Decidable Subtyping for Path Dependent Types

by

Julian Mackay

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2020

Abstract

Path dependent types form a central component of the Scala programming language. Coupled with other expressive type forms, path dependent types provide for a diverse set of concepts and patterns, from nominality to F-bounded polymorphism. Recent years have seen much work aimed at formalising the foundations of path dependent types, most notably a hard won proof of type safety. Unfortunately subtyping remains undecidable, presenting problems for programmers who rely on the results of their tools. One such tool is Dotty, the basis for the upcoming Scala 3. Another is Wyvern, a new programming language that leverages path dependent types to support both first class modules and parametric polymorphism. In this thesis I investigate the issues with deciding subtyping in Wyvern. I define three decidable variants that retain several key instances of expressiveness including the ability to encode nominality and parametric polymorphism. Wyv_{fix} fixes types to the contexts they are defined in, thereby eliminating expansive environments. $Wyv_{\mathit{non}\text{-}\mu}$ removes recursive subtyping, thus removing the key source of expansive environments during subtyping. Wyv_{μ} places a syntactic restriction on the usage of recursive types. I discuss the formal properties of these variants, and the implications each has for expressing the common programming patterns of path dependent types. I have also mechanized the proofs of decidability for both Wyv_{fix} and Wyv_{μ} in Coq

Acknowledgments

I would most especially like to acknowledge the tremendous support given by my supervisors, Alex Potanin, Lindsay Groves and Jonathan Aldrich. My supervisors have guided and advised me, and are owed much of the credit for the work within this thesis.

I would like to thank the input to this work from Ross Tate, who's invaluable insight and advice changed the entire trajectory of the research in this thesis.

I would also like to thank the examiners for their work in evaluating this thesis.

I am also grateful to the friendship I have received from the many office mates I have had over the years, in the many offices I have occupied. While I appreciate the support and friendship of many of my colleagues, I should in particular acknowledge Michael Homer, Tim Jones, Alex Sawcuk da Silva, Chen Wong and Erin Greenwood-Thessman.

I would also like to thank the wider School of Engineering and Computer Science for providing me a community over the past few years.

Finally, I would like to thank my family for their support, and perhaps most importantly I recognise that I owe my partner Fiona McNamara an immense amount of emotional support.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

The last decade has seen much work in laying the formal foundations of so-called *Path Dependent Types*, a language feature that features most prominently as a core component of the Scala programming language. Simply stated, type definitions can exist within objects (in the same manner a value or function can), and a path dependent type is the type that is derived from accessing that type via a specific path. As an example consider the following definition:

```
val o : Object = new {
    type L = Integer
}
val i : o.L = 4
```

Here we initialise a new object `o` with a type `L`, and subsequently use that type in the creation of `i` by selection on `o`.

While path dependent types are fairly simple conceptually, they allow for a rich variety of patterns, especially when combined with other powerful type forms such as intersection, recursive and dependent function types. They offer an alternative to the F-bounded polymorphism of functional languages such as System $F_{<:}$, type classes of Haskell and class generics of Java. Path dependent types also introduce a notion of nominality, which allows for the modelling of ML-style modules.

With this rich conceptual ecosystem comes some significant trade-offs. The formal properties of path dependent types have been quite difficult to nail down. Much effort has been spent on deriving a type safety proof for path dependent types in presence of type refinements, recursive types or a lattice structure for subtyping. Several variations on the core formal semantics for Scala have been proposed and they were eventually demonstrated to be type safe in 2016 by Rompf and Amin for the Dependent Object Types (DOT). Central to the struggle for type safety was the transitivity of subtyping, or rather ensuring subtype transitivity while also ensuring other key properties such as well-formedness of type bounds and environment narrowing (the narrowing of variable types to more specific forms during subtyping or reduction). The resulting formalism explicitly included subtype transitivity as part of the subtype rules, but demonstrated that this at no point introduced ill-formed expressions, even if ill-formed types might exist.

Rich type systems also have other downsides. It has long been known that subtyping of bounded quantification is undecidable, but more recently Grigore demonstrated that subtyping of Java generics was also undecidable. Since a language such as DOT, which features both path dependent types and some form intersection types, subsumes the subtyping of both bounded quantification and Java generics, it also captures the undecidability of their subtyping.

Not only do languages such as DOT capture the decidability issues of bounded quantification and Java generics, but it also introduces several other problems: recursive types, mutually defined types and intersection types.

This thesis categorizes the issues of decidability present in the Wyvern programming language, a language closely related to DOT that includes path dependent types, recursive types and limited intersection types. The thesis is organised in the following way:

- Chapter 2 covers the background and related work associated with this thesis.

- Chapter 3 introduces a minimal core calculus for Wyvern called Wyv_{core}. I then discuss the associated decidability issues with it.

- Chapter 4 defines a Material/Shape separation on Wyvern, a syntactic separation on types that constrains recursion in order to achieve subtype decidability.

- Chapter 5 defines a general decidability argument for decidable variants of Wyv_{core}. I then define Wyv_{fix}, a decidable variant that removes the narrowing of environments during subtyping. A proof of decidability is provided. This decidability proof has been mechanised in Coq [2].

- Chapter 6 defines a decidable variant of Wyvern removes recursive subtyping in order to achieve subtype decidability. After proving decidability, I then prove type safety.

- Chapter 7 defines a decidable variant of Wyvern that places a syntactic restriction on recursive types. A proof of decidability is provided, along with a proof of type safety. The proof of decidability has been mechanised in Coq [2].

- Chapter 8 concludes.

Parts of the work in this thesis has been published. The novel type systems presented in Chapters 5 and 7 are the topic of our POPL 2020 paper [55].

Finally, I have mechanised two of the proofs of decidability, that of Wyv_{fix} and Wyv_{μ} in Coq [2].

Chapter 2

Background

In this thesis I investigate the issues of subtype decidability in type systems (formal models of typed languages) for object oriented languages with path dependent types. I present several variations on a single type system that contain some common features with other programming languages. These languages and features have been investigated to varying degrees, and the work in this thesis draws directly upon work that other researchers have done over the past few decades.

In this Chapter I provide background among others on

- In Section 2.1 I discuss type systems, the formal method I use to model the languages discussed in this thesis. Section 2.1 takes a tutorial approach initially, introducing the type systems of a family of languages referred to as the λ-calculus. These are foundational calculi for functional languages, but they have much in common with the object oriented languages that occupy the rest of this thesis.

- Section 2.2 introduces object oriented languages, and discusses subtyping in object oriented languages. I also introduce and discuss the Dependent Object Types (DOT) calculus, an object oriented language that is the perhaps the most prominent formal calculus that features the language features at the centre of this thesis: recursive types and path dependent types. I end this section with a discussion on Java and its associated issues with decidability.

- Section 2.3 introduces the Wyvern, the basis and motivation of the work in this thesis. I introduce the syntax of Wyvern, and discuss topics that motivate it.

2.1 Type Systems

In Computer Science, a type system is a commonly used formal method for defining a programming language and demonstrating a variety of properties held by that language, usually with the purpose of excluding problematic behaviour. Pierce [70] provides the following definition of a type system:

> A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute.

Generally, a type system uses a syntax to categorize components of a program and a semantics to define properties and behaviour of elements of that syntax. In this Section I present a lineage of type systems, each building on the last to successively exclude a greater class of problematic programs. This both demonstrates how type systems are used to model programming languages, as well as providing the conceptual foundations to the more complex type systems defined in the following chapters. The problem of subtype decidability does not arise suddenly with the introduction of Wyvern, the language we focus on in this thesis, but rather arises from the confluence of features described in several smaller calculi. For now, we start with the λ-Calculus.

2.1.1 λ-Calculus

The λ-calculus is a family of higher-order functional programming languages defined by Church[30, 42]. Some of these will be discussed in this section,

$$\begin{array}{llr} t & ::= & \textbf{Terms} \\ & x & \textit{variable} \\ & \lambda x.t & \textit{abstraction} \\ & t\ t & \textit{application} \end{array}$$

Figure 2.1: λ-Calculus Syntax

$$\lambda x.t \longrightarrow \lambda y.[y/x]t \quad (\alpha - conversion)$$

$$(\lambda x.t_1)\ t_2 \longrightarrow [t_2/x]t_1 \quad (\beta - reduction)$$

Figure 2.2: λ-Calculus Operational Semantics

each building on the last to demonstrate the use of type systems and provide a theoretical basis for the languages modelled in the rest of the thesis. I start with the *untyped λ-calculus* the syntax of which is defined in Figure 2.1. Expressions in the λ-calculus are restricted to only three forms: variables, *λ-abstractions* (or functions) and applications. λ-calculus evaluation is defined in Figure 2.2 and is performed via a series of variable renames ($\alpha - conversion$) and function applications ($\beta - reduction$). Both of these evaluations make use of variable substitution. The substitution $[t'/x]e$ is a replacement of all instances of x in t with t'.

The untyped λ-calculus is extremely simple and captures the key aspects of functional programming languages. For the purposes of meaningful examples I introduce the modelling of some basic data types in the λ-calculus. The boolean values `true` and `false` are encoded below.

$$\texttt{true} = \lambda x.\lambda y.x \qquad \texttt{false} = \lambda x.\lambda y.y$$

Such encodings are referred to as Church encodings [42]. It is further possible to encode simple logic operators.

$$\wedge = \lambda a.\lambda b.a\ b\ a \qquad \vee = \lambda a.\lambda b.a\ a\ b \qquad \neg = \lambda a.a\ \texttt{false}\ \texttt{true}$$

It is easy to demonstrate for the operational semantics defined in Figure 2.2, we get the following identities.

$$\wedge\ \texttt{true}\ \texttt{false} = \texttt{false} \qquad \vee\ \texttt{true}\ \texttt{false} = \texttt{true} \qquad \neg\texttt{true} = \texttt{false}$$

The λ-calculus syntax restricts the form that terms may take, already excluding the category of programs that are syntactically ill-formed. Depending on the definition of ill-formed, however it does not exclude all potentially ill-formed terms. Consider the following terms:

$$(\wedge\ \lambda x.x)\ \texttt{true} \qquad x\ \texttt{true}$$

While the first example is syntactically valid and evaluates to a valid expression, it does not adhere to the intent of the $\wedge$ function. That is, $\wedge$ was defined with the intent that it operates on boolean terms. The second term is also syntactically valid. It is however irreducible since x is not an abstraction. What is more, it does not make much sense. There is no binding for x. There are several problems that arise in these two examples. Firstly, a programmer might expect functions they write to only operate on a specific subset of terms. Secondly, we expect well-formed terms to be *closed* (there are no unaccounted free variables present within the term) unless they are bound within an abstraction. Finallly, we expect terms to take the correct form, that is the left-hand side of an application should be an abstraction.

2.1.2 The Simply Typed λ-Calculus: $\lambda \rightarrow$

The Simply Typed λ-Calculus extends the untyped λ-calculus with simple types. There are typed lambda calculi with more complex types, some of which I will discuss in later sections. In the simply typed lambda calculus, a types is either an atomic type A (perhaps `Bool` or `Int`) or an “arrow type”, $\tau_1 \rightarrow \tau_2$, specifying a function from values of type τ_1 to values of type τ_2. This type syntax is formally defined in Figure 2.3.

$$\tau \ ::= \ A \mid \tau \ \rightarrow \ \tau$$

Figure 2.3: Types in the Simply Typed λ-Calculus

What is a Type?

A common description of a type considers a type as a set of possible values within a programming language. That is, all inhabitants of a type adhere to the specification of the type. In the simply typed λ-calculus all values have either some atomic type, or a function type of the form $\tau_1 \ \rightarrow \ \tau_2$. That is, all values are either some primitive data type, or a function. If the simply typed λ-calculus were extended with boolean values and the type `Bool`, types would now provide a distinction between a function and a boolean term.

Morris provides a different perspective on the concept of "type" in programming languages. They argue that type is not an extensional property in the way that set membership is, rather types are used to define the properties values of that type must have. Specifically, Morris argues that a type does not define what a value is, but where it came from, and how it may be used.

This is a topic that arises frequently. Equality of two sets is extensional, and is defined entirely by the members of those sets. This is not the case with types. Equality of types is generally defined by the specifications the types provide. Later in this Chapter (Section 2.2.1) *intersection* and *union* types introduce the type operators ∩ and ∪. Such type operators imply a set theoretic perspective on types. Without getting into the details of these types, a value has type $\tau_1 \ \cap \ \tau_2$ if it has both type τ_1 and τ_2. Thus, while the analogy of types as sets may not hold in general, it remains a useful analogy under certain circumstances.

t	$::=$	**Terms**
	x	*variable*
	$\lambda x : \tau . t$	*abstraction*
	$t\ t$	*application*
	`true`	
	`false`	
	`if` t `then` t `else` t	*conditional*

Figure 2.4: Simply Typed λ-Calculus with Booleans

$$\texttt{if true then } t_1 \texttt{ else } t_2 \longrightarrow t_1 \qquad (\text{R-}\textsc{If}_1)$$

$$\texttt{if false then } t_1 \texttt{ else } t_2 \longrightarrow t_2 \qquad (\text{R-}\textsc{If}_2)$$

Figure 2.5: Reduction of Boolean Conditionals

Adding Types to the λ-Calculus

I already defined the syntax for types in the simply typed λ-calculus in Figure 2.3. In Figure 2.4 I provide the grammar of a simply typed λ-calculus that includes not only arrow types, but also includes boolean as an atomic type [70]. Types as defined in Figure 2.3 categorize terms into either functions (of type $\tau \rightarrow \tau$) or boolean terms (of type `Bool`).

Terms are also extended, most importantly by adding a type to the syntax of abstractions ($\lambda x : \tau . t$). This type is discussed later, but is used to specify the allowable arguments that the function may take. Some boolean specific terms are also added, `true`, `false` and conditional terms. `true` and `false` are irreducible and thus values. Conditionals however are reducible, and I give their reduction as in Figure 2.5 as an extension to the reduction of the base terms of the λ-Calculus.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \ : \tau} \quad (\text{T-Var}) \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1.t \ : \tau_1 \to \tau_2} \quad (\text{T-Abs})$$

$$\frac{\Gamma \vdash t \ : \tau' \to \tau \qquad \Gamma \vdash t' \ : \tau'}{\Gamma \vdash t\ t' \ : \tau} \quad (\text{T-App}) \qquad \Gamma \vdash \texttt{true} \ : \texttt{Bool} \quad (\text{T-True})$$

$$\Gamma \vdash \texttt{false} \ : \texttt{Bool} \quad (\text{T-False})$$

$$\frac{\Gamma \vdash t \ : \texttt{Bool} \qquad \Gamma \vdash t_1 \ : \tau \qquad \Gamma \vdash t_2 \ : \tau}{\Gamma \vdash \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \ : \tau} \quad (\text{T-If})$$

Figure 2.6: Simply Typed λ-Calculus Type Rules

By adding the `true`, `false` and conditional terms to the grammar, we are able to define boolean operators more effectively and make use of the classification that the types give us. Below are the new encodings for basic boolean operators.

$$\wedge = \lambda a : \texttt{bool}.\lambda b : \texttt{bool}.\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else false}$$
$$\vee = \lambda a : \texttt{bool}.\lambda b : \texttt{bool}.\texttt{if}\ a\ \texttt{then true else}\ b$$
$$\neg = \lambda a : \texttt{bool}.\texttt{if}\ a\ \texttt{then false else true}$$

The types defined in Figure 2.3 can be used to categorize valid terms using a static semantics defined in Figure 2.6. Γ is referred to as a *context* or an *environment*, and is used to store the types of variables during type checking. Thus T-Var is used to type variables within a context. This excludes the class of programs that may use variables that haven't been introduced. T-Abs is used to type abstractions, checking that the body of an abstraction has the correct type given the type of the variable. Applications are typed using T-App, checking that the right-hand side is of a function type that accepts terms of the type on the left-hand side. T-True and T-False state that `true` and `false` have the type `Bool`. Finally, T-IF specifies that the condition of a conditional must be a boolean, and both branches must have the same type.

The type system of the simply typed λ-calculus has allowed programmers to exclude a specific set of undesirable behaviour. Using the rules in Figure 2.6, we can categorize acceptable terms and unacceptable ones. Specifically, we are able to exclude the earlier examples. With the new definition of $\wedge$, the previous example, $(\wedge\ \lambda x.x)$ `true`, is now ill-formed and can be rejected statically. Typing can be used reject terms with unbound, free variables. x `true` is can be rejected by the rules in Figure 2.6, as x is unbound in the empty environment.

Static type checking is however conservative in it's application, and there are trade-offs for this safety. A simple example is given below.

$$\lambda x.x$$

A general identity function is now not generally expressible in the simply typed λ-calculus. Such a function would have to specify a type for the argument x, and a new function would have to be written for each potential data type.

$$\lambda x : \texttt{Bool}.x \qquad \lambda x : \texttt{Bool} \rightarrow \texttt{Bool}.x$$

While this may be a piece of functionality that a language could do without, if lists were introduced to the language, this limitation would have similar implications. For example, when writing a function that appends an element to the end of the list, a new `append` function would have to be written for each data type that could potentially populate a list even if the behaviour of `append` does not necessarily depend on the type of the elements involved.

2.1.3 Polymorphism: System F

Polymorphism is the ability to define a program (or part of a program) that has different behaviour depending which types are used [24]. System F (sometimes referred to as $\lambda 2$ [16]) [37, 74] is another typed λ-calculus that extends

$$
\begin{array}{lll}
t & ::= & \textbf{Terms} \\
 & \vdots & \\
 & \Lambda\alpha.t & \textit{type abstraction} \\
 & t\ \tau & \textit{type application}
\end{array}
\qquad\qquad
\begin{array}{lll}
\tau & ::= & \textbf{Types} \\
 & \vdots & \\
 & \alpha & \textit{type variable} \\
 & \forall\alpha.\tau & \textit{all}
\end{array}
$$

Figure 2.7: System F with Booleans

the simply typed λ-calculus with polymorphism. The extended grammar of System F with boolean terms is given in Figure 2.7.

Types in System F are extended with type variables (α) and universally quantified types ($\forall\alpha.\tau$), where the type τ is quantified over some type α. These types categorize the new terms that System F extends the grammar with: *type abstractions* ($\Lambda\alpha.t$) and *type applications* ($t\ \tau$). A type abstraction similar to an abstraction on a term, except instead of quantifying terms over other terms, terms are quantified over types. Type applications are an application of a type abstraction on a type. Quantification over types allow for programmers to write a general identity function that was previously not possible in the simply typed λ-calculus. Define *id* as

$$id = \Lambda\alpha.\lambda x : \alpha.x$$

id is a generally applicable function giving the following identities:

$$id\ \texttt{Bool}\ \texttt{true} = \texttt{true} \qquad id\ \texttt{Bool} \rightarrow (\texttt{Bool} \rightarrow \texttt{Bool})\ \wedge\ = \wedge$$

Figure 2.8 defines the extension to type rules of the simply typed λ-calculus that System F provides. A type abstraction is typed with the universally quantified type whose body is the type of the type abstraction's body. The type of a type application is the type of the body of the type abstraction on the left where the type variable on the left is substituted for the the argument type on the right.

The evaluation rule for type applications is given in Figure 2.9.

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \Lambda\alpha.t\ : \forall\alpha.\tau}\ (\text{T-Type-Abs}) \qquad \frac{\Gamma \vdash t\ : \forall\alpha.\tau}{\Gamma \vdash t\ \tau'\ : [\tau'/\alpha]\tau}\ (\text{T-Type-App})$$

Figure 2.8: System F Type Rule Extension

$$\Lambda\alpha.t\ \tau\ \longrightarrow [\tau/\alpha]t \quad (\text{R-Type-App})$$

Figure 2.9: System F Evaluation Rule Extension

While System F provides the expressiveness of generalizing behaviour over types, it does not allow the programmer to specify any information about those types. For instance, the following expression is ill-formed in System F.

$$\texttt{apply} = \Lambda\alpha.\lambda x : \alpha.\lambda y : \texttt{Bool}.x\ y$$

The polymorphic `apply` function above takes three parameters, α, x and y. x is specified to be of type α and y of type `Bool`. x is then applied to y, but the type of x (α) is not an “arrow type” (a type of the form $\tau_1 \ \rightarrow\ \tau_2$), thus the entire expression is ill-formed.

2.1.4 Bounded Quantification: System $\mathbf{F}_{<:}$

System $F_{<:}$ [25, 69] adds two new concepts on top of System F: *Subtyping* and *Bounded Quantification*.

Subtyping

If types denote sets of values within a language, *subtyping* captures the notion of a subset. Thus, if τ_1 subtypes τ_2, then any value that inhabits the set denoted by τ_1 also inhabits the set denoted by τ_2. Specifically subtyping is the concept of substitutability [53]. That is, if τ_1 subtypes τ_2, values of type

$$\tau \;<:\; \top \quad (\textsc{S-Top}) \qquad\qquad \tau \;<:\; \tau \quad (\textsc{S-Refl})$$

$$\frac{\tau_2 \;<:\; \tau_1 \qquad \tau_1' \;<:\; \tau_2'}{\tau_1 \to \tau_1' \;<:\; \tau_2 \to \tau_2'} \;(\textsc{S-Arr}) \qquad\qquad \frac{\tau_1 \;<:\; \tau_2}{\forall\alpha.\tau_1 \;<:\; \forall\alpha.\tau_2} \;(\textsc{S-All})$$

Figure 2.10: Subtyping for System F

τ_1 can be substituted in any part of a program that expects a value of type τ_2. If subtyping implies a subset relationship, it is useful to have a concept of a universal set. $\top$ is the universal type that contains all values. Thus all types subtype $\top$.

