

Decidable Subtyping for Path Dependent Types

JULIAN MACKAY, ALEX POTANIN, and LINDSAY GROVES, Victoria University of Wellington, New Zealand

JONATHAN ALDRICH, Carnegie Mellon University, USA

Path dependent types have long served as an expressive component of the Scala programming language. They allow for the modelling of both bounded polymorphism and a degree of nominal subtyping. Nominality in turn provides the ability to capture first class modules. Thus a single language feature gives rise to a rich array of expressiveness. Recent work has proven path dependent types sound in the presence of both intersection and recursive types, but unfortunately typing remains undecidable, posing problems for programmers who rely on the results of type checkers. The Wyvern programming language is an object oriented language with path dependent types, recursive types and first class modules. In this paper we define two variants of Wyvern that feature decidable typing, along with machine checked proofs of decidability. Despite the restrictions, our approaches retain the ability to encode the parametric polymorphism of Java generics along with many idioms of the Scala module system.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*;

Additional Key Words and Phrases: Language Design, Path Dependent Types, Decidability, Subtyping, Nominal Subtyping, Structural Subtyping, Object Oriented Languages, Functional Languages, Wyvern, Scala

ACM Reference Format:

Julian Mackay, Alex Potanin, Lindsay Groves, and Jonathan Aldrich. 2020. Decidable Subtyping for Path Dependent Types. *Proc. ACM Program. Lang.* 4, POPL, Article 66 (January 2020), 27 pages. <https://doi.org/10.1145/3371134>

1 INTRODUCTION

A *type member* is a named member within an object that denotes a type. We can refer to the type denoted by type member L of some object o with the notation $o.L$. The semantics of L are dependent on how it is accessed, i.e. the path taken in the selection. To look up the type of $o.L$, we must look up the definition of L in the type of the *path* o . Thus $o.L$ is a *path dependent type*.

Practically, path dependent types have wide utility. One application is providing a more modular form of bounded parametric polymorphism. Parametric polymorphism was originally designed for functional languages such as ML [Harper 2012], and has more recently been adopted for Object Oriented languages such as C++, Java, C# and Scala. While parametric polymorphism is useful, it can also make types verbose. The example below shows how parametric polymorphism in Java can be encoded using type members. To begin with, we define an empty class `Object`. We then define `Cell` with a type parameter E bounded by `Object` and a field of that type. `Cell` turns into a type with a type member E that is also bounded by `Object`, and has a field of the member type (denoted `this.E` within the surrounding object). A client that instantiates the `Cell` with type `int` can refer to the type of `aCell.member` abstractly via the syntax `aCell.E`, something that is not possible in Java:

Authors' addresses: Julian Mackay; Alex Potanin; Lindsay Groves, School of Engineering and Computer Science, Victoria University of Wellington, New Zealand, julian@ecs.vuw.ac.nz, alex@ecs.vuw.ac.nz, lindsay@ecs.vuw.ac.nz; Jonathan Aldrich, School of Computer Science, Carnegie Mellon University, USA, aldrich@cs.cmu.edu.

2020. 2475-1421/2020/1-ART66
<https://doi.org/10.1145/3371134>

```

1 class Object{}
2 class Cell<E extends Object>{
3     E member;
4 }
5 Cell<Integer> aCell =
6     new Cell<Integer>(...);
7 Integer i = aCell.member;

```

```

1 type Object = {}
2 type Cell = {this ⇒
3     type E <: Object;
4     val member:this.E
5 }
6 val aCell : Cell =
7     new Cell{type E = Integer}
8 val i:aCell.E = aCell.member

```

The ability to access type members within an object increases expressiveness, both when dealing with parametric data structures and more interestingly when modeling module systems or handling dependency injection [Odersky and Zenger 2005]. Type members were first introduced in BETA [Kristensen et al. 1987; Madsen and Moller-Pedersen 1989], and have gained additional prominence recently due to their inclusion in the Scala language [Odersky et al. 2004; Rompf and Amin 2016]. Recently Amin et al. have studied the Dependent Object Types (DOT) calculus as a sound core calculus for Scala [Amin et al. 2012, 2014]. The DOT calculus includes a structural type system that adds type members to model parametric polymorphism as well as a notion of nominality.

The past few years has seen much work elaborating on the foundations of *path dependent types*. They were finally proven type safe by Amin et al. in 2014 using a big step semantics in μ DOT, an early variant of DOT that includes recursive types but not intersection types. Subsequent work derived a type safety proof for the full DOT that included recursive types and full intersection and union types using a small step semantics [Rompf and Amin 2016] that was unfortunately complex. A simpler and more tractable proof was derived [Rapoport et al. 2017] to aid language designers in the construction of extensions of DOT. Many of the problems associated with deriving type safety come down to the many interdependent properties of the DOT calculus, particularly subtype transitivity and environment narrowing (the maintaining of well-formedness when types within the environment are narrowed to more specific types). While type safe, subtyping in DOT is undecidable [Rompf and Amin 2016] due to its ability to encode System $F_{<}$. [Pierce 1992]. Perhaps unsurprisingly, those same properties that complicated the proof of type safety (subtype transitivity and environment narrowing) for path dependent types, confound the construction of a decision procedure for subtyping.

Unfortunately, rich type systems come at a cost: subtyping of Java generics is undecidable [Grigore 2017]. Undecidability in Java comes from recursive inheritance: a class `String` implements `Comparable<String>`, so the `String` type is being defined in terms of itself—and as a result, any procedure intended to check subtyping will loop on some examples. As of the writing of this paper, it is possible to construct such an example in Java that crashes the `javac` compiler with a stack overflow exception. While such errors are relatively rare, we would ideally like to provide useful guidance to programmers in all error cases.

Scala subsumes not only Java’s type system, but introduces several instances of added expressiveness on top of it that further complicate the derivation of a decision procedure for subtyping. Coupled with DOT’s encoding of System $F_{<}$, the construction of any decidable type system that includes path dependent and recursive types must address all of these issues.

While Nieto did develop a decidable subset of the DOT calculus [Nieto 2017], theirs lacked recursive types, a key feature of both DOT and Scala, and critical for the encoding of Java generics.

Greenman et al. defined a decidable subset of Java subtyping by distinguishing between those types that were responsible for cyclic subtype definitions—called *shapes*—and those types that

weren't—called *materials*. They found that a simple restriction on the uses of shapes called Material-Shape Separation resulted in decidable subtyping, without prohibiting commonly-used idioms. Greenman et al. did not, however, apply the approach to a system with path dependent types, a setting which presents a number of additional challenges.

This paper adapts the Material-Shape Separation approach to design two decidable variants of Core Wyvern [Wyvern 2019], a core programming language closely related to the DOT calculus. Greenman et al.'s approach does not solve the full problem in the more complicated setting of DOT, so we identify two additional restrictions, either of which is sufficient to ensure decidability when layered on top of Material-Shape Separation. We demonstrate that despite these restrictions, our approach supports all the same patterns of generic Java class usage identified by Greenman et al., while also expressing key motivating examples of type members in the context of Scala and advanced module systems. We further categorize the decidability issues associated with Wyvern with the aim of furthering the understanding of subtyping in the presence of path dependent and recursive types. The rest of this paper is organized as follows:

- Section 2 provides a background on path dependent types and their subtyping.
- Section 3 introduces a calculus for the language Wyv_{core} , and identifies the problematic aspects of constructing a finite subtype algorithm for Wyv_{core} .
- Section 4 introduces Wyv_{self} a decidable variant of Wyv_{core} that restricts recursive types, along with a sketch of a proof of subtype decidability that is formalized in Coq and an encoding of Greenman et al.'s decidable Java generics.
- Section 5 introduces Wyv_{fix} a decidable variant of Wyv_{core} that provides fixed environments during subtyping, along with a sketch of a proof of subtype decidability that is formalized in Coq.
- Section 6 discusses type safety for Wyv_{core} and its variants.
- Section 7 discusses the decidable variants Wyv_{self} and Wyv_{fix} in the context DOT and Scala.
- Section 8 concludes, discussing the potential for future work.

This paper makes the following contributions:

- A catalog of the factors involved in defining a finite subtype algorithm for a type system with both recursive types and path dependent types.
- An approach to attaining subtype decidability for path dependent types by restricting recursive types while retaining key forms of expressiveness.
- An approach to attaining subtype decidability for path dependent types by removing environment narrowing and a discussion of the trade offs associated with this.
- Subtype Decidability for the both of the above formalisms is formalized in Coq [Anon. 2019b].

2 BACKGROUND

Path dependent types are a language feature that provides a range of expressiveness including parametric polymorphism and nominality. They are most notably featured as a core part of Scala. The strength of path dependent types is the economy of concepts they afford in the presence of several other key language features, specifically intersection and union types and recursive types. In this section we motivate path dependent types and touch on some key concepts that have implications for discussions later in this paper.

2.1 Type Refinements

A type refinement is a mechanism for constructing a more specific version of an existing type via structural refinement. Amin et al. featured them as part of their formalism of path dependent

types. A type refinement captures a common pattern that programmers write when constructing types. The following example demonstrates the use of type refinements in the construction of an `AppendableList` type from a `List` type.

```
1 type List = {def head : Object
2           def tail : List}
3 type AppendableList = List{def append (o : Object) : List}
```

`AppendableList` represents the type of `Lists` that also contain an `append` function. This implies some useful expressiveness: the type extension implied by Java subclassing can be captured using type refinements, along with parametric polymorphism.

2.2 Parametric Polymorphism

Path dependent types completely subsume the polymorphism of System F [Reynolds 1974] and System F_<: [Pierce 1992] and of object oriented languages such as Java. Instead of parameterizing terms with types, types are wrapped as type members of carrier objects. In the example below we translate bounded polymorphism to a language containing path dependent types. It is subsequently fairly simple to sugar the translation back into something resembling traditional bounded polymorphism.

<pre>1 def append[E >: ⊥] 2 (e : E, 3 l : List[E])</pre>	⇒	<pre>1 def append(x : {type E >: ⊥}, 2 e : x.E, 3 l : List{type E = x.E})</pre>
--	---	---

While path dependent types encompasses the expressiveness of parametric polymorphism, the reverse is not true: path dependent types provide increased expressiveness over parametric polymorphism. Path dependent types are implicitly included as part of an object and can be referenced from outside of the containing object. Consider the following shallow copy function.

```
1 type List = {type E <: T}
2 def copy(l : List) : List{type E = l.E}{
3   new List{type E = l.E}
4 }
```

If we were to write the above function using in a language with bounded polymorphism, it would have to include as an argument the element type of the copied list explicitly. There is no way to write generic functions that don't explicitly require the generic type to be mentioned at every step. While this is perhaps a small savings in the above example, requiring explicit parameters can lead to potentially excessive boilerplate if more types are involved.

2.3 Modules

One of the original motivations for including type members in Scala was for the purposes of modularity, echoing the module system of Standard ML [Milner et al. 1997]. Path dependent types can be used to express module signatures such as the `Protocol` signature, here adapted from the FoxNet project which implemented a TCP/IP stack in Standard ML [Biagioni et al. 2001]:

```
1 type Protocol {
2   type Address
3   type Data
4   def send(a:Address, d:Data)
```

```

5      def receive(a:Address):Data
6  }
7
8  module IP : Protocol { ... }
9  module TCP : Protocol { ... }

```

Here `Address` and `Data` are abstract data types representing addresses and data in a particular protocol implementation. Note: while they are not explicitly defined with bounds, type `Address` is implicitly the upper bounded type definition, type `Address <: T`. We show the first version of `Protocol` from the FoxNet paper; the final version is more complicated, but conceptually similar. Different protocols (e.g. TCP, IP, Ethernet...) may represent these types in different ways, so these types are members of their surrounding module. Scala (and Wyvern) add a bit of flexibility by modeling modules as first-class objects, which in this example would allow custom network stacks to be constructed at run time, but otherwise support similar kinds of expressiveness.

