{c}\Gamma \vdash \tau_2 \;<:^{R}\; \tau_1 \\ \Gamma \vdash \tau_1' \;<^{R}:\; \tau_2'\end{array}}{\Gamma \vdash \tau_1 \to \tau_1' \;<:^{R}\; \tau_2 \to \tau_2'} \;(\text{S}^{R}\text{-}\textsc{Arr})$$

$$\frac{\Gamma, (\alpha \leqslant \tau) \vdash \tau_1 \;<:^{R}\; \tau_2}{\Gamma \vdash \forall(\alpha \leqslant \tau).\tau_1 \;<:^{R}\; \forall(\alpha \leqslant \tau).\tau_2} \;(\text{S}^{R}\text{-}\textsc{All-Kernel})$$

$$\frac{\Gamma \vdash \rho_2 \;<:^{R}\; \rho_1 \qquad \Gamma, (\gamma \leqslant \rho_2) \vdash \tau_1 \;<:^{R}\; \tau_2}{\Gamma \vdash \forall(\gamma \leqslant \rho_1).\tau_1 \;<:^{R}\; \forall(\gamma \leqslant \rho_2).\tau_2} \;(\text{S}^{R}\text{-}\textsc{All})$$

Fig. 4. $\text{F}^{R}_{<:}$ Subtyping

subsequently demonstrate that for any calls to a subtype algorithm for $\text{F}^{R}_{<:}$, all resulting subtype calls are strictly smaller that the original call.

Indexed Types Before we define our measure $\mathcal{M}$, we introduce an indexing on type variables and types, along with a related invariant on typing contexts.

We index type variables with a natural number, indicating their position in a context. This is represented as a superscript on type variables: α^n under context Γ is the $(n+1)$th type variable introduced to Γ (the first type variable introduced to Γ being indexed by 0). We extend this indexing to types in the form of an upper bound on type variable indices: τ^n under context Γ indicates that for all α^i occurring in τ^n, $i < n$. Generally the index n is not important, and so we only include it when relevant. We further define a simple form of well-formedness:

Definition 1 (Type Variable Well-Formedness). *A type τ^n is well-formed under context Γ (written $\Gamma \vdash \tau^n$ wf) if and only if $n \leq |\Gamma|$*

We now use this to define a well-formedness property that we assume on all typing contexts:

Definition 2 (Typing Context Well-Formedness). *A typing context Γ is well-formed (written Γ wf) if and only if for all $(\alpha^n \leqslant \tau^i) \in \Gamma$ we have $i < n$.*

That is, a type bound τ in a typing context Γ may only contain occurrences of type variables that were already in Γ when τ was added to it.

Note that indices on types are not unique, and are only an upper bound on type variable occurrences. i.e. if we are able to write τ^n, and $n < m$, then we are equally able to write τ^m. Finally, we use this to define an indexing on typing contexts.

Definition 3 (Indexed Typing Context).

$$\Gamma^n \triangleq \{(\alpha^i \leqslant \tau) | (\alpha^i \leqslant \tau) \in \Gamma \text{ and } i \leq n\}$$

$$\begin{array}{llll}
\mathcal{D}(\Gamma, \alpha) & = 1 + \mathcal{D}(\Gamma', \tau) & \mathcal{D}(\Gamma, \top) & = 0 \\
& \quad \textit{where } \Gamma = \Gamma', (\alpha \leqslant \tau), \Gamma'' & \mathcal{D}(\Gamma, \forall(\alpha \leqslant \tau_1).\tau_2) & = 0 \\
\mathcal{D}(\Gamma, \tau_1 \rightarrow \tau_2) & = 1 + \mathit{max}(\mathcal{D}(\Gamma, \tau_1), \mathcal{D}(\Gamma, \tau_2)) & &
\end{array}$$

Fig. 5. Quantification Depth: the depth of the next instance of bounded polymorphism.

$$\begin{array}{llll}
\mathcal{Q}(\top) & = 0 & \mathcal{Q}(\emptyset) & = \emptyset \\
\mathcal{Q}(\alpha) & = 0 & \mathcal{Q}(\Gamma) & = \mathcal{Q}(\tau) + \mathcal{Q}(\Gamma') \\
\mathcal{Q}(\tau_1 \rightarrow \tau_1) & = \mathcal{Q}(\tau_1) + \mathcal{Q}(\tau_2) & & \quad \textit{where } \Gamma = \Gamma', (\alpha \leqslant \tau) \\
\mathcal{Q}(\forall(\alpha \leqslant \tau_1).\tau_2) & = 1 + \mathcal{Q}(\tau_1) + \mathcal{Q}(\tau_2) & \mathcal{Q}(\Gamma, \tau^n) & = \mathcal{Q}(\Gamma^n) + \mathcal{Q}(\tau^n)
\end{array}$$

Fig. 6. Quantification Size: the number of instances of bounded polymorphism in a type.

A Finite Measure on Types $\mathcal{M}(\Gamma, \tau)$ is defined as a lexicographic ordering on the quantification size and the quantification depth of τ under Γ. Note: we use quantification here to refer to bounded polymorphism, i.e. "*all*" types of the form $\forall(\alpha \leqslant \tau_1).\tau_2$. We define $\mathcal{M}$ using two simpler measures:

1. $\mathcal{D}(\Gamma, \tau)$ (see Figure 5): the depth at which the the next instance of bounded polymorphism occurs in τ, and
2. $\mathcal{Q}(\Gamma, \tau)$ (see Figure 6): the number of instances of bounded polymorphism in τ under context Γ.

$\mathcal{D}$, or quantification depth is defined in Figure 5 as the maximum depth at which the next quantification type occurs. $\mathcal{D}$ is also necessarily finite, since it is bounded by the sizes the context Γ and type τ. Note: we assume a simple well-formedness property, that type variables in the context only refer to types lower down in the context, this allows us to disregard Γ'' in the definition of $\mathcal{D}(\Gamma, \alpha)$.

$\mathcal{Q}(\tau)$, or quantification size of a type is defined in Figure 6 as the number of syntactic instances of quantification within some τ. It is simple to demonstrate that $\mathcal{Q}$ is finite, as it is bound by the (finite) size of τ. We then define $\mathcal{Q}(\Gamma, \tau)$, as the quantification size of both the type τ, and all types in the context Γ. Since context arising from type checking must be finite, it follows that $\mathcal{Q}(\Gamma, \tau)$ must also be finite.