Figure 2.10 provides an initial attempt to introduce subtyping to System F. S-Top has already been discussed, but Figure 2.10 introduces three other rules. S-Refl explicitly introduces *subtype reflexivity* to System F subtyping, that is every type is a subtype of itself. The subtyping of arrow types (S-Arr) enforces inverse relationships on the parameter type versus the return type. The intuition here derives from the substitutability principle [53]. If a program expects an expression of type $\tau_2 \to \tau_2'$, an expression of type $\tau_1 \to \tau_1'$ may only be provided if it may be treated as having the expected type and thus accept expressions of type τ_2 and return an expression of type τ_2'. Thus the argument type τ_1 must be at least τ_2 and τ_1' must be at most τ_2'. These relationships are known as *contra-variant* ($\tau_2 \;<:\; \tau_1$) and *covariant* ($\tau_1' \;<:\; \tau_2'$) relationships respectively [8]. Subtyping of universally quantified types (S-All) is based on covariant subtyping of the bodies of the type.

Bounded Quantification

Bounded Quantification mixes the ideas of subtyping and polymorphism [70]. While both concepts are individually useful, together they provide new expressiveness to the typed λ-calculus. Bounded quantification in typed λ-

calculi means type abstractions specify bounds on the abstracted type. Below the earlier `apply` example is revisited with a slight modification.

$$\texttt{applyBoolean} = \Lambda\alpha \leqslant \texttt{Bool} \rightarrow \texttt{Bool}.\lambda x : \alpha.\lambda y : \texttt{Bool}.x\ y$$

The variable α is now bounded by the type $\texttt{Bool} \rightarrow \texttt{Bool}$. This has two implications, firstly x may now be identified as a function statically during type checking, and secondly any improper usage of the function may be rejected statically.

System $\mathbf{F}_{<:}$

System $F_{<:}$ extends System F with subtyping and bounded quantification to provide greater expressiveness than either polymorphism or subtyping can provide alone. The extended syntax of System $F_{<:}$ over System F is given in Figure 2.11, while the type and subtype rules are given in Figures 2.12 and 2.13 respectively.

Typing in System $F_{<:}$ is uses two environments, Γ a map of expression variables to types and Δ a map of type variables to types (the bounds of the types). The differences when compared to the typing rules of System F are visible in the T-Type-App and T-Sub rules, both of which involve subtyping. The application of type abstractions (T-Type-App) requires the argument type to subtype the bound on the type variable and any expression of type τ can be considered as inhabiting any type that τ subtypes (T-Sub). This final rule is known as the subsumption rule and fulfils the intent of substitutability.

Subtyping of System $F_{<:}$ in Figure 2.13 introduces two new rules (S-Var and S-Trans) and modifies one (S-All) rule in Figure 2.10. S-Var defines subtyping of type variables in relation to their bounds. Type variables are mapped to their bounds in the context Δ. In S-All, additions to Δ are made during the comparison of the bodies of two universally quantified types. S-All also enforces a contra-variant relationship between the bounds of two universally quantified type. The S-Trans subtype rule explicitly introduces

$$
\begin{array}{lll}
e & ::= & \textbf{Expressions} \\
 & \vdots & \\
 & \Lambda\alpha \leqslant \tau.e & \textit{type abstraction}
\end{array}
\qquad\qquad
\begin{array}{lll}
\tau & ::= & \textbf{Types} \\
 & \vdots & \\
 & \top & \textit{top} \\
 & \forall(\alpha \leqslant \top).\tau & \textit{all}
\end{array}
$$

Figure 2.11: System $F_{<:}$ Syntax extension over System F

$$
\frac{\Gamma(x) \;=\; \tau}{\Delta;\Gamma \vdash x \;:\; \tau}\ (\textsc{T-Var})
\qquad\qquad
\frac{\Delta;\Gamma, x:\tau \vdash e \;:\; \tau'}{\Delta;\Gamma \vdash \lambda x:\tau.e \;:\; \tau \to \tau'}\ (\textsc{T-Abs})
$$

$$
\frac{\begin{array}{c}\Delta;\Gamma \vdash e \;:\; \tau' \to \tau \\ \Delta;\Gamma \vdash e' \;:\; \tau'\end{array}}{\Delta;\Gamma \vdash e\ e' \;:\; \tau}\ (\textsc{T-App})
\qquad\qquad
\frac{\Delta;\alpha \leqslant \tau;\Gamma \vdash e \;:\; \tau'}{\Delta;\Gamma \vdash \Lambda(\alpha \leqslant \tau).e \;:\; \forall(\alpha \leqslant \tau).\tau'}\ (\textsc{T-Type-Abs})
$$

$$
\frac{\Delta;\Gamma \vdash e \;:\; \forall\alpha \leqslant \tau'.\tau'' \qquad \Delta \vdash \tau \;<:\; \tau'}{\Delta;\Gamma \vdash e\ \tau \;:\; [\tau/\alpha]\tau''}\ (\textsc{T-Type-App})
$$

$$
\frac{\Delta;\Gamma \vdash e \;:\; \tau' \qquad \Delta \vdash \tau' \;<:\; \tau}{\Delta;\Gamma \vdash e \;:\; \tau}\ (\textsc{T-Sub})
$$

Figure 2.12: System $F_{<:}$ Typing Rules

$$
\Delta \vdash \tau \;<:\; \top \quad (\textsc{S-Top})
\qquad\qquad
\Delta \vdash \tau \;<:\; \tau \quad (\textsc{S-Refl})
\qquad\qquad
\frac{\Delta(\alpha) = \tau}{\Delta \vdash \alpha \;<:\; \tau}\ (\textsc{S-Var})
$$

$$
\frac{\begin{array}{c}\Delta \vdash \tau_1 \;<:\; \tau \\ \Delta \vdash \tau \;<:\; \tau_2\end{array}}{\Delta \vdash \tau_1 \;<:\; \tau_2}\ (\textsc{S-Trans})
\qquad\qquad
\frac{\begin{array}{c}\Delta \vdash \tau_2 \;<:\; \tau_1 \\ \Delta \vdash \tau_1' \;<:\; \tau_2'\end{array}}{\Delta \vdash \tau_1 \to \tau_1' \;<:\; \tau_2 \to \tau_2'}\ (\textsc{S-Arr})
$$

$$
\frac{\begin{array}{c}\Delta \vdash \tau_2 \;<:\; \tau_1 \\ \Delta, \alpha \leqslant \tau_2 \vdash \tau_1' \;<:\; \tau_2'\end{array}}{\Delta \vdash \forall(\alpha \leqslant \tau_1).\tau_1' \;<:\; \forall(\alpha \leqslant \tau_2).\tau_2'}\ (\textsc{S-All})
$$

Figure 2.13: System $F_{<:}$ Subtyping Rules

transitive subtyping, that is if τ subtypes τ', then it transitively subtypes any type that τ' subtypes. Subtyping only uses one environment, Δ, the mapping of type variables to type bounds. An environment for typing term variables, Γ, is not required as types do not include term variables.

2.1.5 Properties of Type Systems

Type systems provide formal means for demonstrating useful properties of programming languages. Primarily, we are concerned with demonstrating undesirable behaviour does not exist, and desirable behaviour does. While "desirable" and "undesirable" are often difficult to define, we can informally say that we want to allow for common patterns used by programmers, and disallow expressions that result in errors.

Subtype Reflexivity

A language is *subtype reflexive* if values of some type τ are substitutable in parts of programs that require type τ. This may seem like an obvious property, but it is not necessarily true of any particular type system. Due to S-Refl, subtype reflexivity is by definition a property System $F_{<:}$.

Subtype Antisymmetry

Subtyping of a language is said to be *anti-symmetric* if whenever some types τ_1 and τ_2 subtype each other, it follows that $\tau_1 = \tau_2$. The anti-symmetry of subtyping implies a set theoretic nature of subtyping in that for sets A and B, if $A \subseteq B$ and $B \subseteq A$, it follows that $A = B$. This is not necessarily true of all type systems, especially since equality implies a syntactic equality rather than a semantic equivalence. In languages where types can have different names, but the same semantic meaning, antisymmetry of subtyping is unlikely to hold unless a semantic notion of equality is defined.

Subtype Transitivity

Subtype transitivity has particular importance for the work in this thesis. Transitivity of subtyping has already been briefly introduced in the subtype rules of System $F_{<:}$, but is expanded on here. A subtyping is transitive if whenever some type τ_1 subtypes a type τ_2 and τ_2 in turn subtypes another type τ_3, it follows that τ_1 subtypes τ_3. While this is definitionally true of subtyping System $F_{<:}$, it is often a fairly difficult result to derive, and conflicts with several other type theoretic properties.

Type Safety

Type Safety (*soundness* of the typing judgement) is one of the central targets of defining a type system for a language. A language is said to be type safe if all well-formed expressions in well-formed contexts reduce to well-formed expressions and do not result in type errors. Type safety is often formulated using two different theorems, *Progress* (Theorem 2.1.1) and *Preservation* (Theorem 2.1.2) [85].

Progress demonstrates that any well-formed expression is either fully evaluated (a value, e.g. an abstraction or type abstraction in System $F_{<:}$), or can be further evaluated (e.g. an application or type application in System $F_{<:}$). This implies that at no point can a well-formed program get stuck.

Property 2.1.1 (Progress)**.** *For any expression e that is well-formed, either e is a value, or there exists e' such that $e \longrightarrow e'$.*

Preservation demonstrates that a well-formed program will never reduce to an ill-formed program.

Property 2.1.2 (Preservation)**.** *For any well-formed expression e in a well-formed context Γ if e reduces to some e', e' has the same type as e.*

Together, *Progress* and *Preservation* ensure type safety, that a well-formed expression is either fully evaluated, or it can evaluate to some other well-formed expression.

Type safety is often a nebulous target to reach. Both type systems of Scala and Java were found to be unsound by Amin and Tate in [11]. The existence of `null` in these languages, coupled with wild-cards in Java [21, 38] and lower bounds in Scala [66] allow any type to indirectly subtype any other type using null as a witness to this relationship.

The type system of TypeScript [7] has several documented instances of unsound behaviour. Many of these are intentionally introduced as part of the language, often to provide for common patterns used in JavaScript [5], the underlying basis of TypeScript.

Syntax Directedness of a Rule Set

We say that a set of rules is syntax directed if every rule is applicable for a unique set of syntactic elements. As an example, the typing of Figure 2.6 is syntax directed. Given any syntactically valid term of the Simply Typed λ-Calculus, there can only be one rule that is applicable in deriving its typing. This is a valuable property of rule sets as it means that the rule set itself constitutes an algorithm for checking that property (in the case of Figure 2.6, typing). If a rule set is syntax directed, then we are able to invert the declarative rule set in to a decision procedure, where each premise in a declarative inference rule now becomes the next step in the algorithm.

Subtype Decidability

For a type system to be useful, the properties that type system enforces have to be checkable. For example, the type rules defined in Figure 2.6 are relatively easy to define a type checking algorithm for, as they are syntax directed and thus are themselves an algorithm. Furthermore, it would be relatively easy to demonstrate that such an algorithm could always come to a conclusion on typing of simply typed λ-calculus expressions since type checking in this instance is bounded by the syntactic depth of the expression. In this case, typing of the simply typed λ-calculus is said to be *decidable*.

Subtype Decidability is what much of this thesis is concerned with. Subtyping provides much expressiveness to languages, but itself presents a problem for type checkers. A more complex algorithm than typing in System F would need to be one defined for the subtype rules of System $F_{<:}$ (Figure 2.13). In [69], Pierce demonstrated that there does not exist an algorithm that is able to decide System $F_{<:}$ subtyping, that Subtyping in System $F_{<:}$ is *undecidable*. Pierce demonstrated this by defining a reduction of the Halting problem for two counter Turing machines to subtyping in System $F_{<:}$.

Subtype Undecidability in System $F_{<:}$

System $F_{<:}$ suffers from a divergence of contexts during certain instances of subtyping. Due to a combination of contra-variant types and context modification during subtyping, infinite types are created and added to the context during subtyping. The System $F_{<:}$ syntax and subtype semantics have already been given in Figures 2.11 and 2.13.

While Figure 2.13 provides the full subtyping rules, examples of divergent contexts and non-terminating subtyping are only dependent on the S-Var and S-All rules. For convenience we use a shorthand for negation types provided at the top of Figure 2.14 in S-Neg, however this shorthand is not necessary in demonstrating divergence. Subtyping of negation types implies a negative subtype relationship. The example of subtyping in Figure 2.14 demonstrates how divergence occurs during subtyping. Using only the rule for universally quantified types and variable lookup, we can see that subtyping quickly diverges. Types are never repeated, rather an infinite number of similar types are created.

The origin of the divergence is a combination of the modification of the context during subtyping and contra-variance. Context modifications allow type definitions to be redefined, while contra-variance allows redefinition to occur on both sides of the subtype relation (if contra-variance were not present, the unmodified type would be bound syntactically and would not diverge). Such a modification can be seen in the second premise of S-All,

$$\frac{\Gamma \vdash (\forall\alpha \leqslant T_1.\top) <: (\forall\alpha \leqslant T_2.\top)}{\Gamma \vdash \neg T_1 <: \neg T_2} \ \text{(S-NEG)} \qquad \text{let } T_0 = (\forall\alpha \leqslant \top.\neg(\forall\beta \leqslant \alpha.\neg\beta))$$

$$\begin{array}{llll}
\emptyset & \vdash & (\forall\alpha_0 \leqslant \top.\alpha_0) & <: & (\forall\alpha_0 \leqslant T_0.(\forall\alpha_1 \leqslant \alpha_0.\neg\alpha_1)) \\
\alpha_0 \leqslant T_0 & \vdash & \alpha_0 & <: & (\forall\alpha_1 \leqslant \alpha_0.\neg\alpha_1) \\
\alpha_0 \leqslant T_0 & \vdash & (\forall\alpha_1 \leqslant \top.\neg(\forall\beta \leqslant \alpha_1.\neg\beta)) & <: & (\forall\alpha_1 \leqslant \alpha_0.\neg\alpha_1) \\
\begin{array}{l}\alpha_0 \leqslant T_0, \\ \alpha_1 \leqslant \alpha_0\end{array} & \vdash & \neg(\forall\beta \leqslant \alpha_1.\neg\beta) & <: & \neg\alpha_1 \\
\begin{array}{l}\alpha_0 \leqslant T_0, \\ \alpha_1 \leqslant \alpha_0\end{array} & \vdash & \alpha_1 & <: & (\forall\beta \leqslant \alpha_1.\neg\beta) \\
 & \vdots & & &
\end{array}$$

Figure 2.14: Non-termination of subtyping in System F$_{<:}$

where α is originally bound by S_1 in T_1, but in subsequent subtype checks is bound by S_2. Contra-variance is present in the subtyping of type bounds.

Kernel F$_{<:}$

Pierce [70] defined a decidable subset of System F$_{<:}$ called Kernel F$_{<:}$. Kernel F$_{<:}$ modifies the subtype semantics of System F$_{<:}$. Figure 2.15 provides the Kernel F$_{<:}$ subtyping rules, modifying the rules in Figure 2.13. There are four critical changes made to the subtyping of System F$_{<:}$. Firstly, S-TRANS is removed. It is not syntax directed, and is especially problematic since any algorithm that could be defined would have to guess at the middle type. To make up for the loss of an explicit transitivity rule, S-VAR is extended to include subtyping of any type that is subtyped by the variable's bound. S-REFL is restricted to only type variables (more general subtype reflexivity

$$\Delta \vdash \tau \ <: \ \top \quad \text{(S-Top)} \qquad \Delta \vdash \alpha \ <: \ \alpha \quad \text{(S-Refl)} \qquad \frac{\Delta \vdash \Delta(\alpha) \ <: \ \tau}{\Delta \vdash \alpha \ <: \ \tau} \quad \text{(S-Var)}$$

$$\frac{\Delta \vdash \tau_2 \ <: \ \tau_1 \qquad \Delta \vdash \tau_1' \ <: \ \tau_2'}{\Delta \vdash \tau_1 \to \tau_1' \ <: \ \tau_2 \to \tau_2'} \quad \text{(S-Arr)} \qquad \frac{\Delta, \alpha \leqslant \tau \vdash \tau_1 \ <: \ \tau_2}{\Delta \vdash \forall \alpha \leqslant \tau.\tau_1 \ <: \ \forall \alpha \leqslant \tau.\tau_2} \quad \text{(S-All)}$$

Figure 2.15: Kernel $\mathrm{F}_{<:}$ Subtyping Rules

$$\mathcal{W}_\Delta(\alpha) = 1 + \mathcal{W}_\Delta(\tau) \qquad \textit{where } \tau = \Delta(\alpha)$$

$$\mathcal{W}_\Delta(\top) = 1$$

$$\mathcal{W}_\Delta(\tau_1 \to \tau_2) = 1 + \mathcal{W}_\Delta(\tau_1) + \mathcal{W}_\Delta(\tau_2)$$

$$\mathcal{W}_\Delta(\forall \alpha \leqslant \tau_1.\tau_2) = 1 + \mathcal{W}_{\Delta,\alpha\leqslant\tau_1}(\tau_2)$$

Figure 2.16: Kernel $\mathrm{F}_{<:}$ Depth Measure

can be demonstrated easily from this). Finally, an invariance is enforced on the bound of universally quantified types during subtyping.

This final modification ensures the subtype decidability of Kernel $\mathrm{F}_{<:}$. Kernel $\mathrm{F}_{<:}$ has several properties. It is subtype reflexive, transitive and decidable, as well as being type safe [70]. Reflexivity and transitivity are easily demonstrable via induction on the size of the type in the case of reflexivity and the size of the derivation in the case of transitivity.

In order to prove a relation decidable, it is necessary to demonstrate that a decision procedure, that is both sound and complete, exists for it. Thus, demonstrating that an algorithm exists for determining subtyping of Kernel $\mathrm{F}_{<:}$ types that terminates on all inputs is enough to prove subtyping decidable. The rules defined in Figure 2.15 are syntax directed, and thus constitute a subtype algorithm for Kernel $\mathrm{F}_{<:}$. Secondly, this algorithm is bounded by a finite depth measure (given in Figure 2.16) that is strictly

$$\frac{\Delta \vdash \tau_2 \ <: \ \tau_1 \qquad \Delta, \alpha \ \leqslant \ \top \vdash \tau_1' \ <: \ \tau_2'}{\Delta \vdash \forall(\alpha \leqslant \tau_1).\tau_1' \ <: \ \forall(\alpha \leqslant \tau_2).\tau_2'}$$

Figure 2.17: $F^{\top}_{<:}$ Subtyping for Universally Quantified Types

decreasing during subtyping. Progress and Preservation (and thus Type Safety) are provable by induction on the size of the derivation of typing.

$\mathbf{F}^{\top}_{<:}$

Another decidable variant of System $F_{<:}$ called $F^{\top}_{<:}$ was developed by Castagna and Pierce [27]. Like Kernel $F_{<:}$, $F^{\top}_{<:}$ targets the environmental divergence present in the subtyping of System $F_{<:}$, however instead of enforcing invariance on the argument type as Kernel $F_{<:}$ does, $F^{\top}_{<:}$ allows for variance of argument types, as the original System $F_{<:}$ does. Castagna and Pierce's modified subtype rules are given in Figure 2.17. Unfortunately, typing for $F^{\top}_{<:}$ was discovered to lack minimality [28], posing problems for type checkers.

Proving a Subtyping Decidable

Kernel $F_{<:}$ serves as a good example of how I approach questions of subtype deciability in this thesis. While the subtype relations that I define in the rest of this thesis are often somewhat more complex than that of Kernel $F_{<:}$, the same general approach still applies. For each subtype relation, I start by defining a subtype algorithm $\mathcal{A}$, I demonstrate that $\mathcal{A}$ terminates for all inputs, I finish by demonstrating that for any for any two types τ_1 and τ_2, $\mathcal{A}(\tau_1, \tau_2) \ = \ \texttt{true}$ if and only if τ_1 subtypes τ_2.

2.1.6 Calculus of Constructions

One notable extension of the Simply Typed λ-Calculus is the Calculus of Constructions (CoC), a language that sits on the opposite end of what is called the "λ-Cube" [15]. CoC includes three key extensions:

1. Polymorphism (System F): terms may be dependent on types.

2. Dependent types (λ-P): types may be dependent on terms.

3. Type Operators (System Fω): types may be dependent on other types.

CoC is notable, because it is the basis for Coq the theorem prover used to verify proofs in this thesis. In recent years it has become fairly common practice for formalisms in parts of theoretical computer science to be developed using a theorem prover, a language and tool that allows formal modelling of mathematical definitions and proofs to be verified. Among the most common theorem provers available are Coq [17, 86], Isabelle/HOL [62, 86], Agda [78, 82], Idris [19] and Twelf [68].

Theorem provers are built upon different mathematical foundations. Coq as I have noted is based on the the calculus of constructions [33], an extension of the λ-Calculus with dependent types, polymorphism and type functions [15]. The theoretical basis for Coq is the *Curry-Howard Correspondence* [34, 45, 81], the discovery that there existed an equivalence between proofs and programs. This was first observed by Curry and later extrapolated on by Howard. Thus a type in Coq is equivalent to a proposition and an term of that type is evidence of, or rather a proof of that proposition [71]. Coq is a one of the more popular theorem provers available, and has developed a wide range of resources, approaches and best practices to developing machine verified proofs [71, 29]. Coq is the tool used to verify the proofs of subtype decidability discussed in this thesis.

t	$::=$	**Terms**
	x	*variable*
	$\lambda(x : \tau).t$	*abstraction*
	$t\ t$	*application*
	$\Lambda(\alpha \leqslant \tau).e$	*type abstraction*
	$e\ \tau$	*type application*
	$\textbf{new}\{z \Rightarrow \overline{d}\}$	*object*
	$t.m$	*access*
d	$::=$	**Declarations**
	$m : \tau\ =\ t$	*method*

τ	$::=$	**Types**
	$\top$	*top*
	α	*variable*
	$\tau \rightarrow \tau$	*arrow*
	$\forall(x \leqslant \tau).\tau$	*all*
	$\{\overline{\sigma}\}$	*structure*
σ	$::=$	**Declaration Types**
	$m : \tau$	*method*

Figure 2.18: System $F_{<:}$ with Objects

2.2 Object Oriented Languages

The past few decades of programming languages, in both research and industrial settings, has seem a proliferation of object-oriented languages. Informally, an object is a data type containing a set of members (methods) that are used to interact with it. Some objects may go further, and contain mutable state. Objects provide an analogy to real world systems, where every component is a discrete object that can be communicated with, and maintains some sense of self [8].