2.4 Nominality

The ability to define path dependent types abstractly, combined with bounds on type members and the ability to provide type refinements, supports a kind of nominal subtyping. In the previous example, `Data` and `Address` are defined with only an upper bound. This means that in practice, they exhibit nominal subtyping. The only way to subtype a type without an explicit lower bound (i.e. a type with a lower bound of \perp) is through type refinement. The following example due to Rompf and Amin demonstrates how nominality arises in DOT.

```

1  type ListAPI = {
2      type List <: { type Elem }
3      def nil(): List{ type Elem >: Bot }
4      def cons(x : {type E},
5              e : x.E ,
6              l : List{type Elem <: x.E} ): List{type Elem <: x.E}
7  }

```

`List` is abstract, meaning it has an (implicit) lower bound of \perp . We give it an upper bound expressing that every `List` has a type member `Elem`. The only way to subtype `List` is as a refinement on `List`.¹ Thus, until a concrete form of `List` is supplied, subtyping of `List` is essentially nominal. In the rest of this paper, when referring to “abstract types”, we mean types that are upper bounded (with lower bound \perp), and only partial type information (or no type information) supplied. This, is almost always in relation to the implications for nominality. The nature of nominality that is provided by path dependent types is of particular concern to the work in this paper. We cover this in more detail in Section 3.4.2.

2.5 Dependent Object Types

The Dependent Object Types (DOT) calculus [Amin 2016; Amin et al. 2016, 2012, 2014; Rompf and Amin 2016] is a core calculus that is intended to be the theoretical basis for Dotty [dot 2019], itself a basis for the upcoming Scala 3. In DOT, a broad array of expressiveness arises from the core features of path dependent, intersection, recursive and dependent function types. Path dependent types encode nominal subtyping and Scala modules. When combined with dependent function types, path dependent types subsumed bounded polymorphism. Intersection types capture the

¹This form of refinement is a special case of *intersection types*.

type aspects of multiple inheritance, and when coupled with recursive types give rise to structural subtyping.

3 WYVERN

Wyvern [Nistor et al. 2013] is an object oriented language based on many of the same ideas explored in DOT. Like Scala, Wyvern incorporates several language features derived from both functional and object oriented paradigms. In this section we present a core calculus for Wyvern: W_{YV}^{core} . We subsequently discuss the issues with constructing a terminating subtyping algorithm for W_{YV}^{core} , and provide context for the variants of W_{YV}^{core} in the following sections.

Like DOT, W_{YV}^{core} features path dependent types, structural subtyping, first class functions and objects and recursive types. W_{YV}^{core} is, however, simpler than DOT in a few key ways. Intersection and union types introduce several kinds of complexity that are not directly related to the complexity of path dependent types. For this reason, W_{YV}^{core} introduces a simplified form of intersection types, type refinements—an idea that was, in fact, present in an early version of DOT that was presented at the FOOL workshop [Amin et al. 2012].

We have already touched on type refinements, but we will say some more here as they form a key component of W_{YV}^{core} . A type refinement ($\tau\{z \Rightarrow \bar{\sigma}\}$) represents a more specific form of intersection type that explicitly introduces a recursive definition. The variable z in the type represents the object whose type is being defined. The member types within the refinement ($\bar{\sigma}$) are restricted to only new members that do not occur in the base type (τ), or member types that are subtypes of already existing members within the base type.

3.1 W_{YV}^{core}

We now present the structure of types and subtyping in the W_{YV}^{core} calculus; we defer the term syntax and typing rules to section 6. The syntax of W_{YV}^{core} types is provided in Figure 1. A **Type** is either the top type (\top), the bottom type (\perp), a type member selection on a variable ($x.L$), a refinement on \top ($\top\{z \Rightarrow \bar{\sigma}\}$), a refinement on a selection type ($x.L\{z \Rightarrow \bar{\sigma}\}$), or a universally quantified type ($\forall(x : \tau_1).\tau_2$). A type refinement $\tau\{z \Rightarrow \bar{\sigma}\}$ is a base type τ refined with a set of declaration types $\bar{\sigma}$ and a self reference z . Universally quantified types ($\forall(x : \tau_1).\tau_2$) type dependent functions, where the return type is dependent on the argument. **Declaration Types** are the types assigned to declarations within objects. Unlike in DOT, these types are distinct from the types of terms in that they can only be used as part of a type refinement. DOT is able to conflate the two kinds of types due to the modeling of record types with intersection types. A declaration type is either a type member declaration that is upper bounded ($L \geq \tau$), lower bounded ($L \leq \tau$), an exact type ($L = \tau$) or a value declaration ($l : \tau$). There are several places in the semantics where rules may apply to multiple syntactic forms of declaration types. An example is E-UPPER in Figure 4, in this case, the syntax $L \leq/\geq \tau$ is used to imply that the rule applies in both cases.

We also define term ($\Gamma \vdash x \ni \sigma$) and type ($\Gamma \vdash \sigma \in^z \tau$) membership in Figure 3. Intuitively, these capture object membership within an environment constructed during subtyping (or typing). An object indicated by variable x contains a member of type σ if $\Gamma \vdash x \ni \sigma$, and a type τ is inhabited by objects that contain members of type σ (where z is the self variable) if $\Gamma \vdash \sigma \in^z \tau$. A declaration type (σ) is a member of a type (τ) if τ is a type refinement, and σ occurs as part of that refinement (M-REFN-1) or is an unrefined member of the base type (M-REFN-2). If τ is some selection type $x.L$ and σ is a member of its upper bound, then σ is also a member of $x.L$ (M-UPPER). A declaration type is considered a member of a variable (or rather the object referenced by the variable) if the type of the variable contains the member (M-VAR).

Type extension ($\Gamma \vdash \tau_1 \leq:: \tau_2$) is defined in Figure 4, as is an important judgment in constructing transitive subtyping for type refinements. A type extends its upper bound (determined using

$\tau ::=$	Type
\top	top
\perp	bottom
$x.L$	type selection
$\top\{z \Rightarrow \bar{\sigma}\}$	top refinement
$x.L\{z \Rightarrow \bar{\sigma}\}$	selection refinement
$\forall(x : \tau). \tau$	all
$\sigma ::=$	Declaration Type
$L \leq \tau$	upper bound
$L \geq \tau$	lower bound
$L = \tau$	equality
$l : \tau$	value
$\Gamma ::=$	Context
\emptyset	
$\Gamma, x : \tau$	

Fig. 1. W_{yVcore} Syntax

$\Gamma \vdash \top$ is extendable
$\Gamma \vdash x \ni L \leq \tau \quad \Gamma \vdash \tau$ is extendable
$\Gamma \vdash x.L$ is extendable
$\Gamma \vdash \tau$ is extendable
$\Gamma \vdash \tau\{z \Rightarrow \bar{\sigma}\}$ is extendable

Fig. 2. W_{yVcore} Type Extendability

$\Gamma \vdash \sigma \in^z \Gamma(x)$	(M-VAR)	$\sigma \in \bar{\sigma}$	(M-REFN-1)
$\Gamma \vdash x \ni [x/z]\sigma$		$\Gamma \vdash \sigma \in^z \tau\{z \Rightarrow \bar{\sigma}\}$	
$id(\sigma) \notin \{id(\sigma') : \sigma' \in \bar{\sigma}\}$		$\Gamma \vdash \sigma \in^z \tau$	(M-REFN-2)
$\Gamma \vdash \sigma \in^z \tau\{z \Rightarrow \bar{\sigma}\}$			
$\Gamma \vdash x \ni L \leq \tau$		$\Gamma \vdash \sigma \in^z \tau$	(M-UPPER)
$\Gamma \vdash \sigma \in^z x.L$			

Fig. 3. W_{yVcore} Membership

$\Gamma \vdash x \ni L \leq \tau$	$\Gamma \vdash \tau$ is extendable	(E-UPPER)
$\Gamma \vdash x.L \leq \tau$		
$\Gamma \vdash \tau \leq \tau'$		(E-REFINE)
$\Gamma \vdash \tau\{z \Rightarrow \bar{\sigma}\} \leq \tau'$	$flat(\tau', \bar{\sigma}, z)$	

Fig. 4. W_{yVcore} Type Extension

$flat(\top, \bar{\sigma}, z)$	$= \top\{z \Rightarrow \bar{\sigma}\}$
$flat(x.L, \bar{\sigma}, z)$	$= x.L\{z \Rightarrow \bar{\sigma}\}$
$flat(\tau\{z \Rightarrow \bar{\sigma}_1\}, \bar{\sigma}_2, z)$	$= \tau\{z \Rightarrow merge(\bar{\sigma}_1, \bar{\sigma}_2)\}$

Fig. 5. Refinement Flattening

$merge(\emptyset, \bar{\sigma})$	$= \bar{\sigma}$	$id(L \leq _) = L$
$merge(\sigma : \bar{\sigma}_1, \bar{\sigma}_2)$	$= \text{if } id(\sigma) \in map(id, \bar{\sigma}_2) \text{ then } merge(\bar{\sigma}_1, \bar{\sigma}_2)$	$id(L \leq _) = L$
	$\text{else } \sigma : merge(\bar{\sigma}_1, \bar{\sigma}_2)$	$id(L = _) = L$
		$id(l : _) = l$

Fig. 6. Associated Functions

membership) if it is a selection type (E-UPPER) or in the case of type refinements, a refined version of any type that the base type extends (E-REFINE). Figure 2 restricts extension to either \top , selection types or type refinements. Related calculi, such as DOT, do not generally have a type extension judgment, as the semantics defined by it are included as part of the subtyping of intersection types. In W_{yVcore} , we separate type extension out from subtyping, as type refinements are the central source of recursion during subtyping. Separating their definition out, allows for their semantics to be changed independently of subtyping.