Finally, we define $\mathcal{M}(\Gamma, \tau)$ along with an ordering in Figure 7. $\mathcal{M}(\Gamma, \tau)$ is defined as $(\mathcal{Q}(\Gamma, \tau), \mathcal{D}(\Gamma, \tau))$. The key property of $\mathcal{M}$ that guarantees subtype decidability, is the fact that restricted types have no bounded polymorphism as subterms, i.e.

Property 1 (Quantification Size of Restricted Types in $F^{R}_{<:}$).

$$\forall \rho, \mathcal{Q}(\rho) = 0$$

Proof of Decidability Since the subtyping defined in Figure 4 is syntax-directed, the inversion of the rules themselves represent an algorithm for sub-

$$\begin{array}{lcll} \mathcal{M} & = & \mathcal{Q} \times \mathcal{D} & \\ \textit{and} & & & \\ (q_1, d_1) < (q_2, d_2) & \iff & q_1 < q_2 \textit{ or} & (1) \\ & & q_1 = q_2 \textit{ and } d_1 < d_2 & (2) \end{array}$$

Fig. 7. Lexicographic ordering on quantification size and depth.

typing of $\mathrm{F}^R_{<:}$. This means that we need not define an algorithm, and are only required to reason about the conclusions of the rules and their premises. We define $\texttt{subtype}_{F^R_{<:}}$ as the algorithm obtained by inverting the rules in Figure 4. Theorem 1 provides a proof of decidability of subtyping in $\mathrm{F}^R_{<:}$.

Theorem 1 (Subtype Decidability of $\mathbf{F}^R_{<:}$). *For all Γ, τ_1, and τ_2, $\texttt{subtype}_{F^R_{<:}}(\Gamma, \tau_1, \tau_2)$ is guaranteed to terminate.*

Proof. Termination of $\texttt{subtype}_{F^R_{<:}}$ is easy to demonstrate by showing that $\mathcal{M}$ represents a strictly decreasing measure on subtyping. That is, for any subtype check

$$\texttt{subtype}_{F^R_{<:}}(\Gamma, \tau_1, \tau_2)$$

for any resulting calls

$$\texttt{subtype}_{F^R_{<:}}(\Gamma', \tau_1', \tau_2')$$

we have

$$\mathcal{M}(\Gamma', \tau_1') + \mathcal{M}(\Gamma', \tau_2') < \mathcal{M}(\Gamma, \tau_1) + \mathcal{M}(\Gamma, \tau_2)$$

Since $\texttt{subtype}_{F^R_{<:}}$ is defined as the inversion of the rules in Figure 4, the above property is demonstrated by showing that the size of the premises (as measured by $\mathcal{M}$) of each rule is strictly smaller than the size of the conclusion. In most cases it is fairly simple to demonstrate this invariant, however in the cases of S^R-Var and S^R-All, the result is not necessarily so obvious.

*Case 1 (*S^R-Var*).*

$$\frac{(\alpha \leqslant \tau') \in \Gamma \qquad \Gamma \vdash \tau' <:^R \tau}{\Gamma \vdash \alpha <:^R \tau} \quad (\text{S}^R\text{-VAR})$$

The only sub-proof that we need demonstrate our invariant for is $\Gamma \vdash \tau' <:^R \tau$. That is, we need to show that

$$(\mathcal{Q}(\Gamma, \tau') + \mathcal{Q}(\Gamma, \tau), \mathcal{D}(\Gamma, \tau') + \mathcal{D}(\Gamma, \tau)) < (\mathcal{Q}(\Gamma, \alpha) + \mathcal{Q}(\Gamma, \tau), \mathcal{D}(\Gamma, \alpha) + \mathcal{D}(\Gamma, \tau))$$

Since $\mathcal{Q}(\Gamma, \tau)$ and $\mathcal{D}(\Gamma, \tau)$ fall on both sides of the ordering, it is sufficient to show that

$$(\mathcal{Q}(\Gamma, \tau'), \mathcal{D}(\Gamma, \tau')) < (\mathcal{Q}(\Gamma, \alpha), \mathcal{D}(\Gamma, \alpha))$$

Γ is in fact an ordered list of type variable bounds, and thus $(\alpha \leqslant \tau') \in \Gamma$ is equivalent to asserting that there exists some Γ' and Γ'' such that $\Gamma = \Gamma', (\alpha \leqslant \tau'), \Gamma''$. Now from the definition of $\mathcal{D}$ we have that

$$\mathcal{D}(\Gamma, \alpha) = 1 + \mathcal{D}(\Gamma', \tau')$$

Therefore, clearly $\mathcal{D}(\mathit{\Gamma}, \alpha) > \mathcal{D}(\mathit{\Gamma}', \tau')$, and since $\mathit{\Gamma}$ (and thus $\mathit{\Gamma}'$) is ordered all variables in τ' are mapped within $\mathit{\Gamma}'$, and $\mathcal{D}(\mathit{\Gamma}, \tau') = \mathcal{D}(\mathit{\Gamma}', \tau')$, giving us

$$\mathcal{D}(\mathit{\Gamma}, \alpha) > \mathcal{D}(\mathit{\Gamma}, \tau')$$

Since $\mathcal{D}$ is decreasing, in order to show that our invariant is obeyed, we need only show that $\mathcal{Q}$ is not increasing, i.e. $\mathcal{Q}(\mathit{\Gamma}, \alpha) \not< \mathcal{Q}(\mathit{\Gamma}, \tau')$. We make use of the well-formedness of typing contexts that we defined in Definition 2. That is, whenever we retrieve a type bound from a well-formed typing context, the index of that type bound is strictly smaller than that of the associated type variable. Suppose that α above is indexed by some n By definition

$$\mathcal{Q}(\mathit{\Gamma}, \alpha^n) = \mathcal{Q}(\mathit{\Gamma}^n) + \mathcal{Q}(\alpha^n)$$

Further, by Definition 2 we know there exists some i such that $i < n$ and

$$\mathcal{Q}(\mathit{\Gamma}, \tau^i) = \mathcal{Q}(\mathit{\Gamma}^i) + \mathcal{Q}(\tau^i)$$

Since $i < n$ and $n \leq n$, by Definition 3 we know that

$$\mathit{\Gamma}^i \cup \{(\alpha^n \leqslant \tau^i)\} \ \subseteq \ \mathit{\Gamma}^n$$