In Figure 2.18 I define an minimal extension to System $F_{<:}$ that includes objects. Objects enrich the syntax with a way to create a new object ($\textbf{new}\{z \Rightarrow \overline{d}\}$), and a way to access the members contained within an object ($t.m$). Objects contain a set of members ($\overline{d}$). A member (d) of an object is a declaration, that is typed with declaration a type (σ). An object is typed with a structure type ($\{\overline{\sigma}\}$), a set of declaration types.

2.2.1 Dependent Object Types

The Dependent Object Types (DOT) calculus [12, 13, 14, 75] is a type system built around some of the core langauage features of the Scala programming language [66, 64]: dependent function types, path dependent types, intersection types, and recursive types. DOT was developed with the intention of formalising the central features of Scala, the goal being to demonstrate type safety and provide a foundation for the theory of these concepts. The formalising of this theory was a long road, but has provided designers of object oriented languages a powerful calculus that is both highly expressive and also type safe. For this reason, I will now introduce DOT as both a foundational calculus in the spirit of System $F_{<:}$, but also as the theoretical union of several important features that I will examine in further chapters, and thus a useful tool for introducing these features.

Wyvern, the language this thesis is primarily concerned with is closely related to Scala, and as such the type systems I present in subsequent chapters are closely related to DOT. In fact most of the type systems discussed in the rest of this thesis contain some combination of path dependent types, intersection types, dependent function types and recursive types. For this reason, the design of DOT and the decisions made in its design are incredibly useful and inform much of the design of the Wyvern type system. The existence of a proof of type safety [14, 75] is of particular importance, as it indicates the boundaries of what is and is not safe in a language that features so much complexity.

Before proceeding with a discussion of DOT, I will elaborate on some terminology. There have been several iterations on the DOT calculus, each building upon the last, but in this thesis I will consider only two of the most recent editions: (i) the DOT of Wadlerfest 2016 [14], and (ii) and the DOT presented by Rompf and Amin in 2016 [75]. The DOT of Wadlerfest 2016 featured path dependent types, intersection types, dependent function types and recursive types but did not include subtyping between recursive types. The DOT of Rompf and Amin extended Wadlerfest DOT with subtyping between recursive types along with union types. For the rest of this thesis, to differentiate between these two versions of DOT, I will use “ Wadlerfest DOT” when speaking specifically about the former, and “ DOT 2016” when speaking about the later (even though Wadlerfest DOT is also dated to 2016). In several places throughout the text I may rather refer to “DOT”. In such cases, unless contextually referring to a pre-established version of DOT (to avoid clumsy wording), I will be generally referring to properties that are present in both versions of DOT mentioned above. I will now discuss the different components of the DOT type system.

Path Dependent Types

Up till now I have only alluded to, or discussed informally the language feature that sits at the centre of this thesis: *path dependent types*, most notably captured in the *Abstract Type Members* of Scala. Scala being the most prominent language to feature path dependent types, and DOT being the formal basis of Scala, it is appropriate to introduce path dependent types in the context of DOT and Scala. In Scala, an object may contain type definitions, or type members, in the same way that it might contain fields or methods.

```scala
trait Cell{ this =>
        type E <: Any
        val member : this.E
}
object intCell extends Cell{
        type E = Int
        val member : this.E = 5
}
def returnMember(c : Cell) : c.E = c.member
```

The trait `Cell` defines an object type that contains two members; a type `E` and a value `member` of type `E`. The object `intCell` implements `Cell`, and specifies `E` as `Int`. The type `E` can now be referred to within the object by the notation `this.E`, and from without as in the example `c.E`, much like a field or a method in a typical object oriented language. The type formed by this selection (`this.E` or `c.E`) is referred to as a path dependent type. The "*path*" referring to the receiver (`this.` or `c.`), and the "*dependent*" referring to the fact that the type that is selected depends on the path that is selected on.

Types as members of objects first appeared in BETA [50], and was later suggested as an alternative to Generics in Java [56, 79, 20, 46, 31] under the name of virtual types. Ultimately, virtual types lost out to parametric polymorphism in Java, however they were adopted for Scala [65, 64] as *Abstract Type Members*, and have proven a central component of the Scala type system.

While path dependent types formed a part of the Scala type system, there was no proof of type safety, and much of the surrounding metatheory remained open. Over the past few years, the metatheory has been greatly expanded, with a proof of type safety for DOT at the centre. The type safety proof for DOT has allowed language designers to reason about extensions to DOT and the theory of path dependent types [44, 72].

Dependent Function Types

A dependent function type is a function type that allows the return type to be defined in terms of the argument. A dependent function type $\forall(x : \tau_1).\tau_2$, is interpreted as the type of a function with argument x of type τ_1 and return type τ_2, where τ_2 may be dependent on x.

In a language such as DOT with no explicit bounded polymorphism, dependent function types are necessary for capturing the bounded polymorphism of System $F_{<:}$. As was already mentioned earlier in this section, in DOT, parametric polymorphism is in fact subsumed by path dependent types. In DOT, polymorphic function types

$$\begin{aligned} fold[\mu(\alpha : \tau)] &: \mu(\alpha : \tau) \rightarrow [\alpha \mapsto \mu(\alpha : \tau)]\tau \\ fold^{-1}[\mu(\alpha : \tau)] &: [\alpha \mapsto \mu(\alpha : \tau)]\tau \rightarrow \mu(\alpha : \tau) \end{aligned}$$

Figure 2.19: Isomorphism on Iso-Recursive Types [70]

```
def append(x : {type E},
           l : List[x.E]) : List[x.E] = { ... }
```

Recursive Types

A core aspect of DOT, and this thesis are recursive types. Simply stated, a recursive type is a type that has some reference to itself, and thus may be constructed recursively, referring to itself as a syntactic sub-component. A common example of a recursive type is the definition of `List` in Haskell [57]:

```
data List a = Nil | Cons a (List a)
```

A `List` in Haskell is either an empty list (`Nil`), or a list constructed from a head (of type `a`) and some other list (of type `List a`). `List` references itself in its definition, and is thus recursive.

The theory of recursive types has been reasonably well covered in the literature [70, 26, 23]. The μ-notation is a commonly used representation for recursive types. A type $\mu(\alpha : \tau)$ represents a recursive type, defined using the μ operator, mapping the type variable α to $\mu(\alpha : \tau)$ in τ. Recursive types in the literature are approached in two different ways: equi-recursive or iso-recursive [70]. They differ in how they treat folding and unfolding of the μ operator. Under the equi-recursive approach, the type $\mu(\alpha : \tau)$ is equivalent to $[\alpha \mapsto \mu(\alpha : \tau)]\tau$. Under the iso-recursive approach, $\mu(\alpha : \tau)$ and $[\alpha \mapsto \mu(\alpha : \tau)]\tau$ are treated as different types, but an isomorphism is defined between the two forms (see Figure 2.19).

Cardelli [23] defined a set of subtyping rules for the Amber language that have since become known as the “Amber Rules”. These rules are given below.

$$\frac{\Sigma, \alpha_1 <: \alpha_2 \vdash \tau_1 \; <: \; \tau_2}{\Sigma \vdash \mu(\alpha_1 : \tau_1) \; <: \; \mu(\alpha_2 : \tau_2)} \quad \text{(S-AMBER)}$$

$$\frac{\alpha_1 <: \alpha_2 \in \Sigma}{\Sigma \vdash \alpha_1 \; <: \; \alpha_2} \quad \text{(S-ASSUMPTION)}$$

In order to subtype two iso-recursive types, an assumption context is introduced (Σ), storing subtype assumptions between types. Two types can subtype each other if such an assumption appears within Σ (S-ASSUMPTION). Thus, if two types have a subtype relationship, assuming their recursive references are subtypes of each other, then it follows that their recursive forms have a subtype relationship too (S-AMBER).

In DOT, recursive types differ from those more broadly discussed in the literature: they are similar to equi-recursive types but are quantified over a term, and not a type [10].

$$\mu(z : \tau)$$

The variable z does not quantify the type $\mu(z : \tau)$, but rather the term typed with $\mu(z : \tau)$. A variable x with type $\mu(z : \tau)$, can be thought of as having type $[x/z]\tau$. Similarly, a variable x has type $[x/z]\tau$, also has type $\mu(z : \tau)$. This distinction has implications for how subtyping is defined between two recursive types in DOT. Below I provide the subtype rule for recursive types from DOT 2016 (written here in μ-notation), along with the packing and unpacking rules for variable typing (note: the notation T^x represents a type where the variable x is free in type T).

$$\frac{\Gamma, z : \mu(z : T_1) \vdash T_1 \; <: \; T_2}{\Gamma \vdash \mu(z : T_1) \; <: \; \mu(z : T_2)} \quad \text{(BIND-X)}$$

$$\frac{\Gamma \vdash x \; : \; T^x}{\Gamma \vdash x \; : \; \mu(z : T^z)} \quad \text{(VARPACK)} \qquad \frac{\Gamma \vdash x \; : \; \mu(z : T^z)}{\Gamma \vdash x \; : \; T^x} \quad \text{(VARUNPACK)}$$

While both Wadlerfest DOT and DOT 2016 includes recursive types, only DOT 2016 features subtyping of recursive types. Rapoport et al. 2017 argue that while Wadlerfest DOT does not include subtyping of recursive types, neither does Scala (the target of DOT). The absence of subtyping for recursive types does however reduce the expressiveness of path dependent types in their ability to model Java subtyping in the absence of class inheritance. This will be discussed on in Section 3.2.3.

Intersection Types and Union Types

If we accept the idea of types as sets of values, then it makes sense that we might expect certain concepts from set theory to translate into our logics. Intersection and union types introduce the concepts of set intersection and unions to types. Intuitively, if a value has both type τ_1 and τ_2, then it also has type $\tau_1 \cap \tau_2$ (the intersection of τ_1 and τ_2), and if a value has either type τ_1 or τ_2, then it also has type $\tau_1 \cup \tau_2$ (the union of τ_1 and τ_2). Intersections and unions enable more expressive subtyping, relating to the meet and join of subtyping respectively.

Intersection type feature in several different programming languages and provide several different instances of expressiveness. Languages such as Java and C$^\sharp$ allow for a form of intersection type in that a class may extend multiple other classes, allowing for multiple inheritance.

```
class List<E> implements Comparable<List<E>>, Iterable<E>
```

A `List` is both an instance of `Comparable` and `Iterable`. While this is a form of intersection type, it does not capture the full expressiveness of intersection types. In Java, such intersections are limited to class inheritance declarations, it is not possible to use intersections in an ad-hoc manner. The type `List<Comparable ∩ Serializable>` is not possible. Nor is it possible to use intersection types as function argument types, or bounds on generic types.

DOT (both forms of DOT) includes full intersection types, that is $\tau_1 \cap \tau_2$

is a valid syntactic type form under all circumstances. We are able to rewrite the above `List` multiple inheritance example in DOT as

```
type List[E] <: Comparable[List[E]] ∩ Iterable[E]
```

But we are further able to use the full expressiveness of intersection types in an ad-hoc fashion. The example below uses an intersection of two types to parametrise a function in an ad-hoc fashion.

```
def cast(x : Comparable ∩ Iterable) : Comparable = x
```

The example above still does not capture the full expressiveness of intersection types in DOT. Until now I have liberally used the syntax `List[Int]` to represent a `List` with the generic parameter `Int`, however DOT does not include explicit parametric polymorphism. In DOT, `List[Int]` is in fact syntactic sugar for the type `List ∩ {type Elem = Int}`! Thus, every instance of parametric polymorphism in DOT is in fact an instance of an intersection type.

Union types are the dual of intersection types, where contra-variant subtyping can be thought of as the involution operator. Union types are perhaps not as widely used as intersection types, but they have some useful applications. One common usage is to capture something similar to pattern matching.

```
def printStringOrFile(s : String ∪ File) = {
        if (s instanceOf String)
                s.print
        else if (s instanceOf File)
                s.stream.forEach(_.print)
}
```

The above function can be called on either a `String` or a `File`, and provide specialized behaviour for both. The type `String ∪ File` guarantees the argument to be either of those two types, and as a result, we need not provide some general, default behaviour.

Another pattern is the ability to provide "nullability" to a language, a more controlled manner of handling null values. In languages such as Java, the value `null` can be typed with any type. This causes many software bugs, allowing uninitialized null values to be treated as containing implemented methods and fields. Hoare famously referred to null references as a "billion dollar mistake" [43, 77]. Further, the very presence of `null` in Java is unsound [11]. Union types, along with a reordering of the subtype hierarchy, allow an elegant solution to this problem. Consider a language where `null` is a singleton value of a type `Null` that has no subtypes and is only super typed by $\top$. Under such typing null, values must be accommodated for in the types of expressions.

```
def filterNulls(l : List[Object ∪ Null]) : List[Object] = {
        l.filter(! _ instanceOf Null)
}
```

In the above example, null pointer errors are avoided statically, the type system ensures that the `List` returned by the above method contains no null values. Further, no object operation can be performed on anything in `l` until it has been filtered of null values.

There are several languages that include union types including C, C++ and Ceylon. The unions of C and C++ represent a kind of union type where an initialized value of a union type contains one of a collection of fields. In Ceylon [49], the type $\tau_1|\tau_2$ represents the union of two types τ_1 and τ_2. Wadlerfest DOT does not include union types, however DOT 2016 does.

Wadlerfest DOT

The syntax of Wadlerfest DOT is provided in figure 2.20. A **Type** is either a recursive type ($\mu(x : \tau)$), a dependent function type ($\forall(x : \tau_1).\tau_2$), a structure with either a field ($\{f : \tau\}$) or type ($\{L : \tau_1 \ldots \tau_2\}$) definition, a selection type ($x.L$), an intersection type ($\tau_1 \wedge \tau_2$), the top type ($\top$), or bottom type ($\bot$). A **Term** is either a variable, an object, an abstraction, a field selection,

s, t, u	$::=$	**Term**
	x	*variable*
	v	*value*
	$x.f$	*selection*
	$\texttt{let}\ x = t\ \texttt{in}\ u$	*let binding*
v	$::=$	**Value**
	$\nu(x : \tau).d$	*object*
	$\lambda(x : \tau).t$	*abstraction*
d	$::=$	**Definition**
	$\{f = t\}$	*field definition*
	$\{L = \tau\}$	*type definition*
	$d \cap d$	*aggregate definition*

τ	$::=$	**Type**
	$\mu(x : \tau)$	*recursive type*
	$\forall(x : \tau).\tau$	*dependent function*
	$\{f : \tau\}$	*field definition*
	$\{L : \tau \dots \tau\}$	*type definition*
	$x.L$	*type selection*
	$\tau \cap \tau$	*intersection*
	$\top$	*top*
	$\bot$	*bottom*

Figure 2.20: Wadlerfest DOT Syntax

or a let binding.

The strength of the DOT type system is in the economy of concepts afforded by the combination of a relatively small variety of types. The relatively small array of types in Figure 2.20 many different common type patterns can be employed, including structural subtyping, nominal subtyping, multiple inheritance style type hierarchies, parametric polymorphism, and f-bounded polymorphism.

The “Bad Bounds” Problem

The so called “bad bounds” problem refers to a problem at the centre of the long search for a type safety proof for DOT. An earlier version of DOT, μDOT [13], did not include subtype transitivity as an explicit rule in subtyping. Rather it was hoped that transitivity would be a latent property of subtyping that could be proven. Amin et al. demonstrated that the properties of subtype transitivity and environment narrowing in μDOT were not present alongside language features such as intersection types or type refinements. Informally environment narrowing refers to the preservation of

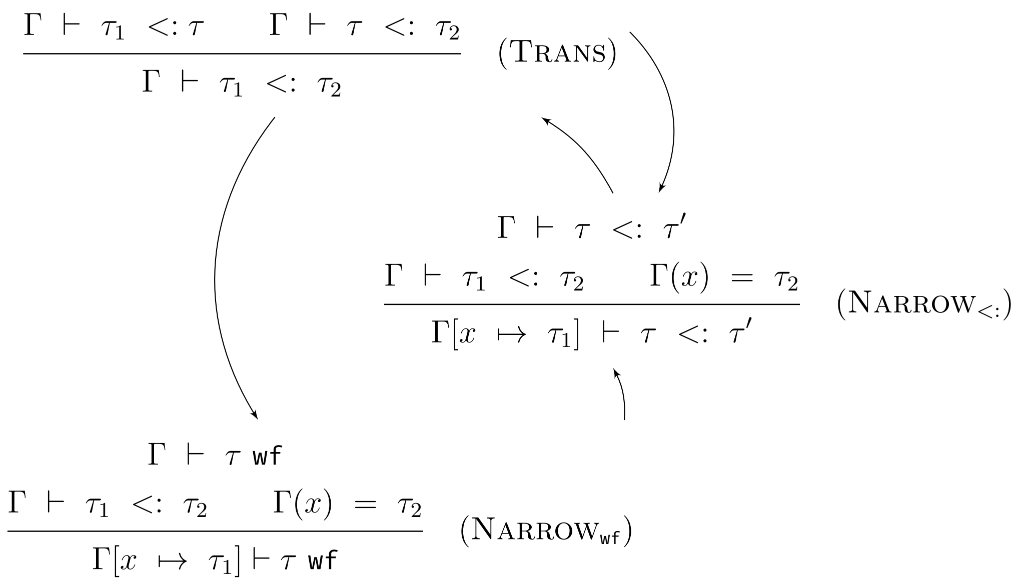
$\dfrac{\Gamma \vdash \tau_1 <: \tau \qquad \Gamma \vdash \tau <: \tau_2}{\Gamma \vdash \tau_1 <: \tau_2}$ (Trans)
$\dfrac{\Gamma \vdash \tau <: \tau' \quad \Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma(x) = \tau_2}{\Gamma[x \mapsto \tau_1] \vdash \tau <: \tau'}$ (Narrow$_{<:}$)
$\dfrac{\Gamma \vdash \tau \ \mathtt{wf} \quad \Gamma \vdash \tau_1 <: \tau_2 \qquad \Gamma(x) = \tau_2}{\Gamma[x \mapsto \tau_1] \vdash \tau \ \mathtt{wf}}$ (Narrow$_{\mathtt{wf}}$)

Figure 2.21: Mutual Dependency between Properties of μDOT

language properties after types in the environment are narrowed more specific types. In μDOT, properties of typing are mutually dependent: the preservation of both subtyping and well-formedness by narrowing, and subtype transitivity are all mutually dependent. The dependencies between these properties is shown in Figure 2.21. These dependencies are shown in Figure 2.21. Subtype transitivity (Trans) is mutually dependent on environment narrowing preserving subtyping (Narrow$_{<:}$). Such a dependency already complicates the proof of transitivity, but subtype transitivity is also dependent on environment narrowing preserving well-formedness of types.

As part of well-formedness of types, μDOT requires the bounds of type members to observe a subtype relationship. i.e. for any type member $L : \tau_1 \ldots \tau_2$, τ_1 must subtype τ_2. This is a requirement for the derivation of subtype transitivity, particularly the case $\tau <: x.L <: \tau'$. If the lower bound of $x.L$ does not subtype the upper bound, then transitivity cannot be expected to hold generally. If, for instance $x.L$ had definition

$L\ :\ \top \ldots \bot$, then as all types subtype $\top$ and $\bot$ subtypes all types, subtype transitivity would imply that any type could subtype any other type, clearly an unsafe property. Unfortunately the presence of intersection types is enough to construct this very example. Consider the variable x of type $\{A\ :\top \ldots \top\} \cap \{A\ :\bot \ldots \bot\}$. The lower bound of $x.A$ is effectively $\top$, and the upper bound $\bot$.

In Wadlerfest DOT, Amin et al. demonstrated that the well-formedness requirement was in fact too strong, and ill-formed bounds could be defined safely as long as it could be ensured that type members of objects refer to specific types and not types with arbitrary bounds. Under such this relaxed notion of well-formedness, $\textsc{Narrow}_{\texttt{wf}}$ can be removed from the dependencies of Figure 2.21. As a consequence of this key property of Wadlerfest DOT, subtyping can be safely admitted, meaning type members may be defined with arbitrary bounds, and programmers may construct bespoke subtype lattices. Type safety is ensured by requiring concrete objects to act as a witness to the bounds of a type member before it code that is quantified by that member may be evaluated. As a demonstration, consider the following Wadlerfest DOT function definition.

```
def crazyCast(x : {type X : String ... Int},
              s : x.X) : Int = { s }
```

The function `crazyCast` takes as a parameter some object `x` that defines a member `X` lower bounded by `String` and upper bounded by `Int`, along with some object of type `x.X`. Clearly any `String` may be supplied for `s`, allowing for the seemingly crazy cast of `String` to `Int`. What makes such a function safe is the requirement that some object must also be supplied for `x`. That object must have a type member `X` that refers to a specific type, say `T`, such that `String` <: `T` <: `Int`. The ability to define `T` acts as a witness to the fact that `String` subtypes `Int`. Since no such type exists, `crazyCast` may never be called.

$$\frac{\Gamma \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2}{\Gamma \vdash \{\overline{\sigma}_1\} \ <: \ \{\overline{\sigma}_2\}} \ (\text{S-S}\textsc{truct}) \qquad \frac{\Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash m : \tau_1 \ <: \ m : \tau_2} \ (\text{S-M}\textsc{eth})$$

Figure 2.22: Structural Subtyping

2.2.2 Structural vs Nominal Subtyping

Subtyping in object oriented languages usually adheres to one of two disciplines (or perhaps both): structural or nominal. The subtyping associated with the types in Figure 2.18 is structural, that is two structure types have a subtype relationship if the structure of the smaller type adheres to the structure of the larger type. I formally state this subtyping in Figure 2.22. The subtyping of Figure 2.22 resembles most of the subtypings defined in the rest of this thesis. Structural subtyping is present in languages such as Scala [66], OCaml [35] and Modula-3 [22, 61]. Structural subyping is based on behaviour. If $\{\overline{\sigma}_1\} \ <: \ \{\overline{\sigma}_2\}$ then we know that if a method of name m occurs in $\{\overline{\sigma}_2\}$, then there must be some analogous method of comparable type in $\{\overline{\sigma}_1\}$. Consider the Scala example, `DBConnection`, below.

```
object mySQLConnection {
        var url : MySQLUrl
        def commit : Unit = {...}
}
def doCommit(c : {def commit : Unit}) : Unit = {
        c.commit;
}
def main() : Unit = {
        doCommit(mySQLConnection);
}
```

A bespoke connection object (`mySQLConnection`) is defined, containing a field (`MySQLUrl`) and method (`commit`). The method `doCommit` specifies that that it merely requires an object containing a `commit` method, and does not specify anything else about the argument. Subsequently the call

```
doCommit(mySQLConnection)
```

type checks because `mySQLConnection` contains a `commit` method.