The associated *flat* and *merge* functions are defined in Figure 5, and are used to capture type extension for type refinements. $flat(\tau, \bar{\sigma}, z)$ flattens the declaration types $\bar{\sigma}$ with self variable z into type τ . Normally this would simply produce the type $\tau\{z \Rightarrow \bar{\sigma}\}$, however in the case where τ is itself a type refinement, the two refinements are merged in order to produce a syntactically valid form.

$$\begin{array}{c}
\boxed{\Gamma \vdash \tau_1 <: \tau_2, \bar{\sigma}_1 <: \bar{\sigma}_2, \sigma_1 <: \sigma_2} \\
\\
\Gamma \vdash x.L <: x.L \quad (\text{S-REFL}) \qquad \Gamma \vdash \perp <: \tau \quad (\text{S-BOTTOM}) \qquad \Gamma \vdash \tau <: \top \quad (\text{S-TOP}) \\
\\
\frac{\Gamma \vdash x \ni L \leqslant \tau' \quad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash x.L <: \tau} \quad (\text{S-UPPER}) \qquad \frac{\Gamma \vdash x \ni L \geqslant \tau' \quad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash \tau <: x.L} \quad (\text{S-LOWER}) \\
\\
\frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma, x : \tau_2 \vdash \tau'_1 <: \tau'_2}{\Gamma \vdash \forall(x : \tau_1). \tau'_1 <: \forall(x : \tau_2). \tau'_2} \quad (\text{S-ALL}) \qquad \frac{\Gamma, z : \tau \{z \Rightarrow \bar{\sigma}_1\} \vdash \bar{\sigma}_1 <: \bar{\sigma}_2}{\Gamma \vdash \tau \{z \Rightarrow \bar{\sigma}_1\} <: \tau \{z \Rightarrow \bar{\sigma}_2\}} \quad (\text{S-REFINE}) \\
\\
\frac{\Gamma \vdash \tau_1 \leqslant \tau \quad \Gamma \vdash \tau <: \tau_2}{\Gamma \vdash \tau_1 <: \tau_2} \quad (\text{S-EXTEND}) \qquad \frac{\forall \sigma_2 \in \bar{\sigma}_2, \exists \sigma_1 \in \bar{\sigma}_1 \text{ s.t. } \Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash \bar{\sigma}_1 <: \bar{\sigma}_2} \quad (\text{S-DECLS}) \\
\\
\frac{\Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash L \leqslant \tau_1 <: L \leqslant \tau_2} \quad (\text{S-DECL-UPPER}) \qquad \frac{\Gamma \vdash \tau_2 <: \tau_1}{\Gamma \vdash L \geqslant \tau_1 <: L \geqslant \tau_2} \quad (\text{S-DECL-LOWER}) \\
\\
\frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash L = \tau_1 <: L = \tau_2} \quad (\text{S-DECL-EQUAL}) \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash l : \tau_1 <: l : \tau_2} \quad (\text{S-DECL-VAL})
\end{array}$$

Fig. 7. Wvy_{core} Subtyping

The subtype semantics for Wvy_{core} are defined in Figure 7. Subtyping is explicitly reflexive only with regard to selection types on variables (S-REFL). Subtyping is bounded above by \top (S-TOP) and below by \perp (S-BOTTOM). A selection type subtypes those types above its upper bound and super types those types below its lower bound (S-UPPER and S-LOWER respectively). The type $L = \tau$ is in fact expressing τ as both the upper bound and the lower bound and thus for brevity's sake, S-UPPER and S-LOWER both apply to types of the form $L = \tau$. This is captured in the rules using $L \leqslant \tau$ and $L \geqslant \tau$. Subtyping of universally quantified types is defined in S-ALL, type bounds are required to have a contra-variant relationship while the type bodies are required to have a covariant relationship. Type refinement subtyping is relatively complex and is captured in the S-REFINE and S-EXTEND rules. S-REFINE compares two refinements on the same type. S-EXTEND is essentially an upper bound lookup that takes refinements into account. Finally, subtyping of member types is done per member type (S-DECLS, S-DECL-UPPER, S-DECL-LOWER, S-DECL-EQUAL and S-DECL-VAL).

3.2 Undecidability of Subtyping in Wvy_{core}

Subtyping in Wvy_{core} is undecidable. This can be demonstrated by its encoding of System $F_{<}$ and the fact that this extension is a conservative one. Undecidability in System $F_{<}$ is primarily the result of a combination of two features: environment narrowing and contra-variance.

3.2.1 Encoding System $F_{<}$ in Wyvern. Subtyping in System $F_{<}$ is subsumed by subtyping of Wvy_{core} . This is easily demonstrated by the encoding in Figure 8 using a combination of type members, structural subtyping and universally quantified types. Such an encoding is simple, and to be expected given the similarity of dependent function types in Wvy_{core} and the polymorphic functions of System $F_{<}$. What might be surprising is that System $F_{<}$ subtyping is subsumed by Wyvern, even in the absence of dependent function types. Figure 9 provides an encoding of the critical aspects of System $F_{<}$ using only selection types and structural types with recursive types. Note: the reader should not attempt to draw any intuition from this encoding other than what is stated here. The encoding of Figure 9 demonstrates how Wvy_{core} captures the problematic aspects

of subtyping in System $F_{<}$ (contra-variance, recursion and divergent contexts), and not necessarily the semantics of System $F_{<}$. Indeed, bounded-quantification in System $F_{<}$ is used to type functions, while its encoding in Figure 9 is an object type. The salient point of Figure 9 is that the $W_{\text{yvern}}^{\text{core}}$ encodes the undecidable fragment of System $F_{<}$ (bounded-quantification), and thus transitively Turing machines. Put another way, a decision procedure for subtyping in $W_{\text{yvern}}^{\text{core}}$ constitutes a decision procedure for subtyping in System $F_{<}$, and thus a general solution to the halting problem. The proof that $W_{\text{yvern}}^{\text{core}}$ represents not just an extension, but a conservative extension of System $F_{<}$ is provided as part of the associated technical report [Anon. 2019a].

$$\begin{array}{ll} F(\top) & = \top \\ F(\alpha) & = x.A \\ F(T_1 \rightarrow T_2) & = \forall(x : F(T_1)).F(T_2) \\ F(\forall(\alpha \leq T_1).T_2) & = \forall(x : \{ A \leq F(T_1) \}).F(T_2) \end{array}$$

Fig. 8. Encoding $F_{<}$ in $W_{\text{yvern}}^{\text{core}}$

$$\begin{array}{ll} F'(\alpha) & = z.A \\ F'(\forall(\alpha \leq T_1).T_2) & = \neg\top \left\{ \begin{array}{l} z \\ A \leq F(T_1) \\ B \geq F(T_2) \end{array} \right\} \\ \text{where } \neg\tau & = \top \{ z \Rightarrow L \geq \tau \} \end{array}$$

Fig. 9. Encoding $F_{<}$ in $W_{\text{yvern}}^{\text{core}}$ without All Types

Cardelli and Wegner defined a subset of System $F_{<}$ called Kernel $F_{<}$ that restricts subtyping of bounded quantification to invariance on type parameters [Cardelli and Wegner 1985]. Pierce provides a proof of decidability for Kernel $F_{<}$ [Pierce 2002]. Intuitively we might be inclined to propose a similar restriction on Wyvern and enforce invariance on the type parameters of universally quantified types in Wyvern, however this would not prevent the encoding of System $F_{<}$ using recursive types (Figure 9). A similar restriction here would perhaps enforce invariance on lower bounded type definitions ($L \geq \tau$), however this would exclude much of the expressiveness of lower bounds, in particular it would have severe consequences for the ability to write to polymorphic data structures.

Castagna and Pierce proposed a decidable variant of bounded quantification, $F_{<}^{\top}$ [Castagna and Pierce 1994], that introduced a modified subtype rule for universally quantified types, allowing for variant type bounds while constraining additions to the context to \top . We provide this rule in Figure 10. Unfortunately, $F_{<}^{\top}$ was ultimately found to lack minimal typing [Castagna et al. 1994]

$$\frac{\Gamma \vdash S_2 <: S_1 \quad \Gamma, \alpha \leq \top \vdash U_1 <: U_2}{\Gamma \vdash (\forall(\alpha \leq S_2).U_2) <: (\forall(\alpha \leq S_1).U_2)} \quad (\text{S-ALL})$$

$$\forall(x : \{\text{type } E\}). \forall(e : x.E). \forall(l : \text{List}\{\text{type } \text{Elem} \leq x.E\}). \text{List}\{\text{type } \text{Elem} \leq x.E\}$$

Fig. 10. Subtyping of Bounded Quantification in $F_{<}^{\top}$.

² While this is a drawback, $F_{<}^{\top}$ might still inform the search for a decidable variant of Wyvern. Unfortunately, $F_{<}^{\top}$ is even less amenable to adaptation to $W_{\text{yvern}}^{\text{core}}$ than Kernel $F_{<}$. This can be seen by observing the restriction implied by the encoding of Figure 9. The subtype rule of Figure 10 excludes the possibility for context divergence during subtyping by requiring all bounds to be treated as \top during subtyping. Adapting this restriction to $W_{\text{yvern}}^{\text{core}}$ would require stripping all type information from type members during subtyping by treating their bounds to \top (or \perp). Such a restriction would prohibit many useful instances of expressiveness from $W_{\text{yvern}}^{\text{core}}$, not least of all the nominality afforded by abstract type members.

²A discussion on this topic in the form of an email exchange is included as a later version of the original paper on the author's homepage.

3.3 Recursive Types, Contra-variance & Environment Narrowing

The problems for decidability in $W_{\text{YV}}^{\text{core}}$ derive from the complex mix of *Recursive Types*, *Subtype Contra-variance* and *Environment Narrowing*. The first two provide key aspects of expressiveness in $W_{\text{YV}}^{\text{core}}$, while the last fundamentally underpins much of the theoretical properties of languages such as $W_{\text{YV}}^{\text{core}}$ and DOT. Recursive types are types that provide a way for types to refer to themselves recursively. Recursive types in $W_{\text{YV}}^{\text{core}}$ and DOT resemble equi-recursive types, however they are quantified over terms and not types. Contra-variance is a subtyping property that requires subtyping in certain instances enforce an inverse relationship between the components of a type. Environment narrowing is a property of DOT-like languages where types within contexts can be changed during type checking to a more specific, “narrower” type. All three are interconnected. In $W_{\text{YV}}^{\text{core}}$, type refinements provide both environment narrowing and recursive types. The central vehicle for narrowing in the the subtype rules of Figure 7 is S-REFINE, subtyping of recursive types. Contra-variant subtyping (exhibited in both subtyping of dependent function types and lower bounds of type members) allows for narrowing to occur on both sides of a subtype derivation. All three collectively contribute to issues of non-termination during subtyping. In this section we describe these three features, try to identify how they connect to each other, and what they each mean for expressiveness.

3.3.1 Recursive Types. Removing recursive types from $W_{\text{YV}}^{\text{core}}$ would reduce the issues of decidability to those of System $F_{<}$: as the only other source of environment narrowing is S-ALL. The wholesale sacrifice of recursive types removes several instances of valuable expressiveness, the most important of which is the critical encoding of certain forms of polymorphic object types and family polymorphism [Ernst 2001].

Languages such as Java, C# and Scala all provide a mechanism to construct new types from existing ones while instantiating generic types using self references to generic types of the new type. Below is an example modeling a Node type in a graph as a map from edges to other Nodes.

```

1 { self0 => type Map <: T{ type K <: T, type V <: T }
2     type Node <: Map{ self1 =>
3         type E <: T
4         type Value <: T
5         type K = self1.E
6         type V = self0.Node }}
```

Intrinsic to this encoding is the ability to refer to the current type (self₁.E and self₀.Node), i.e. recursive types.

Family polymorphism refers to a type pattern where an entire family of types can be defined and used polymorphically. The canonical example from Ernst is the family structure of Graph similar to the example below.

```

1 type Graph = { self =>
2     type Node <: T{
3         val neighbors : Map{ type K = self.Edge
4                             type V = self.Node }
5     }
6     type Edge <: T{val origin, destination : self.Node}}
```

A graph is defined with a family of types, Node and Edge that are defined mutually. Specific instantiations of Graph must maintain these relationships. The above set of types are entirely dependent

on the existence of recursive types since every use of Node or Edge is in fact a selection on the implicit self variable of Graph.

Evidently recursive types are integral to the expressiveness of W_{core} . In fact without recursive types, no recursive data structures can be defined within W_{core} !

3.3.2 Contra-variant Subtyping. The most common example of contra-variance is subtyping of arrow types in the typed lambda calculus that require argument types to maintain an inverse relationship to the subtyping of the arrow type; this is captured in our S-ALL rule. In W_{core} lower bounds are contra-variant with regard to the member that refers to them. Variant lower bounds are often used to model writable datatypes. The example below defines a polymorphic append function.

```

1 def append[E >: ⊥](e : E, l : List[E]) : List[E] =
2   match l with
3     nil = e :: nil
4     e'::t = e'::(append e t)

```

Defining E as lower bounded allows the body of append to be correctly typed as List[E]. Since $E >: \perp$ in fact implies a lower bounded type member, any use (append[Int](...)) requires a variance on the lower bound. Completely removing contra-variant subtyping would exclude an entire class of useful examples.