Therefore,

$$\mathcal{Q}(\mathit{\Gamma}^n) \geq \mathcal{Q}(\mathit{\Gamma}^i) + \mathcal{Q}(\tau^i)$$

and finally we get

$$\mathcal{Q}(\mathit{\Gamma}^n) + \mathcal{Q}(\alpha^n) \geq \mathcal{Q}(\mathit{\Gamma}^i) + \mathcal{Q}(\tau^i)$$

*Case 2 (*S^R-All*).*

$$\frac{\mathit{\Gamma} \vdash \rho_2 \ <:^R \ \rho_1 \qquad \mathit{\Gamma}, (\gamma \leqslant \rho_2) \vdash \tau_1 \ <:^R \ \tau_2}{\mathit{\Gamma} \vdash \forall(\gamma \leqslant \rho_1).\tau_1 \ <:^R \ \forall(\gamma \leqslant \rho_2).\tau_2} \quad (\mathrm{S}^R\text{-}\textsc{All})$$

Firstly, it is simple to show that

$$\mathcal{M}(\mathit{\Gamma}, \rho_1) + \mathcal{M}(\mathit{\Gamma}, \rho_2) < \mathcal{M}(\mathit{\Gamma}, \forall(\gamma \leqslant \rho_1).\tau_1) + \mathcal{M}(\mathit{\Gamma}, \forall(\gamma \leqslant \rho_2).\tau_2)$$

Secondly, the key observation is that by Property 1 we know that

$$\mathcal{Q}(\rho_1) = \mathcal{Q}(\rho_2) = 0$$

As a result we also have that

$$\mathcal{Q}(\mathit{\Gamma}, (\gamma \leqslant \rho_2) = \mathcal{Q}(\mathit{\Gamma}))$$

Thus we have

$$\mathcal{Q}(\mathit{\Gamma}, (\gamma \leqslant \rho_2), \tau_1) + \mathcal{Q}(\mathit{\Gamma}, (\gamma \leqslant \rho_2), \tau_2) = \mathcal{Q}(\mathit{\Gamma}, \tau_1) + \mathcal{Q}(\mathit{\Gamma}, \tau_2)$$

It is thus simple to show that

$$\mathcal{Q}(\mathit{\Gamma}, \tau_1) + \mathcal{Q}(\mathit{\Gamma}, \tau_2) < \mathcal{Q}(\mathit{\Gamma}, \forall(\gamma \leqslant \rho_1).\tau_1) + \mathcal{Q}(\mathit{\Gamma}, \forall(\gamma \leqslant \rho_2).\tau_2)$$

and subsequently we get the desired result.

3.2 Properties of $\mathbf{F}^{\boldsymbol{R}}_{<:}$

One of the most useful aspects of $\mathrm{F}^R_{<:}$, is that it represents a subset of System $\mathrm{F}_{<:}$. That is, not only is any type τ in $\mathrm{F}^R_{<:}$ also a type in System $\mathrm{F}_{<:}$, but subtyping in $\mathrm{F}^R_{<:}$ implies subtyping in System $\mathrm{F}_{<:}$, and typing in $\mathrm{F}^R_{<:}$ implies typing in System $\mathrm{F}_{<:}$. This means that $\mathrm{F}^R_{<:}$ inherits several useful properties of System $\mathrm{F}_{<:}$ metatheory.

In this Section, we discuss some of the properties of $\mathrm{F}^R_{<:}$, and in doing so, we refer to both the typing judgment, and operational semantics. These are identical to those of System $\mathrm{F}_{<:}$, and so are not given here, but are provided in the accompanying technical report [1]. As with subtyping, we often need to refer to several different forms of typing, and we make this distinction by annotating the judgment appropriately. Typing in System $\mathrm{F}_{<:}$ is indicated as $\Gamma \vdash \tau_1 \ : \ \tau_2$, and in $\mathrm{F}^R_{<:}$ as $\Gamma \vdash \tau_1 \ :^R \ \tau_2$.

Subtype Transitivity Unlike other variants on System $\mathrm{F}_{<:}$ [9], $\mathrm{F}^R_{<:}$ retains the subtype transitivity of System $\mathrm{F}_{<:}$.

Theorem 2 (Subtype Transitivity in $\mathbf{F}^R_{<:}$). *For all τ_1, τ_2, and τ_3, if $\Gamma \vdash \tau_1 \ <:^R \ \tau_2$ and $\Gamma \vdash \tau_2 \ <:^R \ \tau_3$, then $\Gamma \vdash \tau_1 \ <:^R \ \tau_3$.*

Proof. Subtype transitivity is proven as part of more general theorem that includes narrowing of the typing context. i.e. we prove the following properties mutually hold:

$$\frac{\Gamma \vdash \tau_1 \ <:^R \ \tau_2 \qquad \Gamma \vdash \tau_2 \ <:^R \ \tau_3}{\Gamma \vdash \tau_1 \ <:^R \ \tau_3}\ (\textsc{Trans}) \qquad \frac{\Gamma_1, (\alpha \leqslant \tau), \Gamma_2 \vdash \tau_1 \ <:^R \ \tau_2 \qquad \Gamma_1 \vdash \tau' \ <:^R \ \tau}{\Gamma_1, (\alpha \leqslant \tau'), \Gamma_2 \vdash \tau_1 \ <:^R \ \tau_2}\ (\textsc{Narrowing})$$

The proof can be found in the associated technical report [1].

Subtyping in $\mathbf{F}^{\boldsymbol{R}}_{<:}$ $\subset$ Subtyping in System $\mathbf{F}_{<:}$ $\mathrm{F}^R_{<:}$ is not a significant change to the semantics of bounded polymorphism from System $\mathrm{F}_{<:}$, in fact subtyping in $\mathrm{F}^R_{<:}$ is a subset of subtyping in System $\mathrm{F}_{<:}$. That is any subtyping that can be derived in $\mathrm{F}^R_{<:}$ can also be derived in System $\mathrm{F}_{<:}$.

Theorem 3 ($\mathbf{F}^R_{<:}$ $\subset$ System $\mathbf{F}_{<:}$). *For all Γ, τ_1, and τ_1, if $\Gamma \vdash \tau_1 \ <:^R \ \tau_2$ then $\Gamma \vdash \tau_1 \ <:^N \ \tau_2$.*

Proof. The result is easily reached by noting that every rule in Figure 4 has a counterpart in Figure 2 that is at least as permissive.

This is a useful property because it implies that existing type checkers need only introduce syntactic checks at key points (when checking subtyping between polymorphic types with different bounds), and do not need significant modifications to the subtyping algorithm.