Languages such as Java [38] and C$^\sharp$ [3] take a different approach to object subtyping, based around classes. In Java a *Class* is a template that combines several complex aspects of the language and type system. A class is responsible for object initialization, implementation of methods, naming of types, inheritance of methods and defining the type hierarchy. While surely complex, in this thesis, I am primarily concerned with the implications that classes have for types. Objects initialized by a class C are labelled as having type C. Java also incorporates class inheritance as a mechanism for both code reuse (when initializing objects) and as the basis for defining a class hierarchy and from that, subtyping.

Consider the code snippet below, a variant on the Scala example.

```
class DBConnection {
        DBConnection(String url){ ... }
        void commit(){ ... }
}
class SQLConnection extends DBConnection{
        SQLConnection(Sring url){ ... }
        ...
}
class AdHocConnection {
        commit(){ ... }
}
void doCommit(DBConnection c)
        c.commit();
```

```
14 }
15 void main(){
16         SQLConnection c = new SQLConnection( ... );
17         AdHocConnection c' = new AdHocConnection();
18         doCommit(c);
19         doCommit(c'); /* type error */
20 }
```

Two classes are declared, a general database connection `DBConnection` and a SQL database connection `SQLConnection`. The **`extends`** keyword is the basic building block of class inheritance in Java. `SQLConnection` not only inherits all the methods declared in `DBConnection`, but is also declared as a subtype of `DBConnection`. Subtyping is thus an explicit property that is defined between classes. The call to `doCommit` on line `18` type checks because of the explicit subtype relationship, however the call on line `19` does not. Even though `AdHocConnection` contains a commit method, it does not extend `DBConnection`. Subtyping that is defined around these kinds of explicit relationships is referred to as *Nominal Subtyping*.

Nominal and structural subtyping represent different philosophies in subtyping. Nominal subtyping relies on the intent of a type while structural subtyping defines subtyping from a purely behavioural perspective. A language such as Scala in fact provides facilities for both nominal and structural subtyping. While structural subtyping is included in the manner seen in Scala example, it is also possible to provide some form nominality in two ways: class declarations and type declarations. Scala does include classes and thus a class based form of nominal subtyping.

2.2.3 Java

Java [9] is an object oriented programming language that sees widespread use in industry. This widespread use and popularity mean that Java is a useful reference point for demonstrating a minimal level of expressiveness, even if it is not as expressive as a language such as Scala. Throughout this thesis, I use Java and encodings from Java to demonstrate different properties of the type systems defined. While it is not the only reason I provide an encoding of Java, the primary purpose in this thesis is to demonstrate that the calculi presented are as at least as expressive as some form of Java.

Java includes many of the features of object-oriented languages, and recently has been extended with features from functional languages too. The features that this thesis is most concerned with however is the subtyping of Java and Java Generics. Java is class based language with nominal subtyping built around a class inheritance hierarchy. Both classes and methods may include generic type parameters. Generic types in Java may be either covariant or contra-variant. Variance in Java Generics is use-site as opposed to the declaration site variance of parametric polymorphism in Scala. As a popular language, Java has had much research devoted to it in the literature, and certain aspects of its type system have been vigorously investigated, much more so than a language such as Scala.

Featherweight Java and Featherweight Generic Java

Igarashi et al. developed a minimal core type system for Java called Featherweight Java (FJ) in [47]. They extended the core FJ calculus with Generics as Featherweight Generic Java (FGJ). Igarashi et al. was then able to prove both FJ and FGJ sound. Given the complexity with formalizing a full modern language, a minimal core has a clear advantage over a full type system when proving formal properties, specifically that complexity is removed thus making proofs that much more tractable. The downside to proving properties on a minimal core means that the results of such a proof are not generally applicable to the wider language especially if unforeseen interactions exist between different language features. The Java type system for instance has since been proven unsound [11] in a manner that many might not have expected.

Even though properties proven of a minimal core is not necessarily a guarantee that those properties are held by the wider language, the advantage of proofs on a minimal core can nonetheless remove specific fears about the potential for problematic interactions between related language properties. FGJ as an example demonstrated that Java Generics was safe for a core set of programs, and has been the basis of other work in investigating how the core Java type system interacts with other language features [54]. It is also notable that the features that introduced unsoundness to Java were present in Java 5, released more than a decade before the publication of the findings of unsoundness, implying that this particular instance of unsoundness had marginal effect on the Java community.

2.2.4 Decidability of Subtyping in Java

Kennedy and Pierce [48] questioned the decidability of Java's Generics, and postulated that it is decidable as it did not contain multiple instantiation inheritance (a property that they identified as a key determinant of undecidability in generics). Unfortunately this was not the case. The question of decidability of subtyping Java's wildcards remained. Wehr and Thiemann [83] demonstrated that subtyping of bounded existential types in JavaGI [84] was undecidable. Java wildcards represent a restricted form of the existential types in JavaGI however, and were not necessarily implicated by this proof. Java Generics were ultimately proven Turing Complete by Grigore in 2017 [40], and thus undecidable. Grigore encoded Turing machines into a fragment of Java's subtyping that was limited to Java Generics and contravariant type parameters. It is useful to have the result of undecidability, however the literature before Grigore already contained several instances of proposed decidable subsets of Java Generics, indicating a strong suspicion of undecidability.

Zhang et al. [87] propose a variation on Java Generics that allows constraints on generics to be defined in a manner reminiscent of type classes in Haskell. Greenman et al. [39] demonstrated that such separation, when used to restrict the use of Shapes from parametrizing recursive inheritance definitions, ensured decidability of subtyping. What Greenman et al. found, via a survey of Java code, was that this was a separation that was already being observed by Java programmers, implying that such a restriction could be applied to Java with minimal effect on existing code bases. What is more, the syntactic nature of the restriction means that it can be introduced to existing Java type checkers with minimal modification. The advantages that the Material/Shape separation represent make it a particularly promising approach for this thesis, and as such I will go into more detail on the specifics of the work by Greenman et al..

Material/Shape Separation

Greenman et al. [39], defined a syntactic subset of Java that ensures decidable subtyping. That is, if a Java program follows some simple syntactic rules, subtyping for that program is guaranteed to be decidable. These syntactic rules take the form of a separation on types. Types are classified as either *Materials*, that may be freely used, or *Shapes*, that are restricted from use in generic parameters on generic classes.

Informally, a Material is a concrete datatype used to type terms within a program. A Shape refers to abstract types that are used to specify a program. In Java, the distinction between a Material and a Shape loosely lines up with the distinction between concrete classes, and interfaces. That is, Materials are types that might be instantiated (e.g. `ArrayList` or `Integer`), while Shapes are used to specify a program, i.e. as bounds on generic types, argument types, or extended types in class definitions (e.g. `Iterable` or `Comparable`).

Defining Materials and Shapes

Thus far, all we have is a fairly vague definition of Materials and Shapes. Greenman et al. formalized the definition and their separation using cycle detection on type definitions. They use class definitions to construct type usage graphs, where any cycles represent a recursive type definition. Since

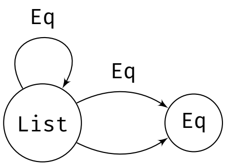

Figure 2.23: Type Usage Graph of `List`

the recursion in Java class definitions arises from the inheritance declaration, they perform edge labeling on the graph to identify type inheritance. As an example, consider the class definition below.

```
interface Eq<T>
class List<E> implements Eq<List<Eq<E>>>
```

`List` is defined in terms of itself, and is is thus an example of a recursive type definition in Java. This recursive definition is defined using two language features, the inheritance declaration (... `implements Eq`), and generic class parameters (`Eq<` ... `>`).

Using the graph construction rules of Greenman et al., I provide a type usage graph for `List` and `Eq` in Figure 2.23. Vertices in the type usage graph represent type names, and edges represent usages in those type definitions. i.e. `Eq` is used in the definition of `List`, thus there is an edge from `List` to `Eq`. For a particular type definition, edges from the defined type to types used as parameters on the extended type are labeled with the name of the extended type. i.e. `List` is defined as an extension of `Eq` parameterized with `List<Eq<E>>`, thus there is an edge from `List` to both `Eq` and `List`, and those edges are labeled with `Eq`. Finally, there is an unlabeled edge from `List` to `Eq` capturing the the inheritance (or extension) relationship `List` has with `Eq`.

The labeling of edges in the type usage graph is important. Labeling of edges is used to identify recursion in type definitions (represented by cycles in the usage graph). Note, in Java it is not possible to have an entirely unlabeled cycle, as unlabeled edges imply an inheritance relationship, and thus an unlabeled cycle would require cyclic inheritance, something that is not allowed in Java. Recursion in Java occurs in the parameters on the inherited type (e.g. the parameters on `Eq` in the case of `List`), thus any subtyping that results in recursion must be done via the extended type (i.e. recursive subtyping of `List` must occur via `Eq`). This is in fact a result of the subtype nominality of Java. While not addressed by the work of Greenman et al., it is important to note that this would not be true of a structural type system. If Java had general structural subtyping, there would be no assurance that recursive subtyping of `List` would include `Eq`. This distinction between nominal and structural subtyping in relation to the Material/Shape Separation, is important for the work described in this thesis.

As I have already mentioned, recursion is captured in the type usage graph by type cycles. Further, the labels on edges represent inheritance declarations. Thus, an edge on a cycle, labeled with some class name `T`, represents a recursive definition for a type defined as inheriting from `T`. The graph in Figure 2.23 features only one cycle, from `List` back to itself. The only edge in the cycle is labeled with `Eq`. Formally, Greenman et al. provide the following definition of the Material/Shape separation:

Definition 2.2.1. *Let $\mathcal{M}$ be all classes/interfaces used as type arguments. For some set $\mathcal{S}$ of classes/interfaces such that removing all edges labeled with an element of $\mathcal{S}$ from the usage graph results in an acyclic graph, $\mathcal{M}$ and $\mathcal{S}$ are disjoint.*

Put another way, all type cycles (i.e. recursive definitions) in a program must have at least one edge that is labeled with a Shape. This means that the classification of Material/Shape is not generally unique for any program, as there may be more than one labeled edge in a cycle, or the set of Shapes may include types that do not label any edges. In the above `List`/`Eq` example, the only type that fulfills Definition 2.2.1 is `Eq` since it is the only label on the only cycle. Thus, for the `List`/`Eq` example, there are two possible Shape classifications: {`Eq`} or {`Eq`, `List`}.

If we pick the first classification ($\mathcal{M} = \{$`List`$\}$, $\mathcal{S} = \{$`Eq`$\}$), this seems to fit with the intuition that we started with: Materials are concrete types, while Shapes are abstractions. We might want to create an instance of `List`, but we would never instantiate `Eq`. On the other hand, we would expect `Eq` to be used to specify a program, for example, as a argument type to a method, or as a bound on a generic method.

Restricting Shape Usage

As mentioned earlier, the separation places a single, simple restriction on the use of Shapes in Java: Shapes may not be used in type parameters on classes. Greenman et al. importantly demonstrated, via a survey of Java code, that this separation represented a latent rule already followed by Java programmers. For most Java programs that they included in their survey (with a small number of minor exceptions), some classification of Materials and Shapes existed such that the program adhered to the Material/Shape separation. Thus, they found programmers generally write programs that instantiate types such as `List<Integer>` and not `List<Comparable>`. The fact that programmers already follow this distinction is important because it means that (i) such a separation in Java is more likely to be backward compatible with existing Java code bases, and (ii) that while the separation represents a clear reduction in expressiveness, it does not exclude programs that programmers actually write. It is important to state that Greenman et al. did find a small number of Java programs that did not obey the Material/Shape separation, however they did find that those programs could easily be rewritten to follow a Material/Shape separation, or exhibited obvious errors.

Nominality with Path Dependent Types

Languages with path dependent types provide a form of nominality through the ability to define a type in an abstract manner. Abstract is often an overloaded word, but in this case it is meant to refer to types that are not defined as a specific type, but rather defined with some upper bound. To illustrate this nominality, I now provide an example that reoccurs often in the following chapters.

```
type ListAPI = { self =>
        type Equatable <: ⊤{ z =>
                type E >: ⊥
                def equals : E -> bool
        }
        type List <: self.Equatable{ z =>
                type E = self.List{ type T = z.T }
                type T <: ⊤
        }
        def nil : self.List{type T = ⊥}
        def cons : (x : {type Elem <: ⊤},
                    e : x.Elem,
                    l : List{type T <: x.Elem}) ->
                    self.List{type T <: x.Elem}
        def peek : (x : {type Elem <: ⊤},
                    l : List{type T <: x.Elem}) ->
                    x.Elem
        def pop  : (x : {type Elem <: ⊤},
                    l : List{type T <: x.Elem}) ->
                    self.List{type T <: x.Elem}
}
val myListAPI : ListAPI
```

The above example is derived from Rompf and Amin, and depicts an interface for a module called `ListAPI`. Within `ListAPI`, two types are defined, `Equatable` and `List`, along with two constructors for `List`, `nil` and `cons`. While it may not look it initially, the types `Equatable` and `List` exhibit nominal subtyping. As I discussed earlier in this section, nominal subtyping is a form of subtyping that is that has been explicitly declared. An example of this is that `List` subtypes `Equatable` only because it is defined as extending `Equatable`. Because `Equatable` is defined using the upper bound operator (`<:`), the lower bound of `Equatable` is $\bot$, thus the only types that can subtype it are ones that have been explicitly declared so.

This provides some powerful expressiveness, especially in the declaration of modules. The `List` (and `Equatable`) type in the example above is an abstract data type [32], similar to those of Haskell [57]. The type types `List` and `Equatable` are defined, along with a set of operations that have been defined for those types. Given an instance of `ListAPI`, the client is constrained to constructing `List`s using either `nil` or `cons`, and inspecting or removing from `List`s using either `peek` or `pop`.

This is however, a slightly different (or perhaps laxer) notion of nominality to that of Java. `List` can always subtype another type structurally. For instance, `List` subtypes the structure type `{ type E <:` $\top$ `}`. The details of this variation of nominality is very important to this thesis, and is discussed further in Chapter 3.

2.3 Wyvern

Wyvern [52, 51, 63, 1, 67] is a pure object oriented language with first class modules and a focus on high-assurance programming. Wyvern includes path dependent types, leveraging their expressiveness to support nominality and ML-style modules. Wyvern also provides support for concepts from functional programming languages such as lambda expressions. To provide reliable tools for programmers, it is critical that type checking in Wyvern be decidable.

In Figure 2.24 I provide the syntax for Featherweight Wyvern [63], a minimal core calculus for Wyvern in the spirit of Featherweight Java [47]. Featherweight Wyvern contains many ideas that are similar to DOT, it is an object oriented language with functional components, recursive types, and

t	$::=$		**Term**
	x		*variable*
	$\lambda x : \tau . t$		*abstraction*
	$t\ t$		*application*
	$\texttt{new}\ \{\overline{d}\}$		*new*
	$t.f$		*field access*
	$t.f\ =\ t$		*assignment*
	$t.m$		*method access*
τ	$::=$		**Type**
	T		*name*
	$\tau \to \tau$		*arrow*

d	$::=$		**Declaration**
	$\texttt{var}\ f\ :\ \tau\ =\ t$		*field*
	$\texttt{meth}\ m\ :\ \tau\ =\ t$		*method*
	$\texttt{type}\ T\ =\ \{\overline{\tau_d}\}$		*type*
τ_d	$::=$		**Meth Decl. Type**
	$\texttt{meth}\ m\ :\ \tau$		*method*
σ	$::=$		**Declaration Type**
	$\texttt{var}\ f\ :\ \tau$		*field*
	$\texttt{type}\ T\ =\ \{\tau_d\}$		*type*
	τ_d		*method*

Figure 2.24: Featherweight Wyvern Syntax of Nistor et al.

type members. Featherweight Wyvern does not include intersection types, dependent function types or bounds on type members, and subsequently does not leverage the full expressive power of path dependent types. Notably, the exclusion of these types means that the nominality of DOT is lost, along with some of the key encodings of parametric polymorphism. Nistor et al. 2013 also provide an extension to Featherweight Wyvern that includes classes, but note that this extension is a merging of the concept of type and value that can be translated back to Featherweight Wyvern.

2.3.1 Modules for Wyvern

One of the primary motivations for path dependent types in Wyvern is the modelling of modules in the style of ML [58, 41]. Path dependent types can be used to express module signatures from Standard ML. Below I provide an encoding of the `Protocol` signature from the FoxNet project [18].

```
type Protocol {this =>
        type Address
        type Data
        def send(a:Address, d:Data)
        def receive(a:Address):Data
}
module IP : Protocol { ... }
module TCP : Protocol { ... }
```

As in ML (and the `ListAPI` example from Section 2.2.2), it is possible to declare `Address` and `Data` as abstract.

Chapter 3

Issues with Subtype Decidability

In this chapter I discuss the issues present in deciding subtyping in the Wyvern programming language (see 2.3). The chapter is organised as follows.

- **Section 3.1**: I design a core calculus based on the Featherweight Wyvern of Section 2.3 called Wyv_{core}. I introduce the syntax and semantics for this core type system, and discuss expressiveness in Wyv_{core} and some difficulties in deriving a subtype algorithm for Wyv_{core}.

- **Section 3.2**: I demonstrate three ways of proving subtype undecidability in Wyv_{core}, and discuss some of the mechanisms that allow for undecidability of subtyping in Wyv_{core}.

There is an important point to keep in mind while reading this chapter. While the central focus of the chapter is subtype decidability, that is not the only concern. It is also important to consider the expressiveness of any final calculus and the type safety of that calculus.

- **Expressiveness** is often a hard quality to measure, and in this thesis is often measured by presenting instances of known expressiveness in the form of patterns, and discussing the effect different restrictions have on such examples. Where possible, I make stronger claims of expressiveness by demonstrating the power of a calculus to encode another, thereby subsuming its expressive power.

- **Type Safety** is a more concrete goal than expressiveness, and this thesis often leans on type safety argument of DOT [14, 75] as it provides invaluable insight into the nature of subtyping in similar type systems.

With these two factors in mind, in both the design of Wyv_{core} and its subsequent variants I discuss the implications that different restrictions that seek to provide for subtype decidability have on expressiveness and type safety.

3.1 A Core Wyvern Type System

In this section I design and present a core Wyvern type system: Wyv_{core}. In designing Wyv_{core}, I begin this section by presenting a type system for Basic Wyvern (Section 3.1.1). Basic Wyvern is not Wyv_{core}, but rather an intermediate type system between the Wyvern of Section 2.3 and Wyv_{core}. I subsequently discuss Basic Wyvern and some of its properties that fall in, two categories: syntax directedness and expressiveness (Section 3.1.2). I finally end by presenting Wyv_{core} and a discussion of some important properties of subtyping in Wyv_{core} (Section 3.1.3). Wyv_{core} is the base type system for the rest of the Chapter, and for the extensions introduced in Chapters 5, 6, and 7. Subtyping in Wyv_{core} is not decidable for several reasons, but it forms a starting point for the subsequent variants that are decidable, and a basis for understanding the issues of decidability.

3.1.1 Starting with a Basic Wyvern Type System

Basic Wyvern can be thought of as a type system that occupies a position in the design space somewhere between the Featherweight Wyvern of Figure 2.24 and Wyv_{core}. Basic Wyvern includes several language features that empower path dependent types that Featherweight Wyvern lacks. I use this basic type system as a vehicle for discussing some initial decisions made in the design of Wyv_{core}. These decisions fall into two categories: syntax directedness and expressiveness.

What is missing from Featherweight Wyvern?

There are two fundamental concepts that are missing from the Featherweight Wyvern of Figure 2.24: recursive types and bounded type members. *Recursive types* (see 2.2.1) allow types to be defined recursively, in Wyvern this means types can contain references to any other types defined within the same environment. Technically Featherweight Wyvern does have a degree of recursion in that types can use any type name defined within the environment, but this is not included explicitly as a syntactic form. *Bounded type members* provide key expressiveness among other things capturing bounded polymorphism. The type members of Figure 2.24 are in fact merely type aliases, alternate names provided for types within a specific environment.

In Basic Wyvern I make two key extensions to Featherweight Wyvern: (i) a recursive type form $\mu(z : \tau)$, where z is a recursive reference that may be used within τ, and (ii) a bounded form for type members $L \ : \tau_1 \ldots \tau_2$, where τ_1 is the lower bound of type definition L, and τ_2 is the upper bound.