3.3.3 Environment Narrowing. While environment narrowing is harder to point to with explicit examples, it underpins most aspects of typing in a type system with path dependent types. As has already been discussed, it is necessary in the subtyping of both universally quantified and recursive types (S-ALL and S-REFINE in Figure 7). Environment narrowing is fundamentally intertwined with the derivation of subtype transitivity as demonstrated by the long struggle for type safety for DOT. Environments are narrowed during subject reduction of functions, requiring well-formedness to be maintained in the presence of a narrower type. This last instance in particular, with the help of intersection types, can result in strange uninhabitable types being constructed dynamically, throwing many notions of well-formedness into doubt.

3.4 Cyclic Type Definitions

The ability for types in W_{core} to be explicitly cyclic in definition further complicates the derivation of a finite subtyping algorithm. The most obvious form of this is direct circularity ($\{z \Rightarrow L \leq z.L\}$), but this is a nonsensical type definition and can easily be avoided using cycle detection. More complex and useful forms of type definitions do rely on circularity. Below we borrow an example from [Greenman et al.](#).

```

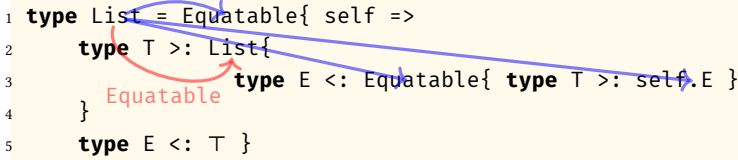
1 { self0 =>
2   type Equatable = T{ self => type T >: ⊥, def equals : self.T -> bool }
3   type List = Equatable{ self =>
4     type T >: self0.List{ type E <: self0.Equatable{ type T >: self.E }}
5     type E <: T
6   }
7   type Tree = self0.List{ type E <: self0.Tree }
8 }

```

List is defined polymorphically as Equatable to lists that contain elements Equatable to the generic type of the current list. Tree is then modeled as a List of element type Tree. While each

definition can be justified, it leads to instances of cycles during subtype derivation. Specifically the question of `Tree <: Equatable[Tree]` requires itself as a subproof. A Java formulation of this example results in a stack overflow error on compilation. Greenman et al. make use of a syntactic separation (called the Material/Shape separation) on Java types that distinguishes between concrete *Material* types such as `Tree` and abstract *Shape* types, such as `Equatable`, that facilitate cyclic definitions. They demonstrated that this ensured decidable subtyping for Java, and resulted in an equivalent Java version of the above example being rejected. They also did an empirical study showing that the Material/Shape separation discipline is already followed in essentially all existing Java code.

Separation of types is based on cycles constructed by dependencies between types. In the previous example, `List` is dependent on both `Equatable` and itself. There is a cycle formed from `List` to itself, facilitated by `Equatable`. Below we draw these dependencies.



```

1 type List = Equatable{ self =>
2   type T >: List{
3     Equatable type E <: Equatable{ type T >: self.E }
4   }
5   type E <: T }

```

The problematic dependency is captured by the `List` →^{Equatable} `List` edge. The definition of a Shape is the set of type names that label edges that, when removed, result in a cycle-free type graph. This may seem an opaque definition, but it has the effect that all type cycles contain at least one Shape. That is, for any type, there is a maximal “shape depth”. In Java, this property coupled with the prohibition of shapes from use in type parameters, provides the key property that for any subtype comparison in Material/Shape separated Java, there is a maximal depth beyond which shapes do not occur. As shapes are by definition required for recursive type definitions, subtyping is guaranteed to terminate.

3.4.1 Designing Material/Shape Separated $W_{yv_{core}}$. We design a Material/Shape separation for $W_{yv_{core}}$ based on that of Greenman et al.. We start first by defining a separation on type names in $W_{yv_{core}}$ in Figure 11. Next we identify a conceptual mapping from Material/Shape separated Java

L	$::=$	Type Label
M		<i>Material</i>
S		<i>Shape</i>

Fig. 11. Material/Shape Separation on Type Names

to $W_{yv_{core}}$. At first glance this seems like a relatively simple task, type definitions play the role of class definitions in Java, while type members also assume the role of generic type parameters. The complexities of path dependent types however make it quite difficult to separate types in such a way as to both imitate the separation in Java and ensure decidable subtyping.

At the center of the complexity is the economy of concepts afforded by languages such as Wyvern and DOT. Path dependent types are used to model several different aspects of modern programming languages: bounded polymorphism, Java generics style parametric polymorphism and nominality. The expression of concepts by a common syntax conflates any separation on one concept with a separation on another. The Material/Shape separation on Java uses qualities of the nominal type hierarchy (modeled in $W_{yv_{core}}$ with type members) to enforce a syntactic prohibition on usages of Shapes in generic type parameters (also modeled in $W_{yv_{core}}$ with type members).

$\sigma ::=$ **Decl. Type**
 \vdots
 $L \leq \tau$

Fig. 12. Nominal Syntax for W_{yv_core}

$$\frac{\Gamma \vdash x \ni M \leq \tau}{\Gamma \vdash x.M \leq:: \tau} \quad (\text{E-MAT}) \qquad \frac{\Gamma \vdash x \ni S \leq \tau \quad \tau = x'.S'\{z \Rightarrow \bar{\sigma}\}}{\Gamma \vdash x.S \leq:: \tau} \quad (\text{E-SHAPE})$$
Fig. 13. Nominal Extension Rules for W_{yv_core}

Further, the subtype decidability argument of Material/Shape separated Java is dependent on the strict nominal nature of the Java type hierarchy, while subtyping with bounded type members always allows subtyping to be derived via structural subtyping.

The conceptual conflation is most apparent when attempting to directly apply the prohibition from Java of Shapes to generic types. As has been noted, generic types are modeled in W_{yv_core} by type members, thus it would seem natural to prohibit Shapes from use in type members. But we have also noted that the type hierarchies and type dependencies that are used to define Shapes are also derived from type members. A prohibition on Shapes in type member definitions would prohibit the very mechanism used to identify Shapes, and ultimately all type cycles.

3.4.2 Nominality with Path Dependent Types. The tension between these two concepts (nominality and polymorphism) would seem to be alleviated by separating them within the syntax. First we seek to understand nominality in W_{yv_core} . Section 2.4 provided an example of how nominality arises in languages such as W_{yv_core} and DOT, and we will refer to that example here.

Nominality in W_{yv_core} or DOT does not constrain the type hierarchy in the same manner as the class hierarchy of a language such as Java. Super types of `List` have no constraints except the structure of `List`. This has implications for a separation on types. Part of the decidability argument of the Material/Shape separation in Java is that Shapes do not give rise to Materials through extension, that is a Shape cannot extend a Material. In W_{yv_core} this is not the case, it is always possible to use the upper bound of an abstract type for subtyping purposes, and it is not possible for every Shape to only extend other Shapes. At some point a Shape must be defined as an extension on a Material (most likely \top).

Nominality in W_{yv_core} is also only relevant to abstractly defined types ($L \leq \tau$). Lower bounded types ($L \geq \tau$ and $L = \tau$) cannot rely on such a hierarchy since it is possible to subtype a use of that type by subtyping the lower bound. Since the separation of Shapes from Materials relies on such hierarchy, this implies that lower bounded types may not use Shapes at all. Thus, we would be inclined to prohibit the use of Shapes from all type definitions of the form $L \geq \tau$ or $L = \tau$. Such a restriction, however would make it impossible Shapes to ever be meaningfully used. While abstractly defining a type allows for implementation to be hidden, in order for it to be initialized, there must be some non-abstract, concrete definition of the type of the form $L = \tau$. As an example, the `List` definition of Section 2.4 is abstract, but at run-time, there must exist a concrete `ListAPI` that contains a specific implementation of `List`. If Shapes themselves must be abstract, then the only way to subtype them would be by refinement, which would necessitate a Shape usage in a concrete type definition ($L = \tau$), which as we have already seen must be prohibited. This contradiction arises from the conflation of patterns that type members provide. In order to allow concrete types to observe nominal subtyping, we introduce nominality in to the syntax.

3.4.3 A Syntactic Form for Nominality. We extend W_{yv_core} with two additions that address the nominality issues in W_{yv_core} : a syntactic form for nominality (Figure 12), and a semantic restriction on the extension of that nominal form (Figure 13). On top of the existing syntactic forms for type members, we also allow the definition $L \leq \tau$, a nominal form for concrete types. In $L \leq \tau$, we mix

τ ::=	Type	σ ::=	Decl. Type	μ ::=	Mat. Type	δ ::=	Mat. Decl. Type
$\mu\{z \Rightarrow \overline{\sigma}\}$		$M \leq \tau$		$\mu\{z \Rightarrow \overline{\delta}\}$		$M \leq \mu$	
$\tau\{z \Rightarrow \overline{\delta}\}$		$M \geq \mu$		$x.M$		$M \geq \mu$	
$x.L$		$M = \mu$		$\forall(x : \mu). \mu$		$M = \mu$	
$\forall(x : \tau). \tau$		$L \leq \tau$		\top		$L \leq \mu$	
\top		$l : \mu$		\perp		$l : \mu$	
\perp							

Fig. 14. Material/Shape Separated WYV_{core} Syntax

the semantics of concrete exact types ($L = \tau$) with that of abstract upper bounded types ($L \leq \tau$). Nominally defined type members can be thought of as exact type members that are not considered as having an explicit lower bound. As with abstract type members, the only way to subtype a type defined as $L \leq \tau$, is by type refinement, and the only way to super type it is by type extension (by S-EXTEND). This allows the familiar form of selection types, but a more restricted semantics that observes a traditional nominal type hierarchy. Extension of a nominally defined type (Figure 13) introduces a semantic change to WYV_{core} by only allowing extension of Shapes by other Shapes (E-SHAPE). While shapes can be defined using \top (a material), they cannot be arbitrarily super typed (in a structural manner), thus during subtyping Shapes can only give rise to other Shapes. This ensures that the Shapes in WYV_{core} can never be structurally inspected, and observe nominality not only from subtyping, but also from super typing.

4 MATERIAL/SHAPE SEPARATED WYV_{core}

We now introduce the Material/Shape separation on the nominal WYV_{core} . The syntax for Material/Shape separated Wyvern is defined in Figure 14. Types are divided into two categories, pure Materials (μ and δ) that contain no uses of Shapes and general unseparated types (τ and σ) that may contain either Shapes or Materials. Shapes are restricted in type definitions to only upper bounded abstract types and nominal type members, thus only to types that obey a nominal style type hierarchy. Shapes may only be refined by pure Material refinements, and refinements containing Shapes may only be placed on pure materials.

Material/Shape separated WYV_{core} is not itself decidable, but provides a common separation on types for the decidable variants presented in this paper. We define a common decidability argument on the type graphs discussed in Section 3.4 that subsequent decidable variants all target as an intermediate representation. To ensure decidability the following separation properties are statically enforced on type definitions before type checking.

SEPARATION PROPERTY 1. *All type names are either Materials or Shapes.*

SEPARATION PROPERTY 2. *All type cycles contain a Shape.*

SEPARATION PROPERTY 3. *Lower bounds do not contain Shapes*

SEPARATION PROPERTY 4. *All usages of Shapes are type refinements.*

These separation properties ensure several aspects of subtyping that aid in the derivation of subtype decidability: (i) There is a maximal depth on each type until a Shape is reached, (ii) new Shapes cannot be introduced on the right-hand side of subtyping, and (iii) Subtyping of a Shape removes all Shapes from the right-hand side.