Subtyping in Kernel $\mathbf{F}_{<:} \subset$ Subtyping in $\mathbf{F}^{R}_{<:}$ $\mathrm{F}^{R}_{<:}$ represents a super-set of Kernel $\mathrm{F}_{<:}$ in terms of subtyping. This provides a useful lower bound on expressiveness. Any valid Kernel $\mathrm{F}_{<:}$ program is also a valid $\mathrm{F}^{R}_{<:}$ program.

Theorem 4 (Kernel $\mathbf{F}_{<:} \subset \mathbf{F}^{R}_{<:}$). *For all Γ, τ_1, and τ_1, if $\Gamma \vdash \tau_1 \ <:^{K} \ \tau_2$ then $\Gamma \vdash \tau_1 \ <:^{R} \ \tau_2$.*

Proof. The result arises from the fact that the S^{R}-Kernel-All rule in Figure 4 is the exact rule for bounded polymorphism in Kernel $\mathrm{F}_{<:}$. Thus, subtyping in $\mathrm{F}^{R}_{<:}$ is at least as expressive than subtyping in Kernel $\mathrm{F}_{<:}$.

Type Safety As subtyping in $\mathrm{F}^{R}_{<:}$ is a subset of subtyping in System $\mathrm{F}_{<:}$, and the two calculi have otherwise identical typing, it follows that every well-typed program in $\mathrm{F}^{R}_{<:}$ is well-typed in System $\mathrm{F}_{<:}$. It is thus unsurprising that given System $\mathrm{F}_{<:}$'s type safety, and that the two calculi have identical operational semantics, any well-typed $\mathrm{F}^{R}_{<:}$ program is guaranteed to not get stuck. In other words, $\mathrm{F}^{R}_{<:}$ is type safe.

Theorem 5 (Type Safety). *For all Γ, t and τ, if $\Gamma \vdash t \ :^{R} \ \tau$, then reduction of t is guaranteed to not get stuck.*

Proof. The result arises immediately from the type safety of System $\mathrm{F}_{<:}$, and the result in Theorem 3.

Minimal Typing As mentioned in Section 2 $\mathrm{F}^{\top}_{<:}$ [5] is another variant of System $\mathrm{F}_{<:}$ that allows for subtyping of bounded polymorphism that is both decidable and contra-variant on type bounds. We also mentioned that typing in $\mathrm{F}^{\top}_{<:}$ is not minimal [6], and thus some terms can be typed with two different types that are not related by subtyping. Specifically, in $\mathrm{F}^{\top}_{<:}$, the term $t = \Lambda(X \leqslant \mathtt{Int}).\lambda(x : X).x$ can be shown to have both the type $\tau_1 = \forall(X \leqslant \mathtt{Int}).X \to X$, and the type $\tau_2 = \forall(X \leqslant \mathtt{Int}).X \to \mathtt{Int}$. In $\mathrm{F}^{\top}_{<:}$, these two types are unrelated, and have no lower bound. The implications of this lack of minimality are that the standard typing algorithm for System $\mathrm{F}_{<:}$ is not complete for $\mathrm{F}^{\top}_{<:}$, and will assign t one type, but not the other, and any usage where t is required to be typed with both types will not type check.

The reason for the loss of minimal typing in $\mathrm{F}^{\top}_{<:}$ is due to a "rebounding" of type variables during subtyping to $\top$. Subtyping of the body of a polymorphic type is done with reduced type information as the bound of the type variable is treated as $\top$, hiding the relationship between the type variable and its bound.

A central motivation in designing $\mathrm{F}^{R}_{<:}$ is to provide reliable and expected behaviour to type checkers, that allows for understandable error messages in type checking. The loss of minimal typing does not provide these assurances. For instance, it seems reasonable to expect that in the example above, τ_1 should subtype τ_2, and if it doesn't a satisfying reason should be provided by the type checker. Subtyping in $\mathrm{F}^{R}_{<:}$ does not perform the same "rebounding", and as a result does not suffer from the same loss of minimal typing.

$$\begin{array}{llll} \tau ::= & & & \\ \quad \top & \textit{top} & x.L & \textit{selection} \\ \quad \bot & \textit{bottom} & \{L\ :\tau \ldots \tau\} & \textit{declaration} \\ \quad \forall(x:\tau).\tau^x & \textit{function} & & \end{array}$$

Fig. 8. $D_{<:}$ Type Syntax

Theorem 6 (Minimal Typing). *For all Δ, Γ, t, τ_1, and τ_2, if $\Delta;\Gamma \vdash t\ :^R\ \tau_1$ and $\Delta;\Gamma \vdash t\ :^R\ \tau_2$, then there exists some τ, such that $\Delta;\Gamma \vdash t\ :^R\ \tau$, $\Delta;\Gamma \vdash \tau\ <:^R\ \tau_1$, and $\Delta;\Gamma \vdash \tau\ <:^R\ \tau_2$.*

Proof. The proof can be found in the associated technical report [1].

4 Separating $D_{<:}$

$D_{<:}$ is a calculus related to System $F_{<:}$ that includes abstract type members and dependent functions, and serves to model core aspects of the Scala type system. The syntax of $D_{<:}$ is given in Figure 8, and at first glance does not immediately resemble that of System $F_{<:}$. Most noticeably, $D_{<:}$ does not include any type variables. The expressiveness of $D_{<:}$ derives from being able to capture System $F_{<:}$ using path types ($x.L$) and dependent function types ($\forall(x:\tau_1).\tau_2$).

A type in $D_{<:}$ is either $\top$, $\bot$, a type declaration $\{L\ :\tau_1 \ldots \tau_2\}$, a selection type $x.L$, or a dependent function type $\forall(x:\tau_1).\tau_2^x$.

In $D_{<:}$, type declarations ($\{L\ :\tau_1 \ldots \tau_2\}$) define a type. Given a path x to the type definition, the defined type can be used by selection on the path: $x.L$. That is, if value x has type $\{L\ :\tau_1 \ldots \tau_2\}$, then $x.L$ refers to the defined type, where τ_1 is the lower bound, and τ_2 the upper bound. This is useful when combined with the dependent function types of $D_{<:}$. The return types of functions in $D_{<:}$ can be dependent on the argument. This dependence is indicated in the syntax by a super-script of the variable identifying the argument. i.e. $\forall(x:\tau_1).\tau_2^x$ indicates that x is free in τ_2.