Basic Wyvern

I give the syntax, along with the variable typing and subtyping relations for Basic Wyvern in Figures 3.1, 3.2, and 3.3 respectively. A **Type** (τ) is either a recursive type ($\mu(z : \tau)$), a dependent function type ($\forall(x : \tau_1).\tau_2$), a selection type ($x.L$), a structure type ($\{\overline{\sigma}\}$) that contains a set of declaration types ($\overline{\sigma}$), the top type $\top$ or the bottom type $\bot$. A **Declaration Type** (σ) is either a type member with an upper and a lower bound ($L \ : \tau \ldots \tau$) or a value member ($l : \tau$). As we are only currently concerned with the metatheory of subtyping, we restrict terms to **variables** (x, y, z) only.

$$\begin{array}{lll r}
x, y, z & & & \textbf{Variable} \\
\tau & ::= & & \textbf{Type} \\
 & \mu(z : \tau) & & \textit{recursive type} \\
 & \forall(x : \tau).\tau & & \textit{dependent function type} \\
 & x.L & & \textit{type selection} \\
 & \{\overline{\sigma}\} & & \textit{structure type} \\
 & \top & & \textit{top} \\
 & \bot & & \textit{bottom} \\
\sigma & ::= & & \textbf{Declaration Type} \\
 & L\ : \tau \ldots \tau & & \textit{type} \\
 & l : \tau & & \textit{value} \\
\Gamma & ::= & & \textbf{Environment} \\
 & \emptyset & & \\
 & \Gamma, x : \tau & &
\end{array}$$

Figure 3.1: Basic Wyvern Syntax

$$\boxed{\Gamma \vdash x\ :\ \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x\ :\ \tau}\ \text{(T-Var)}$$

$$\frac{\Gamma \vdash x\ :\ \mu(z : \tau)}{\Gamma \vdash x\ :\ [x/z]\tau}\ \text{(T-Rec)}$$

$$\frac{\Gamma \vdash x\ :\ \tau \qquad \Gamma \vdash \tau\ <:\ \tau'}{\Gamma \vdash x\ :\ \tau'}\ \text{(T-Sub)}$$

Figure 3.2: Basic Wyvern Typing

$$\boxed{\Gamma \vdash \tau_1 <: \tau_2,\ \overline{\sigma}_1 <: \overline{\sigma}_2,\ \sigma_1 <: \sigma_2}$$

$$\Gamma \vdash x.L\ <:\ x.L\ \text{(S-Refl)} \qquad \frac{\Gamma \vdash \tau_1\ <:\ \tau_2 \qquad \Gamma \vdash \tau_2\ <:\ \tau_3}{\Gamma \vdash \tau_1\ <:\ \tau_3}\ \text{(S-Trans)}$$

$$\Gamma \vdash \bot <: \tau\ \text{(S-Bottom)} \qquad \Gamma \vdash \tau <: \top\ \text{(S-Top)}$$

$$\frac{\Gamma \vdash x : \{L\ : \_ \ldots \tau\}}{\Gamma \vdash x.L\ <:\ \tau}\ \text{(S-Upper)} \qquad \frac{\Gamma \vdash x : \{L\ : \tau \ldots \_\}}{\Gamma \vdash \tau\ <:\ x.L}\ \text{(S-Lower)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau_2\ <: \tau_1 \\ \Gamma, x : \tau_2 \vdash \tau_1'\ <:\ \tau_2'\end{array}}{\Gamma \vdash \forall(x : \tau_1).\tau_1'\ <:\ \forall(x : \tau_2).\tau_2'}\ \text{(S-All)} \qquad \frac{\Gamma, z : \mu(z : \tau_1) \vdash \tau_1 <: \tau_2}{\Gamma \vdash \mu(z : \tau_1)\ <:\ \mu(z : \tau_2)}\ \text{(S-Rec)}$$

$$\frac{\Gamma \vdash \overline{\sigma_1} <: \overline{\sigma_2}}{\Gamma \vdash \{\overline{\sigma_1}\}\ <:\ \{\overline{\sigma_2}\}}\ \text{(S-Str)} \qquad \frac{\forall\ \sigma_2\ \in\ \overline{\sigma}_2,\ \exists\ \sigma_1\ \in\ \overline{\sigma}_1,\ \Gamma \vdash \sigma_1 <: \sigma_2}{\Gamma \vdash \overline{\sigma_1}\ <:\ \overline{\sigma_2}}\ \text{(S-Decls)}$$

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \qquad \Gamma \vdash \tau_1' <: \tau_2'}{\Gamma \vdash L\ : \tau_1 \ldots \tau_1'\ <:\ L\ : \tau_2 \ldots \tau_2'}\ \text{(S-Type)} \qquad \frac{\Gamma \vdash \tau_1 <: \tau_2}{\Gamma \vdash l : \tau_1\ <:\ l : \tau_2}\ \text{(S-Val)}$$

Figure 3.3: Basic Wyvern Subtyping

Note: in the formal statement of Basic Wyvern, I sometimes use an underscore (_) in place of certain syntactic expressions. I use this short-hand here and in other type systems in this thesis in place of meta-variables that are contextually unimportant. In the case of S-Upper (described below) for example, the lower bound of L has no relevance to the rule.

Typing of variables ($\Gamma \vdash x \ : \ \tau$) is defined in Figure 3.2. A variable x in an environment Γ has type τ if $\Gamma(x) = \tau$ (T-Var). If x has some recursive type $\mu(x : \tau)$, then it is also judged as having type $[x/z]\tau$, the "unpacking" of the recursive type, where the recursive reference is substituted for x (T-Rec). T-Rec is similar to the VarUnpack rule of DOT 2016. Typing in Basic Wyvern also includes subsumption, where if x has type τ, it is also judged to have any supertype of τ (T-Sub). Subsumption plays an important role in subtyping, allowing (through width subtyping) a variable x of type $\{\overline{\sigma}\}$ to be typed as $\{\sigma\}$ if $\sigma \in \overline{\sigma}$. Similarly, if a variable x has type $y.L$, then through subsumption, it is also true that x has type τ, where τ is the upper bound of $y.L$.

Subtyping is defined in Figure 3.3. $\Gamma \vdash \tau_1 \ <: \ \tau_2$ formalizes the notion that τ_1 subtypes τ_2 in environment Γ. Subtyping is reflexive for selection types (S-Refl). Subtyping is explicitly transitive (S-Trans). Subtyping is bounded above by $\top$ (S-Top) and below by $\bot$ (S-Bottom). Subtyping of a selection type is defined according to its upper (S-Upper) and lower (S-Lower) bounds. Subtyping of dependent function types is contra-variant (see Section 2.1.4) with respect to the argument types and covariant with respect to the return types (S-All). Subtyping of recursive types is defined by the subtyping of the unpacked type, introducing the recursive variable to the environment (S-Rec). Subtyping of structure types is defined in relation to the declaration types (S-Str). A set of declaration types $\overline{\sigma}_1$ subtypes another set of declaration types $\overline{\sigma}_2$ if ever declaration type in $\overline{\sigma}_2$ is subtyped by some declaration type in $\overline{\sigma}_2$ (S-Decls). Type declarations are subtyped contra-variantly with respect to their lower bounds and covariantly with respect to their upper bounds (S-Type). Value declarations are subtyped covariantly with respect to their types (S-Val).

3.1.2 Properties of Basic Wyvern

While the type syntax is fairly simple, subtyping is already relatively complex and presents several issues for decidability. Firstly, subtyping is not syntax directed, that is, the syntactic form of the the types being compared does not necessarily inform which rule to apply during subtyping. Thus, the rules do not themselves define an algorithm for subtyping. The presence of an explicit subtype transitivity rule (S-Trans) means there is no way to determine which rule to use since there is no way to identify the middle type (τ_2). Secondly, the subtyping of dependent function types (S-All) coupled with type selections ($x.L$) implies the direct encoding of System $\text{F}_{<:}$ subtyping, a problem well-known to be undecidable. Finally, the presence of recursive types coupled with path dependent types also provides an encoding of System $\text{F}_{<:}$, and thus implies subtyping is undecidable. These last two issues will be addressed in Section 3.2, in this Section I discuss the issue of transitivity and address some concerns of expressiveness.

Syntax Directedness: Transitivity

A general transitivity rule is a problem when attempting to define an algorithm from a set of subtyping rules. The need for a third middle type requires searching the set of all possible types. This is often either infeasible due to the large number of potential types, or impossible due to the existence of infinite types. The alternative is to design subtyping rule set in such a way that transitivity arises naturally from the rules. An example of such a rule set is the Kernel $\text{F}_{<:}$ variant of System $\text{F}_{<:}$ [70], where transitivity is defined for type variables only, allowing transitivity in its general form to be proven. I have already discussed Kernel $\text{F}_{<:}$ to some degree in Section 2.1.5, but it is useful to revisit it here as an analogy for changes that can be made to the subtype rules of Basic Wyvern. In Figure 3.4 I provide the relevant change to the rules from System $F_{<:}$ to Kernel $F_{<:}$. The general transitivity rule

$$\frac{\Gamma \vdash \tau_1 \ <: \ \tau_2 \qquad \Gamma \vdash \tau_2 \ <: \ \tau_3}{\Gamma \vdash \tau_1 \ <: \ \tau_3} \ (\textsc{S-Trans})$$

$$\frac{\Gamma(\alpha) = \tau}{\Gamma \vdash \alpha \ <: \ \tau} \ (\textsc{S-Var}) \qquad \Rightarrow \qquad \frac{\Gamma(\alpha) = \tau' \qquad \Gamma \vdash \tau' \ <: \ \tau}{\Gamma \vdash \alpha \ <: \ \tau} \ (\textsc{S-Var})$$

Figure 3.4: Syntax Directed modification to System $F_{<:}$

(S-Trans) and the rule for subtyping type variables (S-Var on the left of Figure 3.4) is replaced with a modified rule for subtyping variables (S-Var on the right of Figure 3.4) that includes transitivity in the premises. These two rule sets are equivalent in that they can prove the same set of judgements (Kernel $F_{<:}$ differs from System $F_{<:}$ in other ways that are not important here).

The subtype rules of Figure 3.3 resemble the rules on the left of the example in Figure 3.4. Transitivity is explicit in Figure 3.3, and type members are judged as subtyping their upper bounds (S-Upper) and supertyping their lower bounds (S-Lower). Earlier formulations of DOT subtyping [13] did not include an explicit rule for transitivity, and resembled the rule on the right of Figure 3.4. I use those formulations to inform a syntax directed form of subtyping for type selections in Figure 3.5.

In Wyvern (and more specifically DOT), such syntax directed subtyping does not provide general transitivity in the same way that it does for Kernel $F_{<:}$. As discussed in Section 2.2.1, transitivity sits in the centre of the long struggle for type safety in DOT. Part of this story is the complex interactions path dependent types have with environment narrowing, and how this can lead to ill-formed bounds on type members. The complexities and details of this topic, while interesting, not entirely relevant to this thesis, and are

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 \ <: \ \tau_2 \\ \Gamma \vdash \tau_2 \ <: \ \tau_3\end{array}}{\Gamma \vdash \tau_1 \ <: \ \tau_3} \ (\textsc{S-Trans})$$

$$\frac{\Gamma \vdash x : \{L \ : \_ \ldots \tau\}}{\Gamma \vdash x.L \ <: \ \tau} \ (\textsc{S-Upper}) \quad \Rightarrow \quad \frac{\begin{array}{c}\Gamma \vdash x : \{L \ : \_ \ldots \tau_1\} \\ \Gamma \vdash \tau_1 \ <: \ \tau_2\end{array}}{\Gamma \vdash x.L \ <: \ \tau_2} \ (\textsc{S-Upper})$$

$$\frac{\Gamma \vdash x : \{L \ : \tau \ldots \_\}}{\Gamma \vdash \tau \ <: \ x.L} \ (\textsc{S-Lower}) \qquad \frac{\begin{array}{c}\Gamma \vdash x : \{L \ : \tau_2 \ldots \_\} \\ \Gamma \vdash \tau_1 \ <: \ \tau_2\end{array}}{\Gamma \vdash \tau_1 \ <: \ x.L} \ (\textsc{S-Lower})$$

Figure 3.5: Syntax Directed modification to Basic Wyvern Subtyping

better left to Rompf and Amin. Their insight however, is very important, as they demonstrate that the loss of transitivity is only relevant to types in ill-formed environments, and that typing guarantees that such environments do not compromise the type safety of the language. This is important, because it suggests that while the modification does not preserve transitivity, it is type safe, and the instances where transitivity is lost relate only to ill-formed type bounds.

Syntax Directedness: Subsumption

Related to transitivity, the subsumption rule T-Sub is a subtyping rule that commonly features in formal descriptions of programming languages. DOT for instance is an example of this. In the case of DOT, subsumption plays an important role in typing, as it allows intersection types to be unpacked during typing, thus allowing terms to be typed by either component of an intersection. This is also true of Basic Wyvern, and as has been discussed in Section 3.1.1, subsumption empowers a variety of useful typings within Basic Wyvern (e.g. if x has type $y.L$, which in turn has upper bound τ, then x is also judged as having type τ).

Subsumption, is however problematic for constructing a typing procedure in the same manner that explicit transitivity is problematic when designing a subtyping procedure. In rule T-Sub, there is no obvious candidate type for τ'. In Figure 3.6 I replace the T-Sub rule with two other rules that introduce explicit subsumption: T-Str and T-Sel. These two rules provide for very specific forms of subsumption. Under T-Str, a variable of some structure type is also judged as having a type with any one of it's associated declaration types. Under T-Sel, a variable x of type $y.L$ is also judged has having type τ, where τ is the upper bound of $y.L$. Unfortunately, as with the syntax directed modification to subtyping defined in Figure 3.5, the new rule set is not as expressive as the old.

$$\frac{\begin{array}{c}\Gamma \vdash x \;:\; \tau \\ \Gamma \vdash \tau \;<:\; \tau'\end{array}}{\Gamma \vdash x \;:\; \tau'}\ \text{(S-Sub)} \quad \Rightarrow \quad \begin{array}{c} \dfrac{\Gamma \vdash x \;:\; \{\overline{\sigma}\} \qquad \sigma \in \overline{\sigma}}{\Gamma \vdash x \;:\; \{\sigma\}}\ \text{(T-Str)} \\[2em] \dfrac{\begin{array}{c}\Gamma \vdash x \;:\; y.L \\ \Gamma \vdash y \;:\; \{L \;:\_ \ldots \tau\}\end{array}}{\Gamma \vdash x \;:\; \tau}\ \text{(T-Sel)} \end{array}$$

Figure 3.6: Syntax Directed modification to Basic Wyvern Typing

Expressiveness: Type Refinements and Intersection Types

The Basic Wyvern type system includes both recursive types and object types, critically allowing recursive object types ($\mu(z : \{\overline{\sigma}\})$). What this basic form of Wyvern does not allow is the refinement of existing types, that is, the reuse of existing type definitions with added type information. This is a critical feature allowing for polymorphic data types and nominality. I provide a deeper discussion of type refinements in Section 2.2.1.

In order to facilitate type refinements, I replace recursive and structure types with a combined syntactic form. Figure 3.7 replaces the recursive and structure types of Basic Wyvern with two forms for type refinement: a refinement on ⊤ ($\{z \Rightarrow \overline{\sigma}\}$), and a refinement on a selection type ($x.L\{z \Rightarrow \overline{\sigma}\}$). This is not a new type form, earlier variants of DOT, namely Amin et al. in 2012 and Amin et al. in 2014 also included similar type refinements.

$$\begin{array}{llll} \tau & ::= & & \textbf{Type} \\ & \vdots & & \\ & \mu(z:\tau) & \textit{recursive type} & \\ & \{\overline{\sigma}\} & \textit{structure type} & \end{array} \quad \Rightarrow \quad \begin{array}{llll} \tau & ::= & & \textbf{Type} \\ & \vdots & & \\ & \{z \Rightarrow \overline{\sigma}\} & & \textit{top refinement} \\ & x.L\{z \Rightarrow \overline{\sigma}\} & & \textit{selection refinement} \end{array}$$

Figure 3.7: Replacing Recursive and Structure types with Recursive Type Refinement

While type refinements are less expressive than full intersection types, they do provide a very specific aspect of intersection types: the refining of an existing object type with new member information. Type refinements do not allow for two useful instances of expressiveness: (i) the kind of type hierarchies one finds in multiple inheritance, and (ii) ad-hoc intersection types. Thus, it is not possible to write the following example:

```
type List[Elem = ⊤] <: Equatable[List[Elem]] & Traversable[Elem]{
    def equals : List[Elem] -> bool
    def traverse[U] : (Elem -> U) -> ⊤
  }
}
```

It is also not possible to specify arguments as inhabiting more than one type, as in the example below:

```
def showTraversable(t : Printable & Traversable) = {
        t.traverse(_.print)
}
```

Thus, type refinements do not allow for the full encoding of a language such as Java, as it includes multiple inheritance. This is certainly a limitation, but it is a limitation that keeps the language in question tractable.

Single Bounded Type Declarations

As a final modification, I replace the syntactic type form for type declarations. This is given in Figure 3.8. I introduce a specific syntactic form for upper

$$
\begin{array}{lll}
\sigma & ::= & \textbf{Declaration Type} \\
 & \vdots & \\
 & L \; : \tau \ldots \tau &
\end{array}
\quad \Rightarrow \quad
\begin{array}{llr}
\sigma & ::= & \textbf{Declaration Type} \\
 & \vdots & \\
 & L \; \leqslant \tau & \textit{upper bound} \\
 & L \; \geqslant \tau & \textit{lower bound} \\
 & L \; = \; \tau & \textit{specific type}
\end{array}
$$

Figure 3.8: Double Bounded Type Declarations with Single Bounded Type Declarations

bounded type declarations ($L \; \leqslant \tau$), lower bounded type declarations ($L \; \geqslant \tau$) and specific type declarations ($L \; = \; \tau$). Note: the upper bounded type declarations do in fact have a lower bound of $\bot$. Similarly, lower bounded type members have an upper bound of $\top$. The specific type declaration $L \; = \; \tau$ is in fact the type declaration with both upper and lower bound τ. These syntactic forms allow discussions about type declarations to be more exact, as each syntactic form has different uses and properties. Thus discussions about each specific form become simpler. It is also notable that this comes at almost no loss in expressiveness. While it is true that there is no single syntactic form that can express $L \; : \tau_1 \ldots \tau_2$, the type $\{L \; : \tau_1 \ldots \tau_2\}$

is easily expressed as

$$\left\{ \begin{array}{ll} L & \geqslant \tau_1 \\ L & \leqslant \tau_2 \end{array} \right\}$$

3.1.3 Wyv_{core}

Now that some guidelines have been established in constructing a starting point for Decidable Wyvern, I present the core Wyvern syntax and semantics. Figure 3.9 gives the syntax of types in Wyv_{core}. **Types** are either a type member selection on a variable ($x.L$), the top type ($\top$) or the bottom type ($\bot$), type refinements ($\tau\{z \Rightarrow \overline{\sigma}\}$) or dependent function types ($\forall(x : \tau_1).\tau_2$). A type refinement $\tau\{z \Rightarrow \overline{\sigma}\}$ is a base type τ refined with a set of declaration types $\overline{\sigma}$ and a self reference z. A type refinement is restricted to either a refinement on $\top$ ($\{z \Rightarrow \overline{\sigma}\}$) or a selection type ($x.L\{z \Rightarrow \overline{\sigma}\}$). Dependent function types ($\forall(x : \tau_1).\tau_2$) type functions where the return type may be dependent on the argument type. **Declaration Types** are the types of declarations. Unlike Basic Wyvern in Figure 3.1, declaration types are explicitly either upper bounded ($L \leqslant \tau$), lower bounded ($L \geqslant \tau$), or an exact type ($L = \tau$). A value declaration ($l : \tau$) is unchanged from that of Figure 3.1.