The key intuition to be taken from the separation of Figure 14 and the 4 separation properties is related to the Material/Shape separation of Java. Material/Shape separated Java guarantees that

during subtype checking there is a maximal proof search depth, beyond which no Shapes occur. Material/Shape separated $W_{\text{YV}}^{\text{core}}$ guarantees that there is a maximal proof search depth, beyond which no Shape occurs on the *right hand side*. Since, by separation property 4, all cyclic type definitions must pass through a shape, subtype checking in $W_{\text{YV}}^{\text{core}}$ is assured to be bounded by a finite measure on the right hand side. Thus, the syntax of Figure 14 prohibits Shapes from those positions that would allow Shapes to be introduced on the right hand side during subtype checking, i.e. lower bounds. Further, the syntax of Figure 14 ensures that type refinements with a Shape base type ($\tau\{z \Rightarrow \bar{\delta}\}$) must be pure materials. The reader might note that parameter types of dependent function types allow for Shape, but are contra-variant, and thus allow for Shapes to be introduced on the right hand side during subtype checking. This is something that will be dealt with specifically by each variant on $W_{\text{YV}}^{\text{core}}$.

4.1 Type Graphs

Sections 5 and 6 provide proofs of subtype decidability for their respective subtype semantics. In this section we provide a subtype algorithm for a common target, type graphs. For the most part, type graphs mirror types, and in the absence of path dependent types, are equivalent to types. Types however do not include all the type information required during subtyping (type member definitions and type extension) as syntactic sub-components. A type graph is a graphical representation of a type that includes this relationship. We provide the full definition of the encoding of type graphs in the associated technical report, but provide the intuition here.

A type graph consists of nodes representing both types and declaration types, with directed edges representing dependencies between types. Edges exist from a type to all types that are syntactic sub-components it (e.g. $\forall(x : \tau_1).\tau_2$ has out-edges to both τ_1 and τ_2). Edges also exist between path dependent types and their definition. We provide the full formal rules for deriving the edges of type graphs (written as $\Gamma \vdash \tau \longrightarrow \tau'$) in the associated technical report, but provide some informal definitions below:

- The type graph of a type $\tau\{z \Rightarrow \bar{\sigma}\}$ in context Γ contains out-edges to each node representing each σ in $\bar{\sigma}$, along with an out-edge to the node representing type τ' , such that $\Gamma \vdash \tau\{z \Rightarrow \bar{\sigma}\} \leq:: \tau'$.
- The type graph of a selection type $x.L$ in context Γ contains an out-edge to the node representing its definition σ such that $\Gamma \vdash x.L \ni \sigma$ and $\text{id}(\sigma) = L$.
- The type graph of a dependent function type $\forall(x : \tau_1).\tau_2$ contains out edges to the nodes representing both τ_1 and τ_2 .

We have actually already discussed the construction of type graphs for type declarations and their edges. Recall the List/Equatable example:

```

1  type List = Equatable{ self0 =>
2    type T >: List{ self1 =>
3      Equatable type E <: Equatable{ self2 => type T >: self.E }
4    }
5    type E <: T }
```

Type refinements not only point to their component types, but also the type extension that they represent. In the above example, the type `List{ type E = Int }` depends not only on its syntactic subcomponents (`Int`), but also on its extended type:

`Equatable{self => type T >: List{type E <: Equatable{type T >: self.E}}, type E = Int}`

The Material/Shape separation defined in Section 4 ensures that all type cycles within a type graph contain at least one Shape. Thus, it follows that from any Material/Shape separated type graph, by splitting the graph at nodes representing Shapes, we are able to derive a forest of trees where terminal nodes represent either \top , \perp or a Shape $x.S$. While the full graph of `List` is fairly large (see associated technical report [Anon. 2019a]), we do provide a portion of the graph for `List` in Figure 15. While the definition of `List` and `Equatable` would be rejected by the Material/Shape

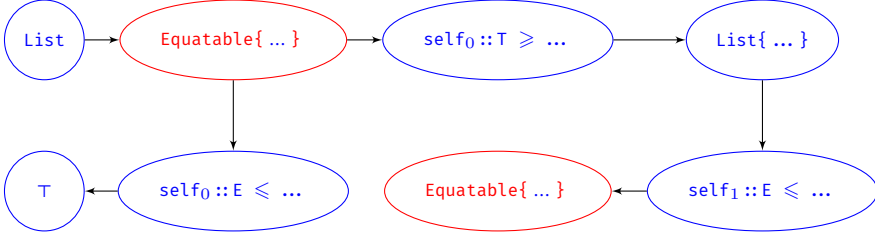


Fig. 15. A Partial Type Graph of `List`

separation due to the use of a Shape, `Equatable`, in the lower bound of `T`, the tree in Figure 15 helps illustrate how type trees are constructed. We now define a measure on type graphs in Figure 16. It is simple to see that since the measure in Figure 16 is a depth measure on Shapes, and every type cycle contains a Shape, $S(\tau)$ is finite for every appropriately separated type (and its type graph).

$$\begin{array}{ll}
 S(\top) & = 0 \\
 S(\perp) & = 0 \\
 S(x.S) & = 0 \\
 S(x.M) & = 1 + S(\text{definition of } x.M) \\
 S(\forall(x : T_1).T_2) & = 1 + \max(S(T_1), S(T_2)) \\
 S(T\{z \Rightarrow \overline{D}\}) & = \begin{array}{l} \text{if } T \text{ is a shape} \\ \text{then } 0 \\ \text{else } 1 + \max(S(T), S(\overline{D})) \end{array}
 \end{array}
 \qquad
 \begin{array}{ll}
 S(L \leq T) & = 1 + S(T) \\
 S(L \geq T) & = 1 + S(T) \\
 S(L = T) & = 1 + S(T) \\
 S(L \leq T) & = 1 + S(T) \\
 S(l : T) & = 1 + S(T)
 \end{array}$$

Fig. 16. Shape Depth Measure on Type Graphs

We state the subtyping algorithm for type graphs in Algorithm 1. We subsequently prove that for type graphs that obey the Material/Shape separation, all evaluations of `subtype` are guaranteed to terminate. This is stated in Theorem 4.1, along with a proof sketch. We provide the full proof in the associated technical report [Anon. 2019a].

THEOREM 4.1 (TYPE GRAPH DECIDABILITY). *Any call to Algorithm 1, `subtype`(T_1, T_2) will terminate for all inputs.*

PROOF. The proof is constructed in three parts:

- (1) All calls of the form `subtype`(S, M), where S is the type graph of a Shape (guaranteed to be a type refinement by Separation Property 4), and M is the type graph of a pure material type, are guaranteed to terminate.
- (2) All calls of the form `subtype`(T, M), where M is the type graph of a pure material type, are reducible to a finite set of calls of the form `subtype`(S, M'), and thus guaranteed to terminate.
- (3) All calls of the form `subtype`(T_1, T_2) are reducible to a finite set of calls of the form `subtype`(T, M), and thus guaranteed to terminate.

```

subtype( $T_1, T_2$ )
  switch  $T_1$  do
    case  $\top$  do
      switch  $T_2$  do
        case  $\top$  do return true;
        case  $x.L$  do return subtype( $T_1$ , lower_bound( $x.L$ ));
        otherwise do return false;
      end
    end
    case  $\perp$  do return true;
    case  $x_1.L_1$  do
      switch  $T_2$  do
        case  $\top$  do return true;
        case  $x_2.L_2$  do return ( $x_2 == x_2 \wedge L_1 == L_2$ )  $\vee$  subtype(upper_bound( $x_1.L_1$ ),  $T_2$ )  $\vee$  subtype( $T_1$ , lower_bound( $x.L$ ));
        otherwise do return subtype(upper_bound( $x_1.L_1$ ),  $T_2$ );
      end
    end
    case  $\forall(x : S_1).U_1$  do
      switch  $T_2$  do
        case  $\top$  do return true;
        case  $x.L$  do return subtype( $T_1$ , lower_bound( $x.L$ ));
        case  $\forall(x : S_2).U_2$  do return (subtype( $S_2, S_1$ )  $\wedge$  subtype( $U_1, U_2$ ));
        otherwise do return false;
      end
    end
    case  $T'_1 \{z \Rightarrow \overline{D}_1\}$  do
      switch  $T_2$  do
        case  $\top$  do return true;
        case  $x.L$  do return subtype( $T_1$ , lower_bound( $x.L$ ));
        case  $T'_2 \{z \Rightarrow \overline{D}_2\}$  do
          if  $T'_1 == T'_2$  then
            return subtypeDecl( $\overline{D}_1, \overline{D}_2$ )
          else
            switch extends( $T_1, \overline{D}_1$ ) do
              case Some( $T$ ) do return subtype( $T, T_2$ );
              otherwise do return false;
            end
          end
        end
      end
    end
  end
end

subtypeDecl( $D_1, D_2$ )
  switch  $D_1, D_2$  do
    case  $L_1 \leq T_1, L_2 \leq T_2$  do
      if  $L_1 = L_2$  then subtype( $T_1, T_2$ );
      else return false;
    end
    case  $L_1 = T_1, L_2 \leq T_2$  do
      if  $L_1 = L_2$  then subtype( $T_1, T_2$ );
      else return false;
    end
    case  $L_1 \geq T_1, L_2 \geq T_2$  do
      if  $L_1 = L_2$  then subtype( $T_2, T_1$ );
      else return false;
    end
    case  $L_1 = T_1, L_2 \geq T_2$  do
      if  $L_1 = L_2$  then subtype( $T_2, T_1$ );
      else return false;
    end
    case  $L_1 = T_1, L_2 = T_2$  do
      if  $L_1 = L_2$  then subtype( $T_2, T_1$ )  $\wedge$  subtype( $T_1, T_2$ );
      else return false;
    end
    case  $L_1 \leq T_1, L_2 \leq T_2$  do
      if  $L_1 = L_2 \wedge T_1 = T_2$  then true;
      else return false;
    end
    case  $l_1 : T_1, l_2 : T_2$  do
      if  $l_1 = l_2$  then subtype( $T_1, T_2$ );
      else return false;
    end
  end
  otherwise do return false;
end

```

Algorithm 1: Wyn_{core} Type Graph Subtyping Algorithm

Case 1. By case analysis on M , it is easy to see that all subsequent calls arising from $\text{subtype}(S, M)$ also have material types on the right hand side (either M itself, or the lower bound of some path dependent type selection). Similarly, the only new types that may arise on the left hand side are types that are extended by S (also Shapes by Figure 13). Since cycles are defined by labeled edges connecting type definitions to refining declaration types, it follows that all extensions are acyclic, i.e. types may not transitively extend themselves. Thus, by an ordering on the sum of the extension depth of S and the Shape depth (S) of M , it is clear that each successive call is strictly smaller than the last.

Case 2.

Case 3.

□

5 RESTRICTING RECURSIVE TYPES

In this section we define a restriction on recursive types that ensures decidable subtyping. $W_{YV_{self}}$ is a variant of $W_{YV_{core}}$ that restricts the usage of recursive types from pure material types. Thus the Material/Shape separation also implies a separation on the use of recursive types. This does not mean that Material types may not use recursive types in their definition, only that recursive types are prohibited from anywhere Shapes are. The modified syntax and subtype semantics are defined in Figure 17. In effect this restricts recursive types from lower bounded ($L \geq \tau$) and exact type members ($L = \tau$). A further semantic restriction inspired by Kernel $F_{<}$: [Cardelli and Wegner 1985] is made on subtyping of universally quantified types to enforce invariance on function argument types.

$$\begin{array}{c}
 \mu ::= \text{Material Type} \\
 \vdots \\
 \mu\{\bar{\delta}\}
 \end{array}$$

$$\frac{\Gamma, x : \tau \vdash \tau_1 <: \tau_2}{\Gamma \vdash \forall(x : \tau). \tau_1 <: \forall(x : \tau). \tau_2} \quad (\text{S-ALL})$$

Fig. 17. $W_{YV_{self}}$ Syntactic/Semantic Extension

5.1 Subtype Decidability

We provide a sketch of the subtype decidability proof here. The full proof is provided as a formulation in Coq [Anon. 2019b].