$D_{<:}$ subtyping is defined in Figure 9, and is indicated by $\Gamma \vdash \tau_1\ <:^D\ \tau_2$. Subtyping is bound above by $\top$ (Top) and below by $\bot$ (Bot). Subtyping is explicitly reflexive (Rfl). Selection types subtype their upper bounds (Sel1), and super type their lower bounds (Sel2). Subtyping of type declarations are contra-variant with respect to the lower bounds, and covariant with respect to their upper bounds (Bnd). Finally, subtyping of dependent function types is contra-variant with respect to the argument types, and covariant with respect to the return types, with the return types being dependent on the arguments (All).

Coupling type declarations together with dependent function types allows for similar functionality to $F_{<:}$. That is, we can use the encoding below to capture bounded polymorphism from System $F_{<:}$ in $D_{<:}$.

$$[\![\top]\!] \triangleq \top \quad (1) \qquad [\![\tau_1 \to \tau_2]\!] \triangleq \forall(x_\alpha : [\![\tau_1]\!]).[\![\tau_2]\!]^{x_\alpha} \quad (3)$$

$$[\![\alpha]\!] \triangleq x_\alpha.A \quad (2) \qquad [\![\forall(\alpha \leqslant \tau_1).\tau_2]\!] \triangleq \forall(x_\alpha : \{A\ :\bot \ldots [\![\tau_1]\!]\}).[\![\tau_2]\!]^{x_\alpha} \quad (4)$$

$$\Gamma \vdash \tau \ <:^D \ \top \quad (\textsc{Top}) \qquad \Gamma \vdash \bot \ <:^D \ \tau \quad (\textsc{Bot}) \qquad \Gamma \vdash \tau \ <:^D \ \tau \quad (\textsc{Rfl})$$

$$\frac{\Gamma(x) = \{L \ : \tau_1 \ldots \tau_2\}}{\Gamma \vdash x.L \ <:^D \ \tau_2} \ (\textsc{Sel1}) \qquad \frac{\Gamma(x) = \{L \ : \tau_1 \ldots \tau_2\}}{\Gamma \vdash \tau_1 \ <:^D \ x.L} \ (\textsc{Sel2})$$

$$\frac{\begin{array}{l}\Gamma \vdash \tau_1 \ <:^D \ \tau_2 \\ \Gamma \vdash \tau_2 \ <:^D \ \tau_3\end{array}}{\Gamma \vdash \tau_1 \ <:^D \ \tau_3} \ (\textsc{Trans}) \qquad \frac{\begin{array}{c}\Gamma \vdash \tau_2 \ <:^D \ \tau_1 \\ \Gamma, (x : \tau_2) \vdash \tau_1' \ <:^D \ \tau_2'\end{array}}{\Gamma \vdash \forall(x : \tau_1).\tau_1' \ <:^D \ \forall(x : \tau_2).\tau_2'} \ (\textsc{All})$$

$$\frac{\Gamma \vdash \tau_2 \ <:^D \ \tau_1 \qquad \Gamma \vdash \tau_1' \ <:^D \ \tau_2'}{\Gamma \vdash \{L \ : \tau_1 \ldots \tau_1'\} \ <:^D \ \{L \ : \tau_2 \ldots \tau_2'\}} \ (\textsc{Bnd})$$

Fig. 9. $D_{<:}$ Subtyping

The above encoding is in fact not enough to demonstrate the undecidability of $D_{<:}$ due to the fact that subtyping of System $F_{<:}$ types is not equivalent to subtyping of their encoding in $D_{<:}$. That is, while the following holds:

$$\Gamma \vdash \tau_1 \ <:^N \ \tau_2 \ \Rightarrow \ [\![\Gamma]\!] \vdash [\![\tau_1]\!] \ <:^D \ [\![\tau_2]\!]$$

the inverse does not.

$$[\![\Gamma]\!] \vdash [\![\tau_1]\!] \ <:^D \ [\![\tau_2]\!] \ \not\Rightarrow \ \Gamma \vdash \tau_1 \ <:^N \ \tau_2$$

The reasons for this are due to the fact that functions in System $F_{<:}$ are unrelated to polymorphic types, but in $D_{<:}$ they are both captured using dependent function types. A simple counter-example to the inverse are the types $\forall(\alpha \leqslant \top).\top$ and $\top \rightarrow \top$. Both polymorphic types and arrow types in System $F_{<:}$ are encoded as dependent function types, and $[\![\forall(\alpha \leqslant \top).\top]\!]$ subtypes $[\![\top \rightarrow \top]\!]$, however, it is clear that $\forall(\alpha \leqslant \top).\top$ does not subtype $\top \rightarrow \top$. The full proof for this was demonstrated by Hu and Lhoták [9]. This result does not affect the undecidability result for $D_{<:}$, as the proof of undecidability in System $F_{<:}$ does not rely on arrow types. Pierce's [13] proof of undecidability uses a subset of System $F_{<:}$ that does not include arrow types, and thus while the encoding of System $F_{<:}$ into $D_{<:}$ is not complete, it is possible to define a complete encoding of the fragment of System $F_{<:}$ that is undecidable. We leave the details of this to Hu and Lhoták [9].

Figure 10 presents the syntax for $D^R_{<:}$, a separated variant of $D_{<:}$. $D^R_{<:}$ introduces a similar separation on syntax to that of $F^R_{<:}$. Where $F^R_{<:}$ places a restriction on the bounds of type variables, $D^R_{<:}$ places a restriction on the bounds of type members. That is, we distinguish restricted type definitions from unrestricted ones. A restricted type definition ($\{R \ : \rho_1 \ldots \rho_2\}$) is a type definition that does not contain any dependent function types in either the upper or lower bound. As with $F^R_{<:}$This restriction is indicated by restricted types (ρ). Note: restricted types are only separated from dependent function types, and not function types in general. As we have already mentioned, dependent functions in $D_{<:}$

$$
\begin{array}{ll}
\tau ::= & \textbf{D}^{R}_{<:}\ \textbf{Type} \\
\quad \top & \textit{top} \\
\quad \bot & \textit{bottom} \\
\quad \{U\ : \tau \ldots \tau\} & \textit{declaration} \\
\quad \{R\ : \rho \ldots \rho\} & \textit{restricted declaration} \\
\quad x.L & \textit{selection} \\
\quad \forall(x : \tau).\tau^{x} & \textit{dependent function}
\end{array}
\qquad
\begin{array}{ll}
L ::= & \textbf{Type Label} \\
\quad U & \textit{unrestricted} \\
\quad R & \textit{restricted} \\
\rho ::= & \textbf{D}^{R}_{<:}\ \textbf{Restricted Type} \\
\quad \top & \textit{top} \\
\quad \bot & \textit{bottom} \\
\quad \{L\ : \rho \ldots \rho\} & \textit{declaration} \\
\quad \forall(x : \rho).\rho & \textit{function} \\
\quad x.R & \textit{selection}
\end{array}
$$