Throughout the type system I use several shorthands for different syntactic variations. Type refinements combine the recursive and structure types of Figure 3.1 with a restricted form of intersection types. Type refinements can thus be used to capture one or more of these concepts at a time and it is sometimes useful to distinguish between those uses. As an example, type refinements are not required to be refinements on a selection type, the refinement $\top\{z \Rightarrow \overline{\sigma}\}$ corresponds to the recursive structure type $\{z \Rightarrow \overline{\sigma}\}$. Depending on which aspects of type refinement are in use, we use two shorthands: type refinements with $\top$ as the base type are written as $\{z \Rightarrow \overline{\sigma}\}$ and type refinements where the self variable is not free in the refining declarations is written as $\tau\{\overline{\sigma}\}$. Both of these combined give us the structure types of Figure 3.1: $\{\overline{\sigma}\}$. There are several places in the definition of the semantics where rules may apply to multiple syntactic forms of declaration types. In

$$
\begin{array}{lll}
\tau & ::= & \textbf{Type} \\
 & \forall(x:\tau).\tau & \textit{dependent function} \\
 & \{z \Rightarrow \overline{\sigma}\} & \textit{top refinement} \\
 & x.L\{z \Rightarrow \overline{\sigma}\} & \textit{selection refinement} \\
 & x.L & \textit{type selection} \\
 & \top & \textit{top} \\
 & \bot & \textit{bottom} \\
\sigma & ::= & \textbf{Declaration Type} \\
 & L \leqslant \tau & \textit{upper bound} \\
 & L \geqslant \tau & \textit{lower bound} \\
 & L = \tau & \textit{equality} \\
 & l : \tau & \textit{value} \\
\Gamma & ::= & \textbf{Environment} \\
 & \emptyset & \\
 & \Gamma, x:\tau &
\end{array}
$$

Figure 3.9: *Wyv*$_{core}$ Syntax

$$\boxed{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ (\text{T-Var})$$

$$\frac{\Gamma \vdash x : \tau\{z \Rightarrow \overline{\sigma}\} \qquad \sigma \in \overline{\sigma}}{\Gamma \vdash x : \{[x/z]\sigma\}}\ (\text{T-Rec})$$

$$\frac{\Gamma \vdash x : \tau\{z \Rightarrow \overline{\sigma}\}}{\Gamma \vdash x : \tau}\ (\text{T-Rfn})$$

$$\frac{\Gamma \vdash x : y.L \qquad \Gamma \vdash y : \{L \leqslant \tau\}}{\Gamma \vdash x : \tau}\ (\text{T-Sel})$$

$$\frac{\Gamma \vdash x : \{L = \tau\}}{\Gamma \vdash x : \{L \leqslant \tau\}}\ (\text{T-Eq}_1)$$

$$\frac{\Gamma \vdash x : \{L = \tau\}}{\Gamma \vdash x : \{L \geqslant \tau\}}\ (\text{T-Eq}_2)$$

Figure 3.10: *Wyv*$_{core}$ Typing

$$\frac{\Gamma \vdash x : \{L \leqslant \tau\} \qquad \tau \textit{ is extendable}}{\Gamma \vdash x.L \leqslant:: \tau}\ (\text{E-Sel}) \qquad \frac{\Gamma \vdash \tau \leqslant:: \tau'}{\Gamma \vdash \tau\{z \Rightarrow \overline{\sigma}\} \leqslant:: \textit{flat}(\tau', \overline{\sigma}, z)}\ (\text{E-Rfn})$$

Figure 3.11: *Wyv*$_{core}$ Type Extension

$$x.L \textit{ is extendable} \qquad \top \textit{ is extendable}$$

$$\tau\{z \Rightarrow \overline{\sigma}\} \textit{ is extendable}$$

Figure 3.12: *Wyv*$_{core}$ Type Extendability

$$
\begin{array}{lcl}
\textit{flat}(\top, \overline{\sigma}, z) & = & \top\{z \Rightarrow \overline{\sigma}\} \\
\textit{flat}(x.L, \overline{\sigma}, z) & = & x.L\{z \Rightarrow \overline{\sigma}\} \\
\textit{flat}(\tau\{z \Rightarrow \overline{\sigma}_1\}, \overline{\sigma}_2, z) & = & \tau\{z \Rightarrow \overline{\sigma}_1, \overline{\sigma}_2\}
\end{array}
$$

Figure 3.13: Refinement Flattening

$$\boxed{\Gamma \vdash \tau_1 <: \tau_2,\ \ \overline{\sigma}_1 <: \overline{\sigma}_2,\ \ \sigma_1 <: \sigma_2}$$

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

$$\frac{\Gamma \vdash x \ : \ \{L \ \leqslant \tau'\} \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash x.L \ <: \ \tau} \quad (\text{S-}\textsc{Upper})$$

$$\frac{\Gamma \vdash x \ : \ \{L \ \geqslant \tau'\} \qquad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash \tau \ <: \ x.L} \quad (\text{S-}\textsc{Lower})$$

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

$$\frac{\Gamma, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 <: \overline{\sigma}_2}{\Gamma \vdash \tau\{z \Rightarrow \overline{\sigma}_1\} <: \tau\{z \Rightarrow \overline{\sigma}_2\}} \quad (\text{S-}\textsc{Rfn}) \qquad \frac{\Gamma \vdash \tau_1 \ \leqslant:: \ \tau \qquad \Gamma \vdash \tau \ <: \ \tau_2}{\Gamma \vdash \tau_1 \ <: \ \tau_2} \quad (\text{S-}\textsc{Ext})$$

$$\frac{\forall\ \sigma_2 \ \in \ \overline{\sigma}_2,\ \exists\ \sigma_1 \ \in \ \overline{\sigma}_1 \ s.t.\ \Gamma \vdash \sigma_1 \ <: \sigma_2}{\Gamma \vdash \overline{\sigma}_1 <: \ \overline{\sigma}_2} \quad (\text{S-}\textsc{Decls})$$

$$\frac{\Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash L \leqslant\!/\!= \tau_1 <: \ L \ \leqslant \tau_2} \quad (\text{S}_\sigma\text{-}\textsc{Upper}) \qquad \frac{\Gamma \vdash \tau_2 \ <: \ \tau_1}{\Gamma \vdash L \geqslant\!/\!= \tau_1 <: \ L \ \geqslant \tau_2} \quad (\text{S}_\sigma\text{-}\textsc{Lower})$$

$$\frac{\Gamma \vdash \tau_2 \ <: \ \tau_1 \qquad \Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash L \ = \ \tau_1 <: \ L \ = \ \tau_2} \quad (\text{S}_\sigma\text{-}\textsc{Equal}) \qquad \frac{\Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash l : \tau_1 <: \ l : \tau_2} \quad (\text{S}_\sigma\text{-}\textsc{Value})$$

Figure 3.14: Wyv_{core} Subtyping

these cases, the syntax $L \leqslant\!/\!= \tau$ or $L \geqslant\!/\!= \tau$ is used to indicate that we are interested in either the upper or lower bound of a type definition, and do not care whether it is an exact type or not.

I define variable **typing** in Figure 3.10. Typing in some ways looks similar to that of Basic Wyvern, and in other ways is modified in line with Section 3.1.2. T-VAR is unchanged from Basic Wyvern, capturing typing of a variable within an environment. T-REC in some ways resembles the T-REC of Basic Wyvern in the way the recursive type is "unpacked". In Wyv_{core} however, recursive types have been replaced with the combined form of recursive type refinements, thus the T-REC in Wyv_{core} also resembles the T-STR of Figure 3.6. Thus, T-REC captures the typing: if a variable x has type $\tau\{z \Rightarrow \overline{\sigma}\}$, then it is also judged as having the the type of specific σ in $\overline{\sigma}$, with the self reference replaced with x. As with the subtyping of Figure 3.6, T-REC is a case of explicit subsumption. Another instance of explicit subsumption can be seen in T-RFN. Here a variable x of type $\tau\{z \Rightarrow \overline{\sigma}\}$ is also judged as having type τ. The T-SEL rule again, is an instance of explicit subsumption, and is identical to the T-SEL rule of Figure 3.6. Finally, I introduce two new forms of explicit subsumption: a variable with type $\{L \ = \ \tau\}$ is also judged as having (i) type $\{L \ \leqslant \tau\}$ (T-EQ$_1$), and (ii) type $\{L \ \geqslant \tau\}$ (T-EQ$_2$).

Type extension ($\Gamma \ \vdash \ \tau_1 \ \ \leqslant:: \ \ \tau_2$) is defined in Figure 3.11. Type extension is an important relation, and is conceptually similar to the relationship implied by the `extends` keyword in Java. In Java, the class declaration `class C<E> extends D<E>` describes not only a subclass relationship, but the subtype relationship between different instances of `C` and `D`. For example, one might say that `C<Int>` extends `D<Int>`. In Wyv_{core}, a type declaration $C \ \ \leqslant \ x.D\{z \Rightarrow \ E \ \ \leqslant \top, \ T \ \ = \ \ z.E\}$ implies a similar relationship. As in Java, we want to be able say that $x.C\{E \ = \ \texttt{Int}\}$ extends $x.D\{z \Rightarrow E \ = \ \texttt{Int}, \ T \ = \ z.E\}$. Type extension captures this relationship:

$$\Gamma \vdash x.C\{E \ = \ \texttt{Int}\} \ \leqslant:: \ x.D \left\{ z \Rightarrow \begin{array}{l} E \ = \ \texttt{Int} \\ T \ = \ z.E \end{array} \right\}$$

This is an important judgement in capturing transitive subtyping for type refinements. To extend the previous example, if $x.D\{z \Rightarrow E = \texttt{Int}, T = z.E\}$ subtypes some other type τ, then it should follow that $x.C\{E = \texttt{Int}\}$ subtypes τ.

Type extension is captured by two rules: E-Sel and E-Rfn. All selection types extend their upper bound (E-Sel). Type extension for type refinements is more complex. A type refinement $\tau\{z \Rightarrow \overline{\sigma}\}$ extends the type extended by τ (τ'), with the refinement ($\_\{z \Rightarrow \overline{\sigma}\}$) flattened into it. Flattening is defined by the $flat$ function in Figure 3.13, and is simply a merging of multiple refinements. $flat$ takes three parameters: the base type being flattened into τ, the self variable z, and the refinement $\overline{\sigma}$ to be flattened into τ. If τ (the type being flattened into) is either top or a selection type, the result of $flat(\tau, z, \overline{\sigma})$ is the type refinement $\tau\{z \Rightarrow \overline{\sigma}\}$. If τ is some type refinement $\tau'\{z \Rightarrow \overline{\sigma}'\}$, then $flat(\tau, z, \overline{\sigma})$ merges the two refinements and returns $\tau\{z \Rightarrow \overline{\sigma}', \overline{\sigma}\}$.

Subtyping is defined in Figure 3.14. Subtyping is explicitly reflexive only with regard to selection types on variables (S-Rfl). Subtyping is bounded below by $\bot$ and above by $\top$ (S-Bot and S-Top). Subtyping of type selections is defined in relation to their upper and lower bounds (S-Upper and S-Lower respectively). Subtyping of dependent function types is defined in S-All, the argument types are subtyped contra-variantly while the return types are subtyped covariantly. Type refinement subtyping is relatively complex and is captured in the S-Rfn, S-Ext. S-Rfn compares two refinements on equal types. S-Ext uses type extension to determine the upper bound of a type that may include a refinement. 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).

A Note on Deriving a Subtyping Algorithm for *Wyv*$_{core}$

In the design of *Wyv*$_{core}$ earlier in this section, I discussed the importance of being able to construct an algorithm for subtyping in *Wyv*$_{core}$. For this reason Wyv_{core} does not include a rule for subtype transitivity. The reader may however, have noticed that subtyping in Wyv_{core} is not in fact syntax directed, and thus constructing a subtyping algorithm is perhaps not trivial. There are three reasons for this:

1. A proof search for $x.A \ <: \ y.B$ has no information that would favour either S-Upper or S-Lower over the other. Indeed, both may apply. Such a subtyping question represents an instance of branching, and any subtype algorithm must search both branches.

2. The S-Ext rule is not syntax directed as there is no syntactic information supplied by the rule that would inform a subtype algorithm when such a rule should be used. The definition of type extension ($\leqslant::$) however, implies that there are only two instances when it is applicable: (i) when $\tau_1 = x.L$ where τ is its upper bound, and (ii) when τ_1 and τ_2 are both type refinements. The first case is subsumed by S-Upper, and so can be ignored during construction of a subtyping algorithm (although it is important for completeness for the type extension relation). The second case can be further refined (after introducing static cycle detection in Chapter 4) by noting that cycle detection can exclude the case where the two base types are equal (types that extend themselves can be easily excluded by cycle detection, and is similar to the way that a Java disallows cyclical inheritance). This leaves the subtype algorithm designer with only one applicable instance, when τ_1 and τ_2 are both type refinements with differing base types.

3. The definition of type refinements does not restrict declaration type names to be unique, thus the variable typing (specifically those derived during T-Rec and T-Rfn and subsequently the upper bounds of selection types) in S-Upper and S-Lower is not guaranteed to be unique. While typing may not be unique, there are guaranteed to be finitely many typings for any variable x (as variable typing does not introduce new type information to the environment, and so is not di-

vergent). Further it seems unlikely that a variable would have very many typings as intuitively multiple definitions for a type seem most likely to arise from type extension, and type extension hierarchies seem unlikely to be very deep. This has not been demonstrated empirically, and is based solely on the author's intuition.

3.2 Subtype Undecidability in Wyv_{core}

Subtyping in Wyv_{core} is undecidable. Undecidability can be demonstrated in three different ways, each indirectly demonstrating a reduction of subtyping in Wyv_{core} to the halting problem [80], and each implicating a different aspect of subtyping.

1. *Encoding System $F_{<:}$ in* Wyv$_{core}$ *using Dependent Function Types* (Section 3.2.1): Subtyping in System F$_{<:}$ has long been known to be undecidable. I define an encoding from System F$_{<:}$ into Wyv_{core} using dependent function types, to demonstrate that subtyping in Wyv_{core} subsumes subtyping in System F$_{<:}$.

2. *Encoding System $F_{<:}$ in* Wyv$_{core}$ *using Recursive and Path Dependent Types* (Section 3.2.2): I demonstrate a distinct encoding of System F$_{<:}$ in Wyv_{core} using only recursive and path dependent types. This demonstrates that undecidability of subtyping in Wyv_{core} is not merely a matter of constraining dependent function types.

3. *Encoding Java Generics in* Wyv$_{core}$ (Section 3.2.3): As I discuss in Section 2.2.3, Java Generics have recently been demonstrated to be Turing complete, and thus subtyping in Java Generics is undecidable. I demonstrate that the fragment of Java Generics proven to be Turing complete is encodable in Wyv_{core}, in turn implying that Wyv_{core} is Turing complete.

All three of these encodings make use of different aspects of Wyv_{core} subtyping.

1. The encoding of Section 3.2.1 uses dependent function types to encode System F$_{<:}$.

2. The encoding of Section 3.2.2 uses recursive and path dependent types to encode System F$_{<:}$.

3. The encoding of Section 3.2.3 uses nominality and recursive type refinement to encode Java Generics.

3.2.1 Encoding System F$_{<:}$: Dependent Function Types

As Section 2.1.5 discussed, subtyping in System F$_{<:}$ is undecidable. In this section I demonstrate that *Wyv*$_{core}$ encodes the decidability problems of System F$_{<:}$. The encoding of System F$_{<:}$ is fairly straightforward given the combination of dependent function types and path dependent types. The encoding is easily achieved in Figure 3.15 using a combination of path dependent types and dependent function types. Note: I use the Latin T for types and in Figure 3.15 instead of the Greek τ of Section 2.1.4, to avoid confusion with the types of *Wyv*$_{core}$. I also use Greek α (instead of the Latin x used in *Wyv*$_{core}$) to represent both type variables of System F$_{<:}$ and the variables of *Wyv*$_{core}$.

The fundamental insight into the encoding of Figure 3.15, is that type variables can be encoded as variables with type members. Thus we encode a type variable α as a selection type $\alpha.A$. We assume that it is possible to use the same variable identifiers in *Wyv*$_{core}$ as in System F$_{<:}$ for ease of following examples. Function types in System F$_{<:}$ are encoded using dependent function types, ignoring the argument in the return type. The encoding of universally quantified types in *Wyv*$_{core}$ is achieved again using dependent function types. This time we do not ignore the argument, rather allowing the return type to use the argument variable. Thus, in the same way that subtyping of universally quantified types in System F$_{<:}$ introduce a type variable and a bound to the environment, dependent function types introduce a variable typed with a structure type containing an upper bound type member.

$$\begin{array}{lcl}
\mathcal{F}(\alpha) & = & \alpha.A \\
\mathcal{F}(T_1 \to T_2) & = & \forall(\_ : \mathcal{F}(T_1)).\mathcal{F}(T_2) \\
\mathcal{F}(\forall(\alpha \leqslant T_1).T_2) & = & \forall(\alpha : \{A \ \leqslant \mathcal{F}(T_1)\}).\mathcal{F}(T_2) \\
\mathcal{F}(\emptyset) & = & \emptyset \\
\mathcal{F}(\alpha \leqslant T, \Delta) & = & \alpha \ : \ \{A \ \leqslant \mathcal{F}(T)\},\ \mathcal{F}(\Delta)
\end{array}$$

Figure 3.15: Encoding System $F_{<:}$ in Wyv_{core}

Finally, the type environment Δ is encoded as a series of variable typings, and not as a series of type variable upper bounds.

The encoding of Figure 3.15 is somewhat unsurprising considering the similar of the semantics of System $F_{<:}$ and the semantics for dependent functions in Wyv_{core}. As Pierce actually proves the undecidability of System $F_{<:}$'s subtyping using a normal form of System $F_{<:}$: $F^N_{<:}$. I give the subtype rules for $F^N_{<:}$ in Figure 3.16 (again using Latin characters for types): In fact, it is the NALL rule (S-ALL in System $F_{<:}$) that is the main cause of undecidability. For the encoding to be sound, we require $\mathcal{F}$ to preserve the three properties of the subtyping described by NALL: (i) contra-variance on the type bounds, (ii) covariance on the returns type, and (iii) environment narrowing in the subtyping of the return types, introducing the bound on the right hand side to the environment. To give an intuitive sense of why the encoding of Figure 3.15 makes sense, consider the transformation of the derivation in Figure 3.17

It is easy enough to demonstrate that the specific property described by Figure 3.17 holds by demonstrating that the more general Theorem 3.2.1 holds.

Theorem 3.2.1 ($\mathcal{F}$ preserves subtyping). $\Delta \vdash T_1 \ <: \ T_2 \Rightarrow \mathcal{F}(\Delta) \vdash \mathcal{F}(T_1) \ <: \ \mathcal{F}(T_2)$

Proof. Let n be the depth of some $F^N_{<:}$ subtyping derivation $\Delta \vdash T_1 \ <: \ T_2$. We proceed by induction on n.

$$\Delta \vdash T \ <: \ \top \quad (\textsc{NTop}) \qquad\qquad \Delta \vdash \alpha \ <: \ \alpha \quad (\textsc{NRefl})$$

$$\frac{\Delta(\alpha) = T_1 \qquad \Delta \vdash T_1 \ <: \ T_2}{\Delta \vdash \alpha \ <: \ T_1} \ (\textsc{NVar})$$

$$\frac{\Delta \vdash T_2 \ <: \ T_1 \qquad \Delta \vdash T_1' \ <: \ T_2'}{\Delta \vdash T_1 \to T_1' \ <: \ T_2 \to T_2'} \ (\textsc{NArr})$$

$$\frac{\Delta \vdash T_2 \ <: \ T_1 \qquad \Delta, \alpha \leqslant T_2 \vdash T_1' \ <: \ T_2'}{\Delta \vdash \forall(\alpha \leqslant T_1).T_1' \ <: \ \forall(\alpha \leqslant T_2).T_2'} \ (\textsc{NAll})$$

Figure 3.16: $\mathrm{F}_{<:}^{N}$ Subtyping

Induction Hypothesis: for any subtype derivation $\Delta' \ \vdash T_1' \ <: \ T_2'$ of depth $m \ < \ n$, then $\mathcal{F}(\Delta') \vdash \mathcal{F}(T_1') \ <: \ \mathcal{F}(T_2')$.
Using the induction hypothesis, we show that for each subtype rule in Figure 3.16, $\mathcal{F}$ preserves subtyping.

Case 1 (Base Case: NTop)**.** Trivial.

Case 2 (Base Case: NRefl)**.** Trivial.

Case 3 (Base Case: NVar)**.**

$$\frac{\Delta(\alpha) = T_1 \qquad \Delta \vdash T_1 \ <: \ T_2}{\Delta \vdash \alpha \ <: \ T_1} \ (\textsc{NVar})$$

Since $\Delta(\alpha) = T_1$, it follows by the definition of $\mathcal{F}$ that

$$\mathcal{F}(\Delta)(\alpha) = \{A \ \leqslant \mathcal{F}(T_1)\}$$

Thus, by T-Var, we have

$$\mathcal{F}(\Delta) \vdash \alpha \ : \ \{A \ \leqslant \mathcal{F}(T_1)\}$$

$$\frac{\Delta \vdash T_2 \ <: \ T_1 \qquad \Delta, \alpha \leqslant T_2 \vdash T_1' \ <: \ T_2'}{\Delta \vdash \forall(\alpha \leqslant T_1).T_1' \ <: \ \forall(\alpha \leqslant T_2).T_2'}$$

$$\Downarrow$$

$$\frac{\dfrac{\mathcal{F}(\Delta) \vdash \mathcal{F}(T_2) <: \mathcal{F}(T_1)}{\mathcal{F}(\Delta) \vdash \{A \ \leqslant \mathcal{F}(T_2)\} <: \{A \ \leqslant \mathcal{F}(T_1)\}} \qquad \mathcal{F}(\Delta), \alpha : \{A \ \leqslant \mathcal{F}(T_2)\} \vdash \mathcal{F}(T_1') <: \mathcal{F}(T_2')}{\mathcal{F}(\Delta) \vdash \forall(\alpha : \{A \ \leqslant \mathcal{F}(T_1)\}).\mathcal{F}(T_1') <: \forall(\alpha : \{A \ \leqslant \mathcal{F}(T_2)\}).\mathcal{F}(T_2')}$$

Figure 3.17: $\mathrm{F}^N_{<:}$ NALL ⇒ *Wyv*$_{core}$ S-ALL

By the induction hypothesis, we have

$$\mathcal{F}(\Delta) \vdash \mathcal{F}(T_1) \ <: \ \mathcal{F}(T_2)$$

Thus by S-UPPER we get the desired result:

$$\mathcal{F}(\Delta) \vdash \alpha \ <: \ \mathcal{F}(T_2)$$

Case 4 (Inductive Case: NARR)**.**

$$\frac{\Delta \vdash T_2 \ <: \ T_1 \qquad \Delta \vdash T_1' \ <: \ T_2'}{\Delta \vdash T_1 \to T_1' \ <: \ T_2 \to T_2'} \quad (\text{NARR})$$

By the induction hypothesis, we have

$$\mathcal{F}(\Delta) \vdash \mathcal{F}(T_2) \ <: \ \mathcal{F}(T_1) \qquad \mathcal{F}(\Delta) \vdash \mathcal{F}(T_1') \ <: \ \mathcal{F}(T_2')$$

By environment weakening in *Wyv*$_{core}$ we get

$$\mathcal{F}(\Delta), \_ : \mathcal{F}(T_2) \vdash \mathcal{F}(T_1') \ <: \ \mathcal{F}(T_2')$$

Thus, by S-ALL we get the desired result:

$$\mathcal{F}(\Delta) \vdash \forall(\_ : \mathcal{F}(T_1)).\mathcal{F}(T_1') \ <: \ \forall(\_ : \mathcal{F}(T_2)).\mathcal{F}(T_2')$$

Case 5 (Inductive Case: NALL)**.**

$$\frac{\Delta \vdash T_2 \ <: \ T_1 \qquad \Delta, \alpha \leqslant T_2 \vdash T_1' \ <: \ T_2'}{\Delta \vdash \forall(\alpha \leqslant T_1).T_1' \ <: \ \forall(\alpha \leqslant T_2).T_2'} \ (\text{NALL})$$

By the induction hypothesis, we have

$$\mathcal{F}(\Delta) \vdash \mathcal{F}(T_2) \ <: \ \mathcal{F}(T_1) \qquad \mathcal{F}(\Delta), \alpha : \{A \ \leqslant \mathcal{F}(T_2)\} \vdash \mathcal{F}(T_1') \ <: \ \mathcal{F}(T_2')$$

Thus, by S-ALL we get the desired result:

$$\mathcal{F}(\Delta) \vdash \forall(\alpha : \{A \ \leqslant \mathcal{F}(T_1)\}).\mathcal{F}(T_1') \ <: \ \forall(\alpha : \{A \ \leqslant \mathcal{F}(T_2)\}).\mathcal{F}(T_2')$$

□

Wyvern encodes the entire of System $F_{<:}$ and one could easily create a similar example in Wyvern to that in Figure 2.14 using the encoding in Figure 3.15. An encoding of $F^N_{<:}$ to Wyv_{core} is not in itself sufficient to demonstrate subtype undecidability, for that a demonstration that Wyv_{core} is a conservative extension to $F^N_{<:}$ is needed, that is that not only does every subtype derivation in $F^N_{<:}$ have an equivalent in Wyv_{core}, but every subtype relationship that is not derivable in $F^N_{<:}$ is also not derivable in Wyv_{core}.