THEOREM 5.1 (SUBTYPING IN $W_{YV_{self}}$ IS DECIDABLE). *For all Γ , τ_1 and τ_2 , there exists a finite algorithm subtype that returns true if and only if $\Gamma \vdash \tau_1 <: \tau_2$.*

PROOF. The proof is constructed in two parts:

- (i) Problems of the specific form $\Gamma \vdash \tau <: \mu$ are decidable.
- (ii) Problems of the general form $\Gamma \vdash \tau_1 <: \tau_2$ reduce to the more specific form $\Gamma \vdash \tau <: \mu$, and thus decidable.

Case 1 (Subtyping of materials in $W_{YV_{self}}$ is decidable). The first fact to be noted is that in problems of the form $\Gamma \vdash \tau <: \mu$, narrowing does not occur since recursive types may not occur on the right-hand side, and the rule S-ALL enforces invariance on the argument type. Thus, during subtyping,

ζ	::=	Java Type		
\perp		<i>bottom</i>	\pm	::= Variance
\top		<i>top</i>	$+$	<i>negative</i>
α		<i>variable</i>	$-$	<i>positive</i>
$C\langle\pm\zeta\rangle$		<i>class</i>		
CT	::=	Class Table		
\emptyset				
$C\langle\pm\zeta\rangle$	<::	$D\langle\pm\zeta\rangle; CT$		

Fig. 18. Java-like Syntax

$[\alpha \mapsto \pm_1/\pm_2\zeta] + \zeta'$	=	$+[\alpha \mapsto \pm_1/\pm_2\zeta]\zeta'$
$[\alpha \mapsto \pm_1/\pm_2\zeta] - \zeta'$	=	$-[\alpha \mapsto \pm_1^{-1}/\pm_2\zeta]\zeta'$
$[\alpha \mapsto \pm/\pm\zeta]\alpha$	=	ζ
$[\alpha \mapsto \neg/\pm\zeta]\alpha$	=	\perp
$[\alpha \mapsto +/\neg\zeta]\alpha$	=	\top
$[\alpha \mapsto \pm_1/\pm_2\zeta]C\langle\pm'\zeta'\rangle$	=	$C\langle[\alpha \mapsto \pm_1/\pm_2\zeta]\pm'\zeta'\rangle$ where $+^{-1} = -$ and $-^{-1} = +$

Fig. 19. Java-like Substitution

all types are homomorphic to a type graph constructed during cycle detection, and observe the Separation Properties of Section 3.4. Since the right-hand side of the subtype problem is a pure material type, no Shapes can be introduced on the right-hand side. Thus there exists a finite depth at which a Shape occurs on the left-hand side. If $\tau = x.S\{z \Rightarrow \dots\}$ for some x, S and z , then there can be no subsequent contra-variance because contra-variance would require a Shape on the right-hand side. Subtyping is thus bound by the right-hand side which in the absence of Shapes is necessarily finite.

Case 2 (Subtyping in Wyv_{self} is decidable). Given that narrowing does not affect the left-hand side, τ_1 and types that arise from it are homomorphic to some type graph constructed during cycle detection, and thus there is a finite depth at which a Shape occurs on the right-hand side. If τ_2 is a Material then Case 1 already applies, otherwise if τ_2 is a Shape, then by Separation Property 4, refinements on both sides only contain only pure materials, also fulfilling Case 1.

□

5.2 Expressiveness in Wyv_{self}

5.2.1 Polymorphism. As noted in Section 3.3, recursive types provide key expressiveness examples in using polymorphic types in type definitions and family polymorphism. The major drawback of restricting recursive types is that these examples are restricted. The prior example of defining Node using the Map type is not possible since it would require the use of a recursive type in the lower bound of Node. It is useful then that Wyv_{core} introduces the nominal form of type members that allows concrete types to be defined without defining a lower bound. Below we rewrite the example of Section 3.3.

```
1 type Node[E <: T, V <: T] ≤ Map[self.E, Node]
```

As the nominal definition of Node can not be subtyped by a lower bound, there does not need to be a restriction on recursive types. The same strategy can be used to encode the example of family polymorphism in a different way (since the original approach, with normal lower bounds, doesn't work with this restriction).

```
1 type Graph ≤ {type Node <: {val neighbors : Map[Edge, Node]}}
2 type Edge <: {val origin, destination : Node}}
```

5.2.2 Encoding Java in Wyv_{self} . It is notable that Material/Shape separated Java is encodable in Wyv_{self} . In the associated technical report we provide an encoding for a fragment of Java Generics in to Wyv_{self} [Anon. 2019a]. We also provide a proof that Java subtyping is subsumed by Wyv_{self}

$$\begin{array}{c}
\Gamma_1 \vdash \tau <: \top \dashv \Gamma_2 \quad (\text{S-TOP}) \qquad \Gamma_1 \vdash \perp <: \tau \dashv \Gamma_2 \quad (\text{S-BOTTOM}) \qquad \Gamma_1 \vdash x.L <: x.L \dashv \Gamma_2 \quad (\text{S-REFL}) \\
\\
\frac{\Gamma_1 \vdash x \ni L \leq / = \tau'}{\Gamma_1 \vdash \tau' <: \tau \dashv \Gamma_2} \quad (\text{S-UPPER}) \qquad \frac{\Gamma_2 \vdash x \ni L \geq / = \tau'}{\Gamma_1 \vdash \tau <: \tau' \dashv \Gamma_2} \quad (\text{S-LOWER}) \\
\\
\frac{\Gamma_2 \vdash \tau_2 <: \tau_1 \dashv \Gamma_1 \quad \Gamma_1, x : \tau_1 \vdash \tau'_1 <: \tau'_2 \dashv \Gamma_2, x : \tau_2}{\Gamma_1 \vdash \forall(x : \tau_1). \tau'_1 <: \forall(x : \tau_2). \tau'_2 \dashv \Gamma_2} \quad (\text{S-ALL}) \qquad \frac{\Gamma_1 \vdash \tau_1 \leq:: \tau \quad \Gamma_1 \vdash \tau <: \tau_2 \dashv \Gamma_2}{\Gamma_1 \vdash \tau_1 <: \tau_2 \dashv \Gamma_2} \quad (\text{S-EXTEND}) \\
\\
\frac{\Gamma_1, z : \tau \{z \Rightarrow \bar{\sigma}_1\} \vdash \bar{\sigma}_1 <: \bar{\sigma}_2 \dashv \Gamma_2, z : \tau \{z \Rightarrow \bar{\sigma}_2\}}{\Gamma_1 \vdash \tau \{z \Rightarrow \bar{\sigma}_1\} <: \tau \{z \Rightarrow \bar{\sigma}_2\} \dashv \Gamma_2} \quad (\text{S-REFINE})
\end{array}$$

Fig. 20. $W_{YV_{fix}}$ Subtyping

subtyping. Apart from the fact that we restrict type parameters to a single bound, this encoding mirrors that of [Greenman et al.](#). While restricting type parameters to a single bound is less expressive than Material/Shape separated Java, Java itself only allows a single bound. The absence of intersection types in $W_{YV_{core}}$ (and thus $W_{YV_{self}}$) also presents a loss of expressiveness over Java: $W_{YV_{core}}$ is unable to model the subtyping that arises from multiple inheritance in Java.

6 REMOVING ENVIRONMENT NARROWING

$W_{YV_{fix}}$ is a variant of $W_{YV_{core}}$ that removes environment narrowing entirely from the subtype semantics. $W_{YV_{fix}}$ is syntactically identical to $W_{YV_{core}}$, but modifies the subtype semantics to fix types to their original defining environments. This is done by including a second environment during subtyping in order to type the right-hand type. The resulting subtype judgment is double-headed, contextualizing each type within its own environment. The subtype semantics are defined in Figure 20. We omit the declaration subtype rules as they are straightforward. As noted in Section 4, we must still address the issue of the use of Shapes in the argument types of dependent function types. In Figure 21 we define a restricted syntax for $W_{YV_{fix}}$ that restricts argument types to pure materials.

$$\begin{array}{lcl}
\tau & ::= & \text{Material Type} \\
& & \vdots \\
& & \forall(x : \mu). \tau
\end{array}$$

Fig. 21. $W_{YV_{fix}}$ Syntactic Extension

6.1 Subtype Decidability

The subtype decidability argument is relatively easy for $W_{YV_{fix}}$ given the complete exclusion of any form of environment narrowing. In this Section we provide a proof sketch of Subtype Decidability in $W_{YV_{fix}}$, but the full argument is provided as a Coq formulation [[Anon. 2019b](#)].

THEOREM 6.1 (SUBTYPING IN $W_{YV_{FIX}}$ IS DECIDABLE). *For all Γ_1 , τ_1 , Γ_2 and τ_2 , there exists a finite algorithm subtype that returns true if and only if $\Gamma_1 \vdash \tau_1 <: \tau_2 \dashv \Gamma_2$.*

PROOF. We construct our proof by constructing type graphs for τ_1 in Γ_1 and τ_2 in Γ_2 . Since there is no narrowing, it can be demonstrated that all types involved in subsequent questions that arise

out of $\Gamma_1 \vdash \tau_1 <: \tau_2 \vdash \Gamma_2$ are mappable to type graphs that are a subgraph of those constructed for either τ_1 or τ_2 . The proof of decidability proceeds by comparing these two type graphs and demonstrated in two parts:

- (i) Problems of the specific form $\Gamma \vdash \tau <: \mu$ are decidable.
- (ii) Problems of the general form $\Gamma \vdash \tau_1 <: \tau_2$ reduce to the more specific form $\Gamma \vdash \tau <: \mu \vdash \Gamma_2$, and thus decidable.

Case 1 (Subtyping of materials in $W_{\text{yvf}}_{\text{fix}}$ is decidable). Given the syntactic separation imposed in Figure 14, for any question of subtyping that features a material on the right-hand side, all subproofs must also have a material on the right-hand side. Thus, any algorithm that is equivalent to the subtyping in Figure 20 is bound by a finite Shape depth on the left-hand side. Once a shape is reached on the left-hand side, the restriction on extension of Shapes in Figure 13 means no materials may subsequently occur on the left-hand side. Thus, subtyping is bounded by the finite depth of any type refinement (which must be a material) on the right-hand side.

Case 2 (Subtyping in $W_{\text{yvf}}_{\text{fix}}$ is decidable). Since every type cycle must include a Shape, there is a finite depth at which a subtyping must conduct a Shape comparison using S-REFINE. Since type refinements on Shapes must be pure material types, it follows that all subsequent subtype questions must have a Material on the right-hand side.

□

6.2 Subtype Transitivity

The central drawback of $W_{\text{yvf}}_{\text{fix}}$ is the loss of transitivity. The most effective way to see this is through the following example.

```

1 type A = {w => type L = Integer
2           type L' >: Integer}
3 type B = {w => type L = Integer
4           type L' >: w.L}
5 type C = {w => type L <: T
6           type L' >: w.L}

```

While $A <: B$ and $B <: C$, unfortunately $A \not<: C$. The lack of transitive subtyping in the above example is due to the separation of type information during subtyping between the two environments, thus during subtype checking, neither Γ_A nor Γ_C contain the most specific type information. Γ_B does contain the necessary type information, but is not available during subtype checking, at least not in any kind of algorithmic way. Contra-variance is also required to break transitivity. If the example lacked any types in a contra-variant position, then the left-most environment (Γ_A) would always contain the most specific type information, and thus subtyping would be derivable.