Fig. 10. $\mathrm{D}^{R}_{<:}$ Type Syntax

$$
\Gamma \vdash \tau\ <:^{R}\ \top \quad (\textsc{Top}^{R}) \qquad \Gamma \vdash \bot\ <:^{R}\ \tau \quad (\textsc{Bot}^{R}) \qquad \Gamma \vdash x.L\ <:^{R}\ x.L \quad (\textsc{Rfl}^{R})
$$

$$
\frac{\begin{array}{c}\Gamma(x) = \{L\ : \tau_1 \ldots \tau_2\} \\ \Gamma \vdash \tau_2\ <:\ \tau\end{array}}{\Gamma \vdash x.L\ <:^{R}\ \tau}\ (\textsc{Sel1}^{R}) \qquad \frac{\begin{array}{c}\Gamma(x) = \{L\ : \tau_1 \ldots \tau_2\} \\ \Gamma \vdash \tau\ <:\ \tau_1\end{array}}{\Gamma \vdash \tau\ <:^{R}\ x.L}\ (\textsc{Sel2}^{R})
$$

$$
\frac{\Gamma \vdash \tau_2\ <:^{R}\ \tau_1 \qquad \Gamma \vdash \tau_1'\ <:^{R}\ \tau_2'}{\Gamma \vdash \{L\ : \tau_1 \ldots \tau_1'\}\ <:^{R}\ \{L\ : \tau_2 \ldots \tau_2'\}}\ (\textsc{Bnd}^{R})
$$

$$
\frac{\Gamma, (x : \tau) \vdash \tau_1\ <:^{R}\ \tau_2}{\Gamma \vdash \forall(x : \tau).\tau_1^{x}\ <:^{R}\ \forall(x : \tau).\tau_2^{x}}\ (\textsc{All-Kernel}^{R})
$$

$$
\frac{\Gamma \vdash \rho_2\ <:^{R}\ \rho_1 \qquad \Gamma, (x : \rho_2) \vdash \tau_1\ <:^{R}\ \tau_2}{\Gamma \vdash \forall(x : \rho_1).\tau_1'^{x}\ <:^{R}\ \forall(x : \rho_2).\tau_2'^{x}}\ (\textsc{All}^{R})
$$

Fig. 11. $\mathrm{D}^{R}_{<:}$ Subtyping

capture both abstraction over values and abstraction over types, while in System $\mathrm{F}_{<:}$, bounded polymorphism only captures abstraction over types. To this end, we allow restricted types in $\mathrm{D}^{R}_{<:}$ to include non-dependent function types $(\forall(x : \tau_1).\tau_2)$ that can be identified by the absence of the variable super-script indicating the return type is dependent on the argument type.

4.1 Restricted Subtyping in $\mathbf{D}^{R}_{<:}$

Subtyping for $\mathrm{D}^{R}_{<:}$ is defined in Figure 11. There are several differences between the restricted form of subtyping and that of $\mathrm{D}_{<:}$. As with bounded polymorphism in $\mathrm{F}^{R}_{<:}$, subtyping of dependent function types in $\mathrm{D}^{R}_{<:}$ can be proven using one of two rules: (i) $\textsc{Kernel-All}^{R}$, a subtype rule that enforces invariance on the argument type, and (ii) $\textsc{All}^{R}$, a subtype rule that allows covariance on function argument types of the form ρ.

$$\begin{aligned}
\mathcal{D}(\Gamma, x.L) &= 1 + max(\mathcal{D}(\Gamma, \tau_1), \mathcal{D}(\Gamma, \tau_2)) \\
&\quad \textit{where } \Gamma \vdash x \ : \ \{L \ : \tau_1 \ldots \tau_2\} \\
\mathcal{D}(\Gamma, \forall(x : \tau_1).\tau_2) &= 1 + max(\mathcal{D}(\Gamma, \tau_1), \mathcal{D}(\Gamma, \tau_2))
\end{aligned}
\qquad
\begin{aligned}
\mathcal{D}(\Gamma, \top) &= 0 \\
\mathcal{D}(\Gamma, \bot) &= 0 \\
\mathcal{D}(\Gamma, \forall(x : \tau_1).\tau_1^x) &= 0
\end{aligned}$$

Fig. 12. Quantification Depth: the depth of the next dependent function type.

Subtyping in $D^R_{<:}$ also differs from standard $D_{<:}$ subtyping in how reflexivity and transitivity are formalized. Explicit subtype reflexivity in $D^R_{<:}$ is restricted to type selections ($x.L$). This is similar to the modification $F^N_{<:}$ makes to traditional System $F_{<:}$.

Subtype Transitivity As in $F^N_{<:}$, the explicit transitivity rule, Trans, is removed. Transitivity rules are generally difficult to design an algorithm for as it is not always clear what to choose for the middle type (τ_2 in Trans). To try and recapture some level of transitivity, we modify the subtype rules for upper and lower bounds by introducing a level of transitivity (see Sel1^R and Sel2^R in Figure 11). This mirrors the difference in transitivity between the $F^N_{<:}$ version of System $F_{<:}$ subtyping, and the original definition of Cardelli [4], where the explicit transitivity rule was removed, and replaced with a modified rule for type variable subtyping that accounted for transitivity. In the $F^N_{<:}$ (and $F^R_{<:}$) rule set, general transitivity is provable as a property of subtyping. Unfortunately the same cannot be said for $D^R_{<:}$. Subtyping in $D^R_{<:}$ is not transitive.

The reason for the loss of transitivity is due to the the relationship between the upper and lower bounds of type definitions: there is no requirement that the lower bound subtypes the upper bound. Precursor calculi to $D_{<:}$ attempted to enforce this invariant, but due to a complex set of reasons, this is not generally possible in the presence of another Scala feature: intersection types. A critical insight of previous work on the DOT calculus, is that ill-formed type bounds not necessarily unsound [2] [3] [16] since ultimately at run-time, any type bounds must be fulfilled by some value (a witness), and only well-formed bounds may be fulfilled. The details are interesting, but are fairly complex and so we do not address them further.