Theorem 3.2.2 (Conservative Extension)**.** *If $\Delta \vdash T_1 \ <: T_2$ is not derivable in $F^N_{<:}$, then $\mathcal{F}(\Delta) \vdash \mathcal{F}(T_1) \ <: \mathcal{F}(T_2)$ is not derivable in* Wyv_{core}.

Proof. We prove the contrapositive, (i.e. $\mathcal{F}(\Delta) \vdash \mathcal{F}(\tau_1) \ <: \mathcal{F}(\tau_2)$ implies $\Delta \vdash \tau_1 \ <: \tau_2$) by induction on the structure of the proof of $\mathcal{F}(\Gamma) \vdash \mathcal{F}(\tau_1) \ <: \mathcal{F}(\tau_2)$. The proof is relatively straightforward as the range of $\mathcal{F}$ does not include structure types, $\perp$ or lower bounds, thus one need only consider cases for the rules S-TOP, S-ALL, S-RFL and S-UPPER. The S-TOP and S-RFL cases are trivial as they immediately hold by their $F^N_{<:}$ counterparts. The S-ALL case is derived by an application of the induction hypothesis and the NALL rule in $F^N_{<:}$. Finally, the S-UPPER case holds by the induction hypothesis and the NVAR rule in $F^N_{<:}$. □

$$
\begin{array}{lcl}
F'(\top) & = & \top \\
F'(\alpha) & = & \alpha.A \\
F'(T_1 \ \rightarrow \ T_2) & = & \left\{ \begin{array}{l} A \ \geqslant F'(T_1) \\ B \ \leqslant F'(T_2) \end{array} \right\} \\
F'(\forall(\alpha \leqslant T_1).T_2) & = & \neg\left\{ \alpha \Rightarrow \begin{array}{l} A \ \leqslant F'(T_1) \\ B \ \geqslant F'(T_2) \end{array} \right\} \\
\text{where } \neg\tau & = & \{X \ \geqslant \tau\} \\
\mathcal{F}(\emptyset) & = & \emptyset \\
\mathcal{F}(\alpha \leqslant T, \Delta) & = & \alpha \ : \ \left\{ \begin{array}{l} A \ \geqslant F'(T) \\ B \ \leqslant F'(\_) \end{array} \right\}, \ \mathcal{F}(\Delta)
\end{array}
$$

Figure 3.18: Encoding System $F_{<:}$ in Wyv_{core} without functions

3.2.2 Encoding System $F_{<:}$: Recursive Types

An encoding of the bounded quantification of System $F_{<:}$ is relatively simple, and somewhat to be expected given the presence of both dependent function types and path dependent types in Wyv_{core}. What might be surprising however, is that $F_{<:}^{N}$ subtyping can be encoded in Wyv_{core} even in the absence of dependent function types. Figure 3.18 provides an encoding of the critical aspects of $F_{<:}^{N}$ using only path dependent types and recursive types.

The encoding of Figure 3.18 differs from that of 3.15 in that it does not employ dependent function types to encode either arrow types ($T_1 \ \rightarrow T_2$), or universally quantified type ($\forall(\alpha \leqslant T_1).T_2$). Encodings of both types is achieved by using recursive type refinements and path dependent types. The key to the encoding is the fact recursive types and path dependent types include the two critical components of $F_{<:}^{N}$: (i) contra-variance in the subtyping of lower bounded type members, and (ii) quantification over types (technically Wyv_{core} includes quantification over terms, but since terms may contain types, this distinction is blurred for the purposes of encoding $F_{<:}^{N}$).

The final piece of the encoding in Figure 3.18 is the encoding of the Δ environment of type variable bounds. Note that the encoding of Δ in Figure 3.18 differs from that of Figure 3.15 in that the encoding of the type includes two type declarations, A and B. This is because the encoding of universally quantified types includes two members A and B. I leave the bound of B unspecified because B is never accessed (all type selections are of the form $\alpha.A$), but for the purposes of the proof of subtype preservation, it is enough that there exists some type for which the encoding is sound.

Theorem 3.2.3 ($\mathcal{F}'$ preserves subtyping)**.** $\Delta \vdash T_1 \ <: \ T_2 \Rightarrow \mathcal{F}(\Delta) \vdash \mathcal{F}'(T_1) \ <: \ \mathcal{F}'(T_2)$

Proof. Let n be the depth of some $\mathrm{F}^{N}_{<:}$ subtyping derivation $\Delta \vdash T_1 \ <: \ T_2$. We proceed by induction on n.
Induction Hypothesis: for any subtype derivation $\Delta' \ \vdash T_1' \ <: \ T_2'$ of depth $m \ < \ n$, then $\mathcal{F}'(\Delta') \vdash \mathcal{F}'(T_1') \ <: \ \mathcal{F}'(T_2')$.
Using the induction hypothesis, we show that for each subtype rule in Figure 3.16, $\mathcal{F}'$ preserves subtyping.

Case 1 (Base Case: NTop)**.** Trivial.

Case 2 (Base Case: NRefl)**.** Trivial.

Case 3 (Base Case: NVar)**.** Similar reasoning to the equivalent case in Theorem 3.2.1.

Case 4 (Inductive Case: NArr)**.**

$$\frac{\Delta \vdash T_2 \ <: \ T_1 \qquad \Delta \vdash T_1' \ <: \ T_2'}{\Delta \vdash T_1 \rightarrow T_1' \ <: \ T_2 \rightarrow T_2'} \quad (\textsc{NArr})$$

By the induction hypothesis, we have

$$\mathcal{F}'(\Delta) \vdash \mathcal{F}'(T_2) \ <: \ \mathcal{F}'(T_1) \qquad \mathcal{F}'(\Delta) \vdash \mathcal{F}'(T_1') \ <: \ \mathcal{F}'(T_2')$$

By environment weakening in *Wyv*$_{core}$ we get

$$\mathcal{F}'(\Delta), \_ : \left\{ \begin{array}{l} A \ \geqslant F'(T_1) \\ B \ \leqslant F'(T_1') \end{array} \right\} \vdash \mathcal{F}'(T_2) \ <: \ \mathcal{F}'(T_1)$$

$$\mathcal{F}'(\Delta), \_ : \left\{ \begin{array}{l} A \ \geqslant F'(T_1) \\ B \ \leqslant F'(T_1') \end{array} \right\} \vdash \mathcal{F}'(T_1') \ <: \ \mathcal{F}'(T_2')$$

By S$_\sigma$-Upper and S$_\sigma$-Lower we get

$$\mathcal{F}'(\Delta), \_ : \left\{ \begin{array}{l} A \ \geqslant F'(T_1) \\ B \ \leqslant F'(T_1') \end{array} \right\} \vdash (A \ \geqslant \mathcal{F}'(T_1)) \ <: \ (A \ \geqslant \mathcal{F}'(T_2))$$

$$\mathcal{F}'(\Delta), \_ : \left\{ \begin{array}{l} A \ \geqslant F'(T_1) \\ B \ \leqslant F'(T_1') \end{array} \right\} \vdash (B \ \leqslant \mathcal{F}'(T_1')) \ <: \ (B \ \leqslant \mathcal{F}'(T_2'))$$

Thus, by S-Rfn we get the desired result:

$$\mathcal{F}'(\Delta) \vdash \left\{ \begin{array}{l} A \ \geqslant F'(T_1) \\ B \ \leqslant F'(T_1') \end{array} \right\} \ <: \ \left\{ \begin{array}{l} A \ \geqslant F'(T_2) \\ B \ \leqslant F'(T_2') \end{array} \right\}$$

Case 5 (Inductive Case: NAll)**.**

$$\frac{\Delta \vdash T_2 \ <: \ T_1 \qquad \Delta, \alpha \leqslant T_2 \vdash T_1' \ <: \ T_2'}{\Delta \vdash \forall(\alpha \leqslant T_1).T_1' \ <: \ \forall(\alpha \leqslant T_2).T_2'} \quad (\text{NAll})$$

By the induction hypothesis, we have

$$\mathcal{F}(\Delta) \vdash \mathcal{F}(T_2) \ <: \ \mathcal{F}(T_1)$$

$$\mathcal{F}(\Delta), \alpha \ : \ \left\{ \alpha \Rightarrow \begin{array}{l} A \ \geqslant F'(T_2) \\ B \ \leqslant F'(T_2') \end{array} \right\} \vdash \mathcal{F}(T_1') \ <: \ \mathcal{F}(T_2')$$

By weakening we get

$$\mathcal{F}(\Delta), \alpha \ : \ \left\{ \alpha \Rightarrow \begin{array}{l} A \ \geqslant F'(T_2) \\ B \ \leqslant F'(T_2') \end{array} \right\} \vdash \mathcal{F}(T_2) \ <: \ \mathcal{F}(T_1)$$

By S$_\sigma$-Upper and S$_\sigma$-Lower we get

$$\mathcal{F}'(\Delta), \alpha : \left\{ \alpha \Rightarrow \begin{array}{ll} A & \geqslant F'(T_2) \\ B & \leqslant F'(T_2') \end{array} \right\} \vdash (A \ \leqslant \mathcal{F}'(T_2)) \ <: \ (A \ \leqslant \mathcal{F}'(T_1))$$

$$\mathcal{F}'(\Delta), \alpha : \left\{ \alpha \Rightarrow \begin{array}{ll} A & \geqslant F'(T_2) \\ B & \leqslant F'(T_2') \end{array} \right\} \vdash (B \ \geqslant \mathcal{F}'(T_2')) \ <: \ (B \ \geqslant \mathcal{F}'(T_1'))$$

Thus, by S-Rfn we get:

$$\mathcal{F}'(\Delta) \vdash \left\{ \alpha \Rightarrow \begin{array}{ll} A & \geqslant F'(T_2) \\ B & \leqslant F'(T_2') \end{array} \right\} \ <: \ \left\{ \alpha \Rightarrow \begin{array}{ll} A & \geqslant F'(T_1) \\ B & \leqslant F'(T_1') \end{array} \right\}$$

Finally, by a combination of S$_\sigma$-Lower and S-Rfn we get the desired result:

$$\mathcal{F}'(\Delta) \vdash \neg \left\{ \alpha \Rightarrow \begin{array}{ll} A & \geqslant F'(T_1) \\ B & \leqslant F'(T_1') \end{array} \right\} \ <: \ \neg \left\{ \alpha \Rightarrow \begin{array}{ll} A & \geqslant F'(T_2) \\ B & \leqslant F'(T_2') \end{array} \right\}$$

□

The encoding of Figure 3.18 is more troublesome than that of Figure 3.15 in attempting to derive a restriction that is acceptably expressive. A natural inclination might be to apply a similar restriction to that of Kernel F$_{<:}$, i.e. require invariance on the contra-variant aspect of subtyping. For dependent function types, this implies invariant argument types, a restriction that still permits many instances of valuable expressiveness. For recursive types however, this implies invariant lower bounds, which represents a much larger loss of expressiveness.

A common instance of expressiveness using lower bounds features writing to data structures as in the following append function.

```
def append[E](e : E,
              l : List[Elem >: E]) : bool =
  match l with
    nil = e :: nil
    e'::t = e'::(append e t)
```

Enforcing invariance on lower bounds in this case would restrict usage of `append` on lists with element type equal to the element being appended. Thus `append[Int](5, new List[Int])` would typecheck, but `append[Int](5, new List[Number])` would not. This is clearly not a useful restriction, thus any decidable and practical subset of Wyvern would need to address this issue.

3.2.3 Encoding Java Generics

Grigore [40] showed that Java Generics was Turing complete by demonstrating that Java subtyping could encode a Turing machine. The mechanism that allows such a reduction is the recursive manner in which Java generics can be used coupled with the contra-variant subtyping of Java wildcards, and Java's multiple inheritance (but not necessarily multiple instantiation inheritance). Wyv_{core} exhibits similar kinds of variance and recursive patterns to those present in Java's generics using path dependent types and type refinement. Below is an example of a recursive type definition (derived from an example defined by Greenman et al.) that leads to looping during subtyping:

```
interface Equatable <T extends Object >{}
interface List <T extends Object >
  extends Equatable <List <? extends Equatable <? super T>>>{}
class ArrayList<T extends Object> implements List<T>{}
class Tree extends ArrayList <Tree >{}

public class Function {
    public void func(Equatable<? super Tree> e){
    }
    public static void main(String[] args) {
        Tree t = new Tree (); Function f = new Function ();
        f.func(t); // loops during subtyping
    }
}
```

$$\frac{C\langle t_1, \ldots, t_n\rangle \ <::^* D\langle t'_1, \ldots, t'_m\rangle \qquad t''_1 \ <: \ t'_1 \ \ldots \ t''_m \ <: \ t'_m}{C\langle t_1, \ldots, t_n\rangle \ <: D\langle t''_1, \ldots, t''_m\rangle}$$

Figure 3.19: Fragment of Java Subtyping (Grigore 2017)

The definition of `List` above contains a form of recursion, as `List` is defined in terms of itself. Looping is not evidence for undecidability. While the above code fragment will cause `javac` to crash with a `StackOverflow` error, it will be rejected by the Eclipse IDE's type checker. There are however other more complex examples that will cause Eclipse to crash just by opening them in Eclipse. Such an error has significantly more serious implications for tools programmers rely on.

Grigore defined a fragment of Java's subtyping and demonstrated it Turing complete. A Java type t under their formulation is a class C with n (where $n \geq 0$) type parameters, all of which are contra-variant. Classes are specified in a class table, a list of class inheritance declarations of the form: $C\langle X_{C1}, \ldots, X_{Cn}\rangle <:: D\langle t_1, \ldots, t_m\rangle$. The associated subtyping is given in Figure 3.19. Note: $<::$ is overloaded in Grigore's formulation of Java to include inheritance lookup from the class table coupled with substitution of type variables. Further, $<::^*$ is the transitive and reflexive closure of this second meaning of $<::$. Thus, the usage of $<::^*$ in Figure 3.19 does not introduce any free variables.

Java has what is called use-site variance, in that the variance used during subtyping (co- or contra-variance) is specified at the use site. Other languages, such C$^\sharp$, Scala, Kotlin and Ceylon all use declaration-site variance to specify the variance used during subtyping. In Java, covariant subtyping is indicated by `? extends` at the use site, while contra-variant subtyping is indicated by `? super`. The subtyping provided in Figure 3.19 has declaration site variance (technically every type parameter is contra-variant). Grigore points out that declaration site variance can be encoded in Java by replacing every usage of a co-variant parameter with `? super _`, and every usage of a contra-variant parameter with `? super _` (only `? super _` in that case of Figure 3.19).

By brief inspection of Figure 3.19, it seems unlikely that Wyv_{core} should be able to encode it. It has already been mentioned that path dependent types subsume the type parameters of Java Generics. Type definitions are analogous to classes, and type refinements for path dependent types is analogous to supplying a type parameter to a class. The type extension defined in Figure 3.11 is conceptually similar to class inheritance in so far as it defines type extension. The feature that is missing is multiple inheritance style hierarchies (see Section 2.2.1).

While Wyv_{core} does not contain multiple inheritance, it is possible to encode it at a type level, if not for objects. Java inheritance hierarchies are captured by class inheritance declarations in the class table. Wyv_{core} does not have a class table, instead type definitions are stored within an environment, and type definitions are not restricted to a single definition.

```
type ListAPI = {
  type List[Elem <: ⊤] <: Equatable[List[Elem]]{
    def equals : List[Elem] -> bool
  }
  type List[Elem <: ⊤] <: Traversable[Elem]{
    def foreach[U] : (Elem -> U) -> ⊤
  }
}
```

The code snippet above provides two definitions for `List`, one that extends `Equatable` and another that extends `Traversable`. Any usage of `List` within the context of `ListAPI` can treat `List` as an extension of either type. This has no bearing concrete objects, as in order to instantiate `ListAPI`, the instantiating object would have to contain a single `List` definition that satisfied both

$$
\begin{array}{lcl}
\mathcal{W}(X_{Ci}) & = & z_C.X_C \\
\mathcal{W}(C\langle t\rangle) & = & z_0.C\left\{\ X_C \ \geqslant \mathcal{W}(t)\ \right\} \\
 & & \\
\mathcal{W}(\emptyset) & = & \emptyset \\
\mathcal{W}(\Delta, \{X_1 \geqslant t_1, \ldots, X_n \geqslant t_n\}_D) & = & \mathcal{W}(\Delta), z_0.D\left\{ z_D \Rightarrow \begin{array}{ll} X_1 & \geqslant \mathcal{W}(t_1) \\ X_1 & \geqslant \bot \\ \vdots & \\ X_n & \geqslant \mathcal{W}(t_n) \\ X_n & \geqslant \bot \end{array} \right\} \\
 & & \\
\mathcal{W}(C\langle X_C\rangle <::\ D\langle t\rangle; CT) & = & C \ \leqslant z_0.D\left\{ z_C \Rightarrow \begin{array}{ll} X_D & \geqslant \mathcal{W}(t) \\ X_C & \geqslant \bot \end{array} \right\}, \mathcal{W}(CT)
\end{array}
$$

Figure 3.20: Encoding Java (Grigore 2017) in Wyv_{core}

of the above definitions, which is not possible in Wyv_{core} (unless `Equatable` somehow subtyped `Traversable`, or vice versa). While not useful for typing objects, it is still possible to construct questions of subtyping around them. Thus, for the purposes of encoding Java's subtyping, we can consider Wyv_{core} as containing multiple inheritance.

An encoding ($\mathcal{W}(t) = \tau$) of the Java types used in Figure 3.19 into Wyv_{core} types is provided in Figure 3.20. Firstly, the encoding of Figure 3.20 makes a distinction between a top level self variable (z_0) and all self variables internal to type definitions (z_C). A type variable is encoded as a type selection on the internal self variable. A class type is encoded as a non-recursive type refinement, whose base type is a type selection on the top level self variable. The type refinement refines the type definitions within the base type with the types provided as parameters. An encoding for class tables is also provided, encoding class definitions as type definitions. A class definition $C\langle\ldots\rangle$ extending some other class $D\langle\ldots\rangle$ is encoded as a type definition named C upper bounded by a type refinement on $z_0.D$. The re-

$$\frac{X \geqslant t \ \in \Delta \qquad \Delta \vdash t' \ <: \ t}{\Delta \vdash t' \ <: \ X} \qquad \frac{\begin{array}{c} C\langle t_C \rangle \ \ <::^* \ \ [t_1/X_1, \ldots, t_n/X_n]D\langle t' \rangle \\ \Delta, \{X_1 \geqslant t_1, \ldots, X_n \geqslant t_n\}_D \vdash t \ \ <: \ \ t' \end{array}}{\Delta \vdash C\langle t_C \rangle \ \ <: \ \ D\langle t \rangle}$$

Figure 3.21: Alternative form of Java Subtyping (Grigore 2017) with Explicit Substitutions

finement defines all of C's type members as lower bounded by $\bot$, and refines the members belonging to D as lower bounded by the relevant types. The encoded class table is included in all subtyping as an initial environment: $\Gamma_{CT} = z_0 \ : \ \{z_0 \Rightarrow \mathcal{W}(CT)\}$.

Before we can show that subtyping in Java Generics is a subset of subtyping in *Wyv*$_{core}$, some preliminary definitions and lemmas are needed. First we define a modified version of Java subtyping that explicitly includes the transitivity of the inheritance step, rather than relying on a transitive inheritance relation. This more closely resembles the type extension of *Wyv*$_{core}$. Further, we restrict the number of type parameters to $n \leq 1$ to make the proofs simpler, but it would be just as easy to construct an equivalent proof for an arbitrary number of type parameters. The definition of the alternate form of Java subtyping can be found in Figure 3.21.

The subtyping of Figure 3.21 still differs in a significant way to that of *Wyv*$_{core}$: type variables are substituted out in Java, while in *Wyv*$_{core}$, selection types are stored within an environment. In order to capture this difference, I define type Java extension substitution implied by the Java variant of Figure 3.21.