The above example not an entirely unreasonable example that can be written off as some kind of corner case. A similar scenario might arise in a data structure that had an element type and a programmer wanted to include a write function such as `append` or `replace`. What is encouraging though is that A does not lack members that C contains, and thus it would be possible to construct a workaround with the help of a cast function that cast terms of type A to B.

7 TYPE SAFETY

Before we construct a type safety argument for $W_{\text{yvf}}_{\text{core}}$ or any of its variants, we first define a term syntax (Figure 22), term typing (Figure 24) and an operational semantics (Figure 25). A **Term** is either a *variable*, *new expression*, *lambda abstraction*, *application* or a *member selection*. A **Member**

$t ::=$	term	$d ::=$	Declaration
x, y, z	variable	$L = \tau$	type
v	value	$l : \tau = t$	value
$t \ t$	application	$E ::=$	Eval. Context
$t.l$	selection	\bigcirc	
$v ::=$	Value	$t \ E$	
$\text{new } \tau \{z \Rightarrow \bar{d}\}$	new	$E \ v$	
$\lambda x : \tau. t : \tau$	abstraction	$E.l$	

Fig. 22. $W_{\text{YV}}_{\text{core}}$ Term Syntax

$\Gamma \vdash \top$ is concrete
$\frac{\Gamma \vdash x.L \ni L \leq \tau \quad \Gamma \vdash \tau \text{ is concrete}}{\Gamma \vdash x.L \text{ is concrete}}$
$\frac{\Gamma \vdash \tau \text{ is concrete}}{\Gamma \vdash \tau \{z \Rightarrow \bar{\sigma}\} \text{ is concrete}}$

Fig. 23. $W_{\text{YV}}_{\text{core}}$ Concreteness

$\Gamma \vdash x : \Gamma(x)$ (T-VAR)	$\frac{\Gamma \vdash \bar{\sigma} \in^z \tau \quad \Gamma, z : \tau \vdash \bar{d} : \bar{\sigma} \quad \Gamma \vdash \tau \text{ is concrete}}{\Gamma \vdash \text{new } \tau \{z \Rightarrow \bar{d}\} : \tau}$ (T-NEW)
$\frac{\Gamma, x : \tau_1 \vdash t : \tau \quad \Gamma, x : \tau_1 \vdash \tau <: \tau_2}{\Gamma \vdash (\lambda x : \tau_1. t : \tau_2) : \forall(x : \tau_1). \tau_2}$ (T-ABS)	
$\frac{\Gamma \vdash t : \forall(x : \tau_1). \tau_2 \quad \Gamma \vdash t' : \tau' \quad \Gamma \vdash \tau' <: \tau_1 \quad x \notin f v(\tau_2)}{\Gamma \vdash t \ t' : \tau_2}$ (T-APP-1)	
$\frac{\Gamma \vdash t : \forall(x : \tau_1). \tau_2 \quad \Gamma \vdash y : \tau' \quad \Gamma \vdash \tau' <: \tau_1}{\Gamma \vdash t \ y : [y/x] \tau_2}$ (T-APP-2)	
$\frac{\Gamma \vdash t : \tau_t \quad \Gamma \vdash \tau_t \ni^z l : \tau \quad z \notin f v(\tau)}{\Gamma \vdash t.l : \tau}$ (T-ACC-1)	$\frac{\Gamma \vdash x : \tau_x \quad \Gamma \vdash \tau_x \ni^z l : \tau}{\Gamma \vdash x.l : [x/z] \tau}$ (T-ACC-2)

Fig. 24. $W_{\text{YV}}_{\text{core}}$ Typing

$\frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']} \quad (\text{R-CTX})$	$\lambda x : \tau_1. t : \tau_2 \ v \longrightarrow [v/x] t \quad (\text{R-ABS})$
$\frac{l : \tau = t \in \bar{d}}{\text{new } \tau \{z \Rightarrow \bar{d}\}.l \longrightarrow [\text{new } \tau \{z \Rightarrow \bar{d}\}/z] t} \quad (\text{R-ACC})$	

Fig. 25. $W_{\text{YV}}_{\text{core}}$ Operational Semantics

Declaration is either a *type* or *value* declaration. Variables are typed by context lookup (T-VAR). New expressions are typed as having their declared type if the type is concrete and the declarations conform to the declaration types of that type. A type is deemed concrete (Figure 23) if it is either \top , an exact or nominal type, or is a refinement on a concrete type. Abstractions are typed as a universally quantified type composed of its argument type and its body type if the body conforms to the body type (T-ABS). Applications can be typed in one of two ways: if the argument is a value (T-APP-1) or if it some general term (T-APP-2). Similarly, member accesses can be typed in one of two ways: if the retriever is a value (T-ACC-1), or if it is some general term (T-ACC-2).

Term reduction is captured by context reduction (R-CTX), β -reduction (R-ABS) and member access (R-ACC).

7.1 $W_{YV_{core}}$

Type safety is not that surprising given the similarities to DOT. In fact, the type safety argument is constructed by leaning on the type safety argument of DOT. We construct an encoding from $W_{YV_{core}}$ to DOT: $W_{YV_{core}} \xrightarrow{\mathcal{D}} \text{DOT}$. We subsequently demonstrate that the typing in Figure 24 and the reduction in Figure 25 represent subsets of the semantics of DOT. We provide this encoding and the type safety argument in the associated technical report [Anon. 2019a], but outline the approach here.

The type safety argument for $W_{YV_{core}}$ is constructed in five theorems:

- (1) Lemma 7.1 proves that membership in $W_{YV_{core}}$ implies a typing in DOT.
- (2) Lemma 7.2 proves that subtyping in $W_{YV_{core}}$ implies an equivalent subtyping in DOT.
- (3) Lemma 7.3 proves that typing in $W_{YV_{core}}$ implies an equivalent typing in DOT.
- (4) Lemma 7.4 proves that term reduction in $W_{YV_{core}}$ implies an equivalent term reduction in DOT.
- (5) Theorem 7.1 proves that term reduction in $W_{YV_{core}}$ does not get stuck.

LEMMA 7.1 ($W_{YV_{core}}$ MEMBERSHIP IMPLIES DOT TYPING). *For all $\Gamma, x, \sigma_x, \sigma_\tau$ and τ , if $\Gamma \vdash x \ni \sigma_x$ and $\Gamma \vdash \sigma_\tau \in \tau$ then $\mathcal{D}(\Gamma) \vdash x : \{\mathcal{D}(\sigma_x)\}$ and $\forall y$, such that $\mathcal{D}(\Gamma) \vdash y : \mathcal{D}(\tau)$, $\mathcal{D}(\Gamma) \vdash y : \{\mathcal{D}([y/z]\sigma_\tau)\}$.*

LEMMA 7.2 (DOT SUBSUMES $W_{YV_{core}}$ SUBTYPING). *For all Γ, τ_1 and τ_2 , if $\Gamma \vdash \tau_1 <: \tau_2$ then $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) <: \mathcal{D}(\tau_2)$.*

LEMMA 7.3 (DOT SUBSUMES $W_{YV_{core}}$ TYPING). *For all Γ, t and τ , if $\Gamma \vdash t : \tau$ then $\mathcal{D}(\Gamma) \vdash \mathcal{D}(t) : \mathcal{D}(\tau)$.*

LEMMA 7.4 (DOT TERM REDUCTION SUBSUMES $W_{YV_{core}}$ TERM REDUCTION). *For all t and t' , if $t \longrightarrow t'$ then $\mathcal{D}(t) \longrightarrow \mathcal{D}(t')$.*

THEOREM 7.1 ($W_{YV_{core}}$ IS TYPE SAFE). *For all t and τ , if $\emptyset \vdash t : \tau$, then term reduction of t does not get stuck.*

7.2 $W_{YV_{self}}$

The semantics of $W_{YV_{self}}$ are equivalent to that of $W_{YV_{core}}$, and represents only a syntactic subset, thus the type safety of $W_{YV_{self}}$ is derived implicitly from the type safety of $W_{YV_{core}}$. This is an advantage of a syntactic restriction over a semantic one, the inheritance of semantic properties is immediate.

7.3 $W_{YV_{fix}}$

Type safety in $W_{YV_{fix}}$ is more complex as it represents a semantic departure from $W_{YV_{core}}$ rather than a syntactic one. It is certainly not possible to attain type safety for $W_{YV_{fix}}$ using the small step semantics in Figure 25 due to the absence of subtype transitivity. Ideally, we would be able to say that every well-typed $W_{YV_{fix}}$ program is also a well-typed $W_{YV_{self}}$ program.

8 DISCUSSION

8.1 Nominality

One of the central themes that arises in this work is the lack of structure present in $W_{YV_{core}}$'s type hierarchy upon which to build a Material/Shape separation. Such a structure is present in Java, as subtyping is built upon subclasses that are explicitly defined. Some structure is available in the way type members can be defined abstractly, and thus only used by explicit refinement on the type, but this is lost when subtyping concrete types.

In our formulation of the Material/Shape separation, Shapes are types that must be subtyped nominally, and can never occur in the lower bounds of types. The unseparated $W_{YV}core$ of Figure 1 always allow a type to be compared structurally at the bounds (this is true of DOT too). Thus, without the addition of a mechanism to prevent both the structural comparison of Shapes and to allow concrete types to be defined as extensions to Shapes, Shapes themselves would be rendered pointless.

We provide two additions to the Material/Shape separated $W_{YV}core$ that ensure these properties: (i) a nominal form for concrete type members ($L \leq \tau$) and (ii) a semantic restriction on the definition of Shapes by refinement (Figure 13).

These two additions allow for the application of an expressive Material/Shape separation while also allowing for the writing of types in the manner of DOT (to a limited degree considering the lack of full intersection types). A stronger distinction between types that are used nominally (type definitions), and types that are used for polymorphism (System F polymorphism or Java generics) would further simplify subtyping.

8.2 $W_{YV}'_{self}$

$W_{YV}'_{self}$ enforces a syntactic restriction on the usage of recursive types, the biggest drawback being the restriction of recursive types in concrete exact types. $W_{YV}'_{self}$ is a variant of $W_{YV}core$ that instead imposes a restriction on the semantics by providing a decidable version of subtyping that ignores the presence of recursive types employed only during contra-variant instances of subtyping. All other instances of subtyping remain the same as $W_{YV}core$. Figure 26 defines the $W_{YV}'_{self}$ variation

$$\frac{\Gamma \vdash \tau_2 <:' \tau_1}{\Gamma \vdash L \geq / = \tau_1 <: L \geq \tau_2} \quad (\text{S-DECL-LOWER}) \qquad \frac{\Gamma \vdash \bar{\sigma}_1 <:' \bar{\sigma}_2}{\Gamma \vdash \tau \{z \Rightarrow \bar{\sigma}_1\} <:' \tau \{z \Rightarrow \bar{\sigma}_2\}} \quad (\text{S-REFINE}')$$

Fig. 26. Subtyping for the $W_{YV}'_{self}$ Variant

$W_{YV}core$ subtyping rules. The variant form of subtyping ($\Gamma \vdash \tau_1 <:' \tau_2$) is only introduced during contra-variant subtyping (as in S-DECL-LOWER). The variant $<:'$ judgment only differs from subtyping in $W_{YV}core$ in that recursive types are not considered during subtyping. Thus subtyping of newly introduced recursive instances of types during $<:'$ cannot consider their bounds, since their definition is not introduced to the context. As the derivation of $\Gamma \vdash \tau_1 <:' \tau_2$ does not include narrowing, it is fairly easy to demonstrate that it is decidable. Thus it is easily demonstrable that the more general form of subtyping $\Gamma \vdash \tau_1 <: \tau_2$ in $W_{YV}'_{self}$ is also decidable due to the absence of contra-variance.