4.2 Subtype Decidability in $D^R_{<:}$

The subtype decidability argument for $D^R_{<:}$ is much like that of $F^R_{<:}$: We define an ordering on the number of dependent function types and the depth of a type down to the next dependent function type. We define the measures $\mathcal{D}$ and $\mathcal{Q}$ in Figures 12 and 13. As with $F^R_{<:}$, the finite measure of $D^R_{<:}$ the lexicographic ordering:

$$\mathcal{M} = \mathcal{D} \times \mathcal{Q}$$

We can now prove subtype decidability for $D^R_{<:}$ using much the same logic as we did for $F^R_{<:}$. We define a subtype algorithm for $D^R_{<:}$: $\texttt{subtype}_{D^R_{<:}}$. As with

$$\begin{aligned}
\mathcal{Q}(\top) &= 0 \\
\mathcal{Q}(\alpha) &= 0 \\
\mathcal{Q}(\forall(x:\tau_1).\tau_2) &= \mathcal{Q}(\tau_1) + \mathcal{Q}(\tau_2) \\
\mathcal{Q}(\forall(x:\tau_1).\tau_2^x) &= 1 + \mathcal{Q}(\tau_1) + \mathcal{Q}(\tau_2)
\end{aligned}
\qquad
\begin{aligned}
\mathcal{Q}(\Gamma) &= \mathcal{Q}(\tau) + \mathcal{Q}(\Gamma') \\
&\quad \textit{where } \Gamma = \Gamma', (\alpha \leqslant \tau) \\
\mathcal{Q}(\Gamma, \tau^n) &= \mathcal{Q}(\Gamma^n) + \mathcal{Q}(\tau^n)
\end{aligned}$$

Fig. 13. Quantification Size: the number of dependent function types in a $D^R_{<:}$ type.

`subtype`$F^R_{<:}$, `subtype`$_{D^R_{<:}}$ is the inversion of the rule set in Figure 11. A sketch of the proof of decidability is given below.

Theorem 7 (Subtype Decidability of $\mathbf{D}^R_{<:}$). *For all Γ, τ_1, and τ_2, **subtype**$_{D^R_{<:}}(\Gamma$, τ_1, $\tau_2)$ is guaranteed to terminate.*

Proof. As with the proof of subtype decidability for $F^R_{<:}$, it is fairly simple to demonstrate that for any call `subtype`$_{D^R_{<:}}(\Gamma, \tau_1, \tau_2)$, by the measure $\mathcal{M}$, any resulting calls to `subtype`$_{D^R_{<:}}$ are strictly smaller.

4.3 Type Safety

Subtyping in $D^R_{<:}$ is a subset of subtyping in $D^R_{<:}$, and as with $F^R_{<:}$, this affords $D^R_{<:}$ many of the properties of $D_{<:}$. Type safety is one such property, and arises immediately from Theorem 8 below.

Theorem 8 ($\mathbf{D}^R_{<:} \subset \mathbf{D}_{<:}$). *For all Γ, τ_1, and τ_1, if $\Gamma \vdash \tau_1 \;<:^R\; \tau_2$ then $\Gamma \vdash \tau_1 \;<:^D\; \tau_2$.*

Proof. The result follows directly from the fact that for every rule in Figure 11, there is an corresponding rule in Figure 9 that is at least as permissive.

4.4 Expressiveness

The expressiveness of $D^R_{<:}$ is still an open question, and can only properly be addressed in an empirical way. It is worth noting that our definition of $D^R_{<:}$ is similar in its conception to $F^R_{<:}$, in that we take care to only place restrictions on the use of *dependent* function types, and not function types in general. Argument types may still refer to function types that do not meaningfully modify the context. In fact, $D^R_{<:}$ is actually still too strict, and could potentially be relaxed further in its definition. There is no reason that subtyping of non-dependent function types need to have the same restrictions placed on them as dependent function types. It is likely possible that we could extend the subtyping in Figure 11 with the following rule.

$$\frac{\Gamma \vdash \tau_2 \;<:^R\; \tau_1 \qquad \Gamma, (x:\tau_2) \vdash \tau_1' \;<:^R\; \tau_2'}{\Gamma \vdash \forall(x:\tau_1).\tau_1' \;<:^R\; \forall(x:\tau_2).\tau_2'} \quad (\textsc{All2}^R)$$

While at first glance the above rule looks like it might re-introduce undecidability, note that the return types do not depend on the argument type: that is they lack the super-script ^{x}. In this case, while we are still introducing differing types to the context, they are not referred to in the return types, and so are of no consequence. Such a rule is not without potential problems however. It is not immediately clear what the above rule would mean for other properties of $\mathrm{D}^R_{<:}$.

5 Related Work

5.1 Strong $\mathrm{F}_{<:}$ and Strong $\mathrm{D}_{<:}$

Hu and Lhoták [9], defined decidable variants of System $\mathrm{F}_{<:}$ and $\mathrm{D}_{<:}$ named Strong $\mathrm{F}_{<:}$ and Strong $\mathrm{D}_{<:}$ respectively. Their approach introduces a second typing context to subtyping, one for each type, giving subtyping the following form.

$$\Gamma_1 \gg \tau_1 \;<:\; \tau_2 \ll \Gamma_2$$

Hu and Lhoták refer to this as "stare-at subtyping". Type bounds in τ_1 are appended to Γ_1, while type bounds in τ_2 are appended to Γ_2. This separation of contexts ensures that there is no problematic "rebounding" [13] that might lead to an expansive context. There are however some short comings to this technique, specifically subtype transitivity is lacking in both type systems. Below we demonstrate an instance of subtype transitivity that is lost in Strong $\mathrm{F}_{<:}$.

$$A = \forall(\alpha \leqslant \top).\alpha \qquad B = \forall(\alpha \leqslant \texttt{Int}).\alpha \qquad C = \forall(\alpha \leqslant \texttt{Int}).\texttt{Int}$$

While it can be shown that both A subtypes B and B subtypes C in Strong $\mathrm{F}_{<:}$, the transitive case cannot be derived, i.e. $A \not<: C$. During subtyping of bounded polymorphism in Strong $\mathrm{F}_{<:}$ (and Strong $\mathrm{D}_{<:}$), two typing contexts are maintained, each updated with the bounds of the relevant type. While subtype reflexivity of type variables allows α to subtype α when deriving $A \;<:\; B$, this is not so when attempting to derive $A \;<:\; C$. This is not an especially complex example, and is a subtyping that programmers might expect to hold.

Using a syntactic separation we are able to retain subtype transitivity in $\mathrm{F}^R_{<:}$. The trade off is that we exclude a specific class of programs. These programs, however, can be identified syntactically, and thus $\mathrm{F}^R_{<:}$ enables the type checker to better guide programmers in fixing their error.