Lemma 3.2.1 (*Wyv*$_{core}$ Flattening subsumes Java Substitution)**.** *For all* C*,* D*, and* $t_C, t_D, t_1, X_1, \ldots, t_n, X_n$*, if* $C\langle t_C \rangle \ \ <::^* \ \ [t_1/X_1, \ldots, t_n/X_n]D\langle t_D \rangle$*,* *then*

$$\Gamma_{CT} \vdash \mathcal{W}(C\langle t_C\rangle) \;\; \leqslant ::^* \;\; flat(\mathcal{W}(D\langle t'_D\rangle), \left\{\begin{array}{ll} X_n & \geqslant \mathcal{W}(t_n) \\ X_n & \geqslant \bot \\ \vdots & \\ X_1 & \geqslant \mathcal{W}(t_1) \\ X_1 & \geqslant \bot \end{array}\right\}, z)$$

Proof. By induction on the structure of the proof of $C\langle t_C\rangle \;\; <::^* \;\; [t_1/X_1, \ldots, t_n/X_n]D\langle t_D\rangle$:

Case 1. The base case is the single inheritance step:

$$C\langle t_C\rangle \;\; <:: \;\; D\langle t_D\rangle$$

By the definition of $<::$, there exists t'_D such that $C\langle X_C\rangle \;\; <:: \;\; D\langle t'_D\rangle \in CT$ and $t_D = [t_C/X_C]t'_D$. Thus, by the definition of $\mathcal{W}(CT)$, we have

$$\Gamma_{CT} \vdash z_0 \;\; : \;\; \{C \;\; \leqslant z_0.D \left\{ z_C \Rightarrow \begin{array}{ll} X_D & \geqslant \mathcal{W}(t'_D) \\ X_C & \geqslant \bot \end{array} \right\}\}$$

And by E-Rfn we get

$$\mathcal{W}(C\langle t_C\rangle) = z_0.C\{X_C \;\; \geqslant \mathcal{W}(t_C)\}$$
$$\Gamma_{CT} \vdash z_0.C\{X_C \;\; \geqslant \mathcal{W}(t_C)\} \;\; \leqslant:: \;\; z_0.D \left\{ z_C \Rightarrow \begin{array}{ll} X_D & \geqslant \mathcal{W}(t'_D) \\ X_C & \geqslant \bot \\ X_C & \geqslant \mathcal{W}(t_C) \end{array} \right\}$$

Which is equal to the desired result: $flat(\mathcal{W}(D\langle t'_D\rangle), \left\{ \begin{array}{ll} X_C & \geqslant \bot \\ X_C & \geqslant \mathcal{W}(t_C) \end{array} \right\}, z_C)$

Case 2. The inductive case is the transitive inheritance step:

$$C\langle t_C\rangle \;\; <:: \;\; [t_n/X_n]C'\langle t'_C\rangle \qquad C'\langle t'_C\rangle \;\; <::^* \;\; [t_1/X_1, \ldots, t_{n-1}/X_{n-1}]D\langle t_D\rangle$$

By the induction hypothesis we know that

$$\Gamma_{CT} \vdash \mathcal{W}(C'\langle t'_C \rangle) \;\leqslant ::^{*}\; flat(\mathcal{W}(D\langle t'_D \rangle), \left\{ \begin{array}{l} X_{n-1} \;\geqslant \mathcal{W}(t_{n-1}) \\ X_{n-1} \;\geqslant \bot \\ \vdots \\ X_1 \;\geqslant \mathcal{W}(t_1) \\ X_1 \;\geqslant \bot \end{array} \right\}, z)$$

It is an easy step, using the same reasoning as in Case 1 to demonstrate that $\Gamma_{CT} \vdash z_0.C\{z_C \Rightarrow \mathcal{W}(t_C)\} \;\leqslant ::\; z_0.D \left\{ z_C \Rightarrow \begin{array}{l} X_{C'} \;\geqslant \mathcal{W}(t'_C) \\ X_C \;\geqslant \bot \\ X_C \;\geqslant \mathcal{W}(t_C) \end{array} \right\}$, and thus by successive applications of E-Rfn we get

$$\Gamma_{CT} \vdash \mathcal{W}(C\langle t_C \rangle) \;\leqslant ::^{*}\; flat(\mathcal{W}(D\langle t'_D \rangle), \left\{ \begin{array}{l} X_n \;\geqslant \mathcal{W}(t_n) \\ X_n \;\geqslant \bot \\ \vdots \\ X_1 \;\geqslant \mathcal{W}(t_1) \\ X_1 \;\geqslant \bot \\ X_C \;\geqslant \mathcal{W}(t_C) \\ X_C \;\geqslant \bot \end{array} \right\}, z)$$

□

Note, that for the purposes of well-formedness we include members indicating unassigned type parameters as $X \;\geqslant \bot$ in the definitions of types. These members are kept around during type extension, but can be ignored during subtyping as there is no Java type t such that $\mathcal{W}(t) \;<:\; \bot$, and since type variables in Java are always substituted out, there will always be another more specific member $X \;\geqslant \mathcal{W}(T)$ in the environment.

Theorem 3.2.4. *For all t, t' and Δ, if $\Delta \vdash t \;<:\; t'$, then $\Gamma_{CT}, \mathcal{W}(\Delta) \vdash \mathcal{W}(t) \;<:\; \mathcal{W}(t')$*

Proof. We proceed by induction on the structure of the proof of $\Delta \vdash t \ <: \ t'$.

Case 1 (Base Case: $\Delta \vdash t' \ <: \ X$)**.**

$$\frac{X \geqslant t \ \in \Delta \qquad \Delta \vdash t' \ <: \ t}{\Delta \vdash t' \ <: \ X}$$

By the definition of $\mathcal{W}(\Delta)$, there exists some z such that $\mathcal{W}(X) = z.X$, and

$$\Gamma_{CT}, \mathcal{W}(\Delta) \vdash z \ : \ \_\{\_ \Rightarrow X \ \geqslant \mathcal{W}(t)\}$$

Further, by the induction hypothesis, we get

$$\Gamma_{CT}, \mathcal{W}(\Delta) \vdash \mathcal{W}(t') \ <: \ \mathcal{W}(t)$$

Thus, by S-Lower, we get the desired result:

$$\Gamma_{CT}, \mathcal{W}(\Delta) \vdash \mathcal{W}(t') \ <: \ \mathcal{W}(X)$$

Case 2 (Inductive Case: $\Delta \vdash C\langle t_C \rangle \ <: \ D\langle t \rangle$)**.**

$$\frac{\begin{array}{c} C\langle t_C \rangle \ <::^* \ [t_1/X_1, \ldots, t_n/X_n]D\langle t' \rangle \\ \Delta, \{X_1 \geqslant t_1, \ldots, X_n \geqslant t_n\}_D \vdash t \ <: \ t' \end{array}}{\Delta \vdash C\langle t_C \rangle \ <: \ D\langle t \rangle}$$

By the induction hypothesis, we get

$$\Gamma_{CT}, \mathcal{W}(\Delta), z_D : z_0.D \left\{ z_D \Rightarrow \begin{array}{l} X_1 \ \geqslant \mathcal{W}(t_1) \\ X_1 \ \geqslant \bot \\ \vdots \\ X_n \ \geqslant \mathcal{W}(t_n) \\ X_n \ \geqslant \bot \end{array} \right\} \vdash \mathcal{W}(t) \ <: \ \mathcal{W}(t')$$

By Lemma 3.2.1 we get

$$\Gamma_{CT}, \mathcal{W}(\Delta) \vdash \mathcal{W}(C\langle t_C \rangle) \ \leqslant::^* \ z_0.D \left\{ z_D \Rightarrow \begin{array}{l} X_D \ \geqslant \mathcal{W}(t') \\ X_1 \ \geqslant \mathcal{W}(t_1) \\ X_1 \ \geqslant \bot \\ \vdots \\ X_n \ \geqslant \mathcal{W}(t_n) \\ X_n \ \geqslant \bot \end{array} \right\}$$

Thus by successive applications of S-Ext, and finally an application of S-Rfn, we get the desired result.

□

Chapter 4

Material/Shape Separated Wyv_{core}

In Chapter 3 I define the Wyv_{core} calculus, and demonstrate several ways in which subtyping is undecidable in Wyv_{core}. In this Chapter, I define the basis for decidable subtyping of variants of Wyv_{core}, a Material/Shape Separation for Wyv_{core} based on the work of Greenman et al. [39] for Java. We construct a separation on types, classifying them as either *Materials* or *Shapes*, and restricting where Shapes may be used.

This Chapter begins by discussing the tension between (i) the need to restrict recursion in types to obtain decidable subtyping, and (ii) the expressiveness of nominality, in order to build a design and high level understanding of the Material/Shape Separation in Wyv_{core} (Section 4.1). I finally formalize this separation using a graphical representation of types (Section 4.2).

4.1 Designing a Material/Shape Separation for Wyv_{core}

In Section 2.2.4 I described the Material/Shape separation [39] designed by Greenman et al. for Java. In this section I design a similar separation for

Wyv_{core}. The immediate question is: why choose the Material/Shape separation, and not take some other approach to subtype decidability for Wyv_{core}? Firstly, there is an obvious similarity between the recursion of Wyv_{core} and Java: in both languages types can explicitly be defined recursively. If we recall the example used in Section 2.2.4.

```
interface Eq<T>
class List<E> implements Eq<List<Eq<E>>>
```

As I described in Section 2.2.4, recursive type definitions in Java are defined using two language features, inheritance and parametric polymorphism. In the example above, `List` is recursively defined as inheriting from `Eq<List<Eq<E>>>`. Wyv_{core} does not feature inheritance, however it does include the aspect of Java's inheritance that we are concerned with: nominal subtyping (via abstract type members, see 2.2.4). Similarly, Wyv_{core} does not feature parametric polymorphism, but type members can be used to model type parameters (see 2.2.1). In fact, to some degree, type members model all of the language features required for recursion in Java: type definitions, nominality, and polymorphism. These similarities between recursive type definitions in Java and Wyv_{core} suggest that the Material/Shape separation might be a good fit for Wyv_{core}.

The Material/Shape separation also has the advantage that it does not seem to exclude most real world Java programs. Wyvern is a young language, and thus it is not possible to know what patterns might be commonly used in real world Wyvern programs, and therefore whether the results of the survey by Greenman et al. can be extrapolated to Wyvern. It is however, useful to have the subset of expressiveness defined by the Material/Shape separation of Greenman et al. as a baseline of expressiveness.

In order to design a Material/Shape separation for Wyv_{core} analogous to the one Greenman et al. designed for Java, we need to be able to identify type definitions (class or interface declarations), type extensions (`extends` or `implements` in Java), and parametric polymorphism (class generics in Java).

As we have already noted, in Wyv_{core}, all these features are modeled using type members. Below I rewrite the `List/Eq` example in Wyv_{core}.

```
{ z0 =>
  type Eq <: { type T >: ⊥ }
  type List <: z0.Eq{ z =>
    type E
    type T >: z0.List{ type E = z0.Eq{ type T = z.E } }
}
```

In Java, type definitions, type extension, and parametric polymorphism, all have separate syntactic forms, and thus were easily differentiated. Each of these features play a role in defining the Material/Shape separation. In Wyv_{core}, we must judge by the particular usage of a type member to determine if it is a type definition (e.g. `type List` or `type Eq`), or if it is a polymorphic member definition (e.g. `type E`). We need to also differentiate between type extension usages (e.g. `type List <: Eq{ ... }`), and specification of polymorphic type members (e.g. `type T = z0.List{ ... }`). However, even if we were able to identify the intent for a particular type usage, unfortunately nominal subtyping in Wyv_{core} is not equivalent to the nominal subtyping of Java. In this section, I will address these two problems in the design of a Material/Shape separation. Since the Material/Shape separation is largely intuitive, we will use this intuition (informed by the Material/Shape separation for Java) rather than a formal definition, in discussing these design decisions. I will wait until I have elaborated on the design before presenting the final formal design of the Material/Shape separation.

4.1.1 Relaxing The Material/Shape Separation

A central difficulty in defining a Material/Shape separation in Wyv_{core}, is that in general, there is no obvious way to differentiate between the type definition of `z0.List` and `z.T`. There is no way to determine the intent of a type declaration: is it a definition, or a polymorphic type? In the Material/Shape separation defined for Java these different usages require different rules: Shapes may be used in type definitions, but not in polymorphic positions. In Java this distinction is explicitly included as part of the language syntax. In *Wyv*$_{core}$, these two usages are syntactically indistinguishable.

As an example, consider the `List`/`Eq` example. As yet, we have not formally defined a separation on Materials and Shapes in *Wyv*$_{core}$, but for now, I will look to the same example in Java, and assume that we intend `Eq` to be a Shape, and `List` a Material. In the Java version of the example, it seems easy to restrict Shapes (`Eq` in this case) from use in polymorphic types, but allow them in type definitions. In *Wyv*$_{core}$, these two language features use the same syntactic form: a type member. As a result, if we try to apply the Material/Shape separation defined for Java to *Wyv*$_{core}$, we quickly run into problems. In the case of `Eq`, using our classification of Materials and Shapes, we would correctly disallow the use of `z0.Eq` from the definition of `z.T`, but incorrectly disallow `z0.Eq` from the definition of `z0.List`. In fact, classifying `Eq` as a Shape would make `Eq` essentially unusable in any position, significantly reducing the expressiveness of *Wyv*$_{core}$. While technically usable as an argument type to a function, the function would be unusable since nothing could subtype it, as no type could be defined as extending `Eq`.

The solution to the problem is to relax the restriction on Shapes to only prohibiting them from use in lower bounds. This relaxed restriction is still enough to build an argument of decidability, but not before we address the other issues related to the Material/Shape separation.

4.1.2 Nominality in *Wyv*$_{core}$ vs Java

Recursion in *Wyv*$_{core}$ is possible because of the ability to assign names to types, and to use the name of a type in its definition. Naming of types is also fundamental to another property of *Wyv*$_{core}$, subtype nominality. Nominality allows programmers to easily construct new types by referring to and extending existing types. Nominality also allows programmers to restrict data to a specific origin.

What is notable about the proofs of undecidability in Section 3.2 is less the undecidability of Wyv_{core} through the presence of dependent function types (Section 3.2.1), but rather the two proofs of undecidability that rely on recursive types and nominality. One through the encoding of System $F_{<:}$ (Section 3.2.2), and the other through the encoding of Java (Section 3.2.3). In this Section I discuss nominality in Wyv_{core} and compare it to the nominality in Java. I identify two specific ways nominality in Wyv_{core} differs from Java.

Example: A Nominal List Implementation

Nominal subtyping in Wyv_{core} refers to the nominality that is exhibited by subtyping in DOT (see Section 2.2.1): through *abstract type members*. That is, type members that have been defined in an abstract manner, by way of an upper bound ($L \leqslant \tau$), and not a specific type ($L = \tau$). Thus an abstract type member represents an abstraction of data, and not a specific datatype. To demonstrate this form of nominality, I recall the `ListAPI` example from Section 2.2.4:

```
type ListAPI = { self =>
        type Equatable <: ⊤{ z =>
                type E >: ⊥
                def equals : E -> bool
        }
        type List <: self.Equatable{ z =>
                type E = self.List{ type T = z.T }
                type T <: ⊤
        }
        def nil : self.List{type T = ⊥}
        def cons : (x : {type Elem <: ⊤},
                    e : x.Elem,
                    l : List{type T <: x.Elem}) ->
                    self.List{type T <: x.Elem}
}
```

```
val myListAPI : ListAPI
```

In the above example, we define the interface for a `ListAPI`. `Equatable` is defined using an abstract definition, and is used in the definition of `List`, another abstract type. `myListAPI` is a field that refers to a specific instance of `ListAPI`. The abstract specification of `Equatable` means that subtyping is in effect nominal. Since `Equatable` is defined without a lower bound (in fact the lower bound is actually $\bot$), the only types that may subtype it are either $\bot$, or type refinements on `Equatable` (such as `Equatable{type E = Int}`). Put another way, only types that are explicitly defined as subtyping `Equatable` are considered subtypes of `Equatable`, i.e. nominal subtyping. Similarly, since `List` is an abstract type member, it too exhibits nominal subtyping. This form of nominality allows for the abstraction visible in the function definitions of `nil` and `cons`. Programmers are provided an interface for constructing their own lists, but are unable to access the specific implementation provided by any particular instance of `ListAPI`. Through such a pattern, Wyv_{core} is able to model abstract data types [32].

Nominal Super-typing

Nominal subtyping in Wyv_{core} differs from that of Java in the restrictions placed on which types may super-type another type. As the `ListAPI` example demonstrated, nominal type definitions in Wyv_{core} only restrict which types may subtype a type, and not which types may super-type a type. In Java however, nominality is defined by the class hierarchy, and subtyping is restricted from above as well as from below. A Java type `C<_>` may only be super-typed by types defined by classes that are direct super classes of `C`. In Wyv_{core} however, the `List` type from the `ListAPI` example, may only be subtyped by types that explicitly refine `List`, but may be super-typed by any type that super-types its upper bound. Thus the following subtype relationship holds:

```
List <: { def equals :⊥ -> bool }
```

There is no equivalent subtype relationship in Java. The reader, might however notice that the above statement is not exactly true and the type hierarchies of Wyv_{core} and Java are more alike than I am letting on. The type `{ def equals :⊥ -> bool }` is in fact a type refinement on $\top$, a type that `List` does transitively extend (via `Equatable`). The difference of course is that the equivalent of $\top$ in Java is `Object`, and subtyping in Java does not allow for type parameters on `Object`, let alone arbitrary refinements. In fact, type parameters in Java are restricted to those defined for the class of the type in question. It is therefore perhaps more accurate to say that the nominal type hierarchy of Java restricts arbitrary super-typing of type refinements (type parameters in Java) in a way that Wyv_{core} does not.

This difference is not especially problematic in general, and in fact allows for the kinds of useful expressiveness that structural subtyping is intended for. It does however have implications when designing restrictions on recursive nominal subtyping. This distinction between the semantics of nominality in Wyv_{core} and Java is important for the Material/Shape separation due to the implicit restriction Java's nominal subtyping places on Shapes. A central property of subtyping in Material/Shape separated Java is that Shapes may only be super-typed by other Shapes, and as a result when comparing type parameters during subtyping, Shapes are removed, and are guaranteed to never occur again. In Wyv_{core}, as I have discussed, structural subtyping means that we can not be sure that only Shapes super-type other Shapes.

Extending Wyv_{core}'s Nominal Semantics

We now address the lack of nominal super-typing in Wyv_{core}. The gist of our approach is to provide both nominal subtyping and nominal super-typing for Shapes in Wyv_{core}. We do this by again looking to Java's nominal subtyping. Java restricts how type parameters are used, and subsequently how they are subtyped. What I mean by this is that Java type parameters are unique to the class they are defined within. There is no way to induce structural subtyping of type parameters without employing the class that they are defined within.

Given a class declaration:

```
class C<E> extends D
```

There is no way in Java to define a type of unknown class, that contains a type parameter named `E`, and subsequently compare this type to some `C<T>`. Such a type makes no sense in Java. Structural subtyping is however possible in *Wyv*$_{core}$:

```
C{type E <: ...} <: {type E <: ... }
```

The ellipses represent unimportant type information.

The Material/Shape separation designed for Java achieves decidability by identifying those types critical in constructing recursive inheritance (Shapes), and syntactically ensuring that there exists a maximal depth beyond which Shapes do not occur (by restricting their usage in all but top level positions of types). The definition of Shapes derives from they way they are used, more specifically the way they are parameterized (or extended in *Wyv*$_{core}$) as part of recursive inheritance definitions. It is not just the Shapes that are involved in instances recursive inheritance, but also the type parameters, and how they relate to the type being defined. In other words, while `Equatable` might be identified as a Shape in the `ListAPI` example, the problematic aspect of that example is the way `List` is defined in terms of itself using the type parameter `E` of `Equatable`.

The decidability argument that arises from the Material/Shape separation depends on the fact that through nominality, Java restricts subtyping of parameters to specific cases. The structural subtyping inherent in *Wyv*$_{core}$ allows for Shapes to be "side-stepped". That is, the problematic relationship can be inspected during the evaluation of some subtype algorithm without going through the Shape that acts as a guard. In the previous `ListAPI` example, the problematic relationship was between `List`, and the type parameter (type member in *Wyv*$_{core}$) `E`, originally defined within `Equatable`. In the example below, the Shape `Equatable` can be side-stepped entirely:

```
List{type E <: _} <: {type E <: _}
```

$$\frac{\Gamma \vdash x\ :\{M\ \ \leqslant \tau\}}{\Gamma \vdash x.M \leqslant :: \tau}\quad (\text{E-M}\textsc{at})$$

$$\frac{\Gamma \vdash x\ :\ \{S\ \ \leqslant \tau\} \qquad \tau\ \textit{is a shape}}{\Gamma \vdash x.S \leqslant :: \tau}\quad (\text{E-S}\textsc{ha})$$

Figure 4.1: Material/Shape Separated *Wyv*$_{core}$ Type Extension

In order to enforce a greater degree of nominality on Shapes, I define a modification of the type extension semantics defined in Figure 3.11. Figure 4.1 differentiates type extension for materials and shapes. The type extension rule for materials is as normal, but the type extension rule for shapes requires that the extended type also be a shape. Note: this does not require all Shapes to be defined in terms of Shapes, rather that a Shape can only be judged as extending another Shape. This prevents arbitrary structural subtyping of shapes, and ensures the nominality required for the Material/Shape separation. That is, with this fix, Shapes in *Wyv*$_{core}$ now have both nominal subtyping and nominal super-typing.

Nominality for Concrete Types

Another difference between the nominality in *Wyv*$_{core}$ and Java, is that in *Wyv*$_{core}$, nominal subtyping is reserved for abstract types, and not concrete types. Concrete types (i.e. referring to a specific type: $L = \tau$), i.e. types that might be instantiated, do not exhibit nominal subtyping. In Java, concrete types are defined by class definitions, while abstract types are defined using either interface or abstract class definitions. Both of these types exhibit nominal subtyping. This might not normally be a problem (as we might want to allow structural subtyping for concrete types), but as we have identified, Shapes may only be used in a nominal type definition, i.e. the upper bound of a type. In Wyv_{core}, a concrete type definition ($L = \tau$) defines both its' upper and lower bound (as τ). A prohibition on the used of Shapes in lower bounds means that concrete types may not be defined in using Shapes. This severely restricts the usefulness of Shapes. As an example, while we can define the abstract type `List` as an extension of `Eq`, we could not define a concrete `ArrayList` using `Eq`, meaning that we could not usefully implement `ListApi`. Our solution to this problem is to introduce a new syntactic form for type members that allows nominal subtyping, and thus the use of Shapes, for concrete types in Wyv_{core}. I will now discuss this problem in greater detail, and introduce our new syntactic form.

Note: thus far I have not defined a term syntax or operational semantics for Wyv_{core}, as we are largely concerned with the interactions of types. For the purposes of examples, I assume a term syntax, typing properties and operational semantics similar to that of DOT 2016. I will elaborate on these semantics when necessary, but will not if the details are largely unimportant to the example.

At runtime, `myListAPI` must be initialized with a concrete object of a concrete type (that is, not abstract). Below I provide a minimal example of such an object. I omit the implementation of the methods `nil` and `cons` along with the internals of the types as we are only interested in how the types relate to each other (hence the ellipses).

```
val myListAPI = new ListAPI{ self =>
        type Eq = ⊤{ z => ... }
        type List = self.Eq{ z => ... }
        ...
}
```

The types `myListAPI.Equatable` and `myListAPI.List` do not exhibit nominal subtype behaviour. As I have already discussed, Shapes as