The semantic restriction of $W_{YV}'_{self}$ thus does not need to impose a syntactic one of the form employed in W_{YV}_{self} , allowing exact types to include recursive types. It does however introduce a fairly non-standard approach to subtyping.

8.3 Type Safety

The type safety argument of Section 7.1 is dependent on an encoding into DOT. This is more than just a decision based on convenience and ease, it would be relatively difficult to derive a type safety proof that is self-contained within $W_{YV}core$ primarily for the same reasons that the original DOT type safety proof was difficult to obtain. Type refinements represent a form of intersection type, and thus introduce problems with environment narrowing and transitivity in the presence of ill constructed type bounds. Part of this problem is dealt with by syntactically restricting type definitions to single bound definitions, but through narrowing it would be potentially possible to

construct objects that conceptually contained the following type definitions: $L = \text{String} \wedge L = \text{Int}$. A type safety proof for DOT was achieved by admitting transitivity as a subtype rule, and subsequently demonstrating that this admission did not violate type safety. Such a technique would not be possible in $W_{\text{yvern}}^{\text{core}}$, as an explicit transitivity rule would not allow for a syntax directed definition of subtyping since it is not clear which middle type could be used.

It would be possible to counteract the “bad bounds problem” by restricting type refinements to only concrete types, but this would sacrifice the aspects of $W_{\text{yvern}}^{\text{core}}$. A more satisfying solution would be to introduce full intersection types using the lessons learned in this paper. With intersection types, the gap between $W_{\text{yvern}}^{\text{core}}$ and DOT would be largely bridged.

8.4 Decidability of Typing

It is fairly easy to demonstrate that as in the case of System $F_{<}$, the decidability of term typing in $W_{\text{yvern}}^{\text{core}}$ (Figure 24) and any of its variants is reducible to the decidability of subtyping. It follows that both variants $W_{\text{yvern}}^{\text{self}}$ and $W_{\text{yvern}}^{\text{fix}}$ represent subsets of $W_{\text{yvern}}^{\text{core}}$ in which typing is decidable.

8.5 Comparison with DOT

DOT is more expressive than $W_{\text{yvern}}^{\text{core}}$ (and thus $W_{\text{yvern}}^{\text{self}}$ and $W_{\text{yvern}}^{\text{fix}}$) in several ways, primarily related to the presence of *Intersection* and *Union* types, and the ability to provide both upper and lower bounds to type members.

As we have already noted, intersection types form a core component of DOT, in the absence of which $W_{\text{yvern}}^{\text{core}}$ includes an explicit distinction between term types and declaration types to model structural subtyping and type refinements to gain some minimal instances of the expressiveness of intersection types.

Beyond expressiveness, intersection types provide the ability to achieve type safety in a small step semantics. While intersection types cause the so called “bad bounds” problem that plagued the construction of a type safety proof, full intersection types provide facility for such a proof that the type refinements of $W_{\text{yvern}}^{\text{core}}$ do not. If a type with incompatible bounds is constructed, the fact that it is empty of any well-typed terms saves the soundness proof and is implicit in the semantics of the type. The type refinements of $W_{\text{yvern}}^{\text{core}}$ in effect represent an overwrite, and while they do not allow for the typing of ill-formed terms, they prevent a general transitivity property that allows for ill-formed bounds from being derived.

For the above reason, type members in Wyvern were restricted syntactically to either a single bound or an exact type. This prevents the occurrence of bad bounds, but is a restriction over the type members of DOT that allow for both bounds to be defined. This more general form of type members could likely only be added in the presence of intersection types.

9 FUTURE WORK AND CONCLUSION

9.1 Future work

As has already been discussed, the absence of full intersection types in $W_{\text{yvern}}^{\text{core}}$ limits the ability to derive a type safety argument that is self contained and does not rely on DOT. Further, a lack of intersection types represents a loss of expressiveness, particularly with respect to the encoding Java style multiple inheritance subtype relationships. Both of these provide good reasons to want to introduce full intersection types.

Full intersection types would also facilitate the introduction of the double bounded type members that are present in DOT compared with the single bound type members of $W_{\text{yvern}}^{\text{core}}$ by providing a way to allow ill-formed types but not ill-formed terms.

Muehlboeck and Tate have recently provided a general way to introduce intersection types in a decidable manner that provides a potentially promising approach for constructing a variant of $W_{\text{yvc}}^{\text{core}}$ with intersections that is decidable. In this case, it would be useful to start with a decidable calculus that is free of type refinements before adding intersection types and the later subsumes the former.

9.2 Conclusion

We have cataloged the problems with constructing a language with path dependent types, recursive types, type refinements and decidable subtyping. We have used this insight to construct two variants of our core calculus $W_{\text{yvc}}^{\text{core}}$: $W_{\text{yvc}}^{\text{self}}$ and $W_{\text{yvc}}^{\text{fix}}$. $W_{\text{yvc}}^{\text{self}}$ places a syntactic restriction on recursive types while $W_{\text{yvc}}^{\text{fix}}$ modifies the semantics to extricate environment narrowing from subtyping. They each have their own strengths.

$W_{\text{yvc}}^{\text{self}}$ has a predictable semantics in line with both $W_{\text{yvc}}^{\text{core}}$ and DOT, but the restriction on the use of recursive types means popular patterns such as family polymorphism need to be reformulated. As has already been noted, these patterns can be reclaimed by using either the upper bounded syntactic form for type members ($L \leq \tau$), or the newly introduced nominal form ($L \leq \tau$). This is not without its problems; specifically the nominal form for type members does not exactly fit with the manner in which other syntactic forms for type members work, but it is not so unusual in wider the context of object oriented languages. Languages such as Java, and more importantly Scala, already have distinct syntax that provides for nominal subtyping. $W_{\text{yvc}}^{\text{core}}$'s nominal types behave similarly to explicitly nominal subtype definitions such as the class or trait subtyping of Scala or Java. If programmer intuition is of concern, the following sugaring might moderate programmer expectations.

$$1 \text{ \textbf{type} } C[E, F] \leq D[E]\{ \dots \} \quad \Rightarrow \quad 1 \text{ \textbf{trait} } C[E, F] \text{ \textbf{extends} } D[E]\{ \dots \}$$

$W_{\text{yvc}}^{\text{fix}}$ on the other hand allows recursive types to be freely used, but excludes some instances of subtype transitivity in favor of reflexivity. While practically a programmer could work around the lack transitivity by explicitly introducing a middle type in the form of casts, this implies some confusing semantics in some examples that may appear at first glance to type check but do not. This seems like a more complex semantic peculiarity to convey to programmers.

Finally, the lack of full intersection types in either variant excludes a category of useful examples that use multiple inheritance style type hierarchies. It seems likely that stripping type refinements from $W_{\text{yvc}}^{\text{self}}$ in favor of full intersection types could be done by applying the work of Muehlboeck and Tate. This might finally allow for something approaching a decidable variant of DOT. This is likely easier said than done given the integral role intersection types play in DOT.

ACKNOWLEDGMENTS

We would like to acknowledge Ross Tate, for his invaluable insight, and the reviewers for their comments and feedback. This work was supported by the United States, Department of Defense under contract #H98230-14-C-0140, and Oracle Labs.

REFERENCES

2019. Dotty. <http://dotty.epfl.ch/>. Accessed: 2019-04-21.
- Nada Amin. 2016. Dependent Object Types. (2016), 134. <https://doi.org/10.5075/epfl-thesis-7156>
- Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday* (2016), 249–272.

- Nada Amin, Adriaan Moors, and Martin Odersky. 2012. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*.
- Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-dependent Types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 233–249. <https://doi.org/10.1145/2660193.2660216>
- Anon. 2019a. Decidable Subtyping for Path Dependent Types Technical Report. Supplementary Material: <http://bit.ly/2JAU2Ct>.
- Anon. 2019b. Decidable Wyvern Coq Sources. Supplementary Material: <http://bit.ly/2JAU2Ct>.
- Edoardo Biagioni, Robert Harper, and Peter Lee. 2001. A Network Protocol Stack in Standard ML. *Higher-Order and Symbolic Computation* 14, 4 (2001), 309–356. <https://doi.org/10.1023/A:1014403914699>
- Luca Cardelli and Peter Wegner. 1985. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.* 17, 4 (Dec. 1985), 471–523. <https://doi.org/10.1145/6041.6042>
- Giuseppe Castagna and Benjamin C. Pierce. 1994. Decidable Bounded Quantification. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 151–162. <https://doi.org/10.1145/174675.177844>
- Giuseppe Castagna, Benjamin C. Pierce, and Giorgio Ghelli. 1994. Decidable Bounded Quantification, Appendix B. <http://www.cis.upenn.edu/~bcpierce/papers/fsubnew.ps>. Accessed: 2019.
- Erik Ernst. 2001. Family Polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*. Springer-Verlag, London, UK, UK, 303–326. <http://dl.acm.org/citation.cfm?id=646158.680013>
- Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-bounded Polymorphism into Shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 89–99. <https://doi.org/10.1145/2594291.2594308>
- Radu Grigore. 2017. Java Generics Are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 73–85. <https://doi.org/10.1145/3009837.3009871>
- Robert Harper. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA.
- Bent Kristensen, Ole Madsen, Birger Møller-Pedersen, and Kristen Nygaard. 1987. The BETA Programming Language. *DAIMI Report Series* 16, 229 (1987). <https://doi.org/10.7146/dpb.v16i229.7578>
- O. L. Madsen and B. Møller-Pedersen. 1989. Virtual Classes: A Powerful Mechanism in Object-oriented Programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, New York, NY, USA, 397–406. <https://doi.org/10.1145/74877.74919>
- Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. 1997. *The Definition of Standard ML, Revised Edition*. MIT Press.
- Fabian Muehlboeck and Ross Tate. 2018. Empowering Union and Intersection Types with Integrated Subtyping. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 112 (Oct. 2018), 29 pages. <https://doi.org/10.1145/3276482>
- Abel Nieto. 2017. Towards Algorithmic Typing for DOT (Short Paper). In *Proc. SCALA*. ACM, 2–7. <https://doi.org/10.1145/3136000.3136003>
- Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2013. Wyvern: A Simple, Typed, and Pure Object-oriented Language. In *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and Inheritance (MASPEGI '13)*. ACM, New York, NY, USA, 9–16. <https://doi.org/10.1145/2489828.2489830>
- Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. The Scala language specification.
- Martin Odersky and Matthias Zenger. 2005. Scalable Component Abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 41–57. <https://doi.org/10.1145/1094811.1094815>
- Benjamin C. Pierce. 1992. Bounded Quantification is Undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '92)*. ACM, New York, NY, USA, 305–315. <https://doi.org/10.1145/143165.143228>
- Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.
- Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. 2017. A Simple Soundness Proof for Dependent Object Types. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 46 (Oct. 2017), 27 pages. <https://doi.org/10.1145/3133870>
- John C. Reynolds. 1974. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*. Springer-Verlag, Berlin, Heidelberg, 408–423. <http://dl.acm.org/citation.cfm?id=647323.721503>
- Tiark Rompf and Nada Amin. 2016. 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)*. ACM, New York, NY, USA, 624–641. <https://doi.org/10.1145/2983990.2984008>
- Wyvern. 2019. The Wyvern Programming Language. <http://wyvernlang.github.io/>. Accessed: 2019-04-21.