While we have already mentioned that $\mathrm{D}^R_{<:}$ is not indeed transitive, this is due to the potential for "bad bounds" on type definitions, and the problems associated with ensuring "good bounds". $\mathrm{D}^R_{<:}$ does not exclude the types of transitivity seen in Strong $\mathrm{D}_{<:}$ which lacks transitivity, not only due to the "bad bounds" problem, but also for the same reasons Strong $\mathrm{F}_{<:}$ does. More specifically, the subtyping $A \;<:\; C$ can be derived in $\mathrm{F}^R_{<:}$. Similarly, the equivalent example in $\mathrm{D}_{<:}$ is not derivable in Strong $\mathrm{D}_{<:}$, but is derivable in $D^R_{<:}$.

5.2 Wyvern

Mackay et al. [10] defined two decidable variants of Wyvern [11] [12], a language related to Scala, featuring type members, dependent function types, recursive types, and a limited form of intersection types called type refinements. Their variants of Wyvern were named $\mathit{Wyv}_{\mathit{fix}}$ and $\mathit{Wyv}_{\mathit{self}}$, and took different approaches to ensuring decidability.

Interestingly, $\mathit{Wyv}_{\mathit{fix}}$ introduces essentially the same double headed form of subtyping that Hu and Lhoták [9] did. An independent discovery, Mackay et. al [10] use the the double headed subtyping form in a slightly different setting with the same purpose. While the Strong Kernel $\mathrm{D}_{<:}$ of Hu and Lhoták [9] does not include recursive types or any form of intersection types, $\mathit{Wyv}_{\mathit{fix}}$ does. $\mathit{Wyv}_{\mathit{fix}}$ suffers from the same loss of transitivity that Strong Kernel $\mathrm{D}_{<:}$ does, and as such prohibits several useful forms of expressiveness.

$\mathit{Wyv}_{\mathit{self}}$ does not use a double headed form, and rather makes use of a Material/Shape separation inspired by the work of Greenman et al. [7]. $\mathit{Wyv}_{\mathit{self}}$ does not allow for contra-variance on the argument types of dependent functions.

6 Conclusion

In this paper we have presented $\mathrm{F}^{R}_{<:}$, a variant of System $\mathrm{F}_{<:}$ that is decidable in its subtyping, while retaining several of the desirable qualities of System $\mathrm{F}_{<:}$. Our approach is largely in the form of a syntactic restriction on types, rather than a significant departure from the semantics of subtyping bounded polymorphism. Further, we have shown that this approach can be applied to another related calculus, $\mathrm{D}_{<:}$, to get $\mathrm{D}^{R}_{<:}$, a type system that models core concepts of Scala. $\mathrm{D}^{R}_{<:}$ does not sacrifice certain instances of transitivity and expressiveness that other similar designs in the past have.

In future work, we hope to show that this approach can be further applied to the much more complex DOT calculus, by incorporating intersection types and recursive types. Further, the expressiveness of these restrictions is still an open question. While there are many languages that incorporate bounded polymorphism similar to System $\mathrm{F}_{<:}$, it is not clear how many of them allow for bounded polymorphism within type bounds, the pattern that $\mathrm{F}^{R}_{<:}$ restricts. What is yet harder to say is what the restrictions of $\mathrm{D}^{R}_{<:}$ mean for Scala. As we have noted, the Scala type system potentially suffers from more undecidability issues than just those related to dependent function types, recursive types in Scala are also a source of undecidability [10], and so $\mathrm{D}^{R}_{<:}$ does not ensure decidability of Scala's type system.

To settle the question of expressiveness, it would be valuable to conduct an empirical survey of existing languages with bounded polymorphism to determine either (i) how many of them already restrict the usage of bounded polymorphism in the way that $\mathrm{F}^{R}_{<:}$ and $\mathrm{D}^{R}_{<:}$, or (ii) how many of them are permit such patterns, but are not in practice used by the respective programming communities.

References

1. Syntactically Restricting Bounded Polymorphism for Decidable Subtyping - Technical Report (2020), https://doi.org/10.5281/zenodo.4039832

2. Amin, N., Moors, A., Odersky, M.: Dependent object types. In: 19th International Workshop on Foundations of Object-Oriented Languages (2012)
3. Amin, N., Rompf, T., Odersky, M.: Foundations of Path-dependent Types. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. OOPSLA '14 (2014)
4. Cardelli, L., Martini, S., Mitchell, J.C., Scedrov, A.: An Extension of System F with Subtyping. In: Ito, T., Meyer, A.R. (eds.) Theoretical Aspects of Computer Software. pp. 750–770. Springer Berlin Heidelberg, Berlin, Heidelberg (1991)
5. Castagna, G., Pierce, B.C.: Decidable Bounded Quantification. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 94 (1994)
6. Castagna, G., Pierce, B.C.: Corrigendum: Decidable Bounded Quantification. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 95 (1995)
7. Greenman, B., Muehlboeck, F., Tate, R.: Getting F-bounded Polymorphism into Shape. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14 (2014)
8. Grigore, R.: Java Generics Are Turing Complete. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL 2017 (2017)
9. Hu, J.Z.S., Lhoták, O.: Undecidability of D¡: And Its Decidable Fragments. Proc. ACM Program. Lang. (POPL) (2019)
10. Mackay, J., Potanin, A., Aldrich, J., Groves, L.: Decidable subtyping for path dependent types. Proc. ACM Program. Lang. (POPL) (2019)
11. Nistor, L., Kurilova, D., Balzer, S., Chung, B., Potanin, A., Aldrich, J.: Wyvern: A Simple, Typed, and Pure Object-oriented Language. In: Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance. MASPEGHI '13 (2013)
12. Omar, C., Kurilova, D., Nistor, L., Chung, B., Potanin, A., Aldrich, J.: Safely composable type-specific languages. In: Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586 (2014)
13. Pierce, B.C.: Bounded Quantification is Undecidable. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '92 (1992)
14. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
15. Rapoport, M., Kabir, I., He, P., Lhoták, O.: A Simple Soundness Proof for Dependent Object Types. Proc. ACM Program. Lang. (OOPSLA) (2017)
16. Rompf, T., Amin, N.: Type Soundness for Dependent Object Types (dot). In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2016 (2016)
17. Wadler, P., Blott, S.: How to Make Ad-Hoc Polymorphism Less Ad Hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL 89 (1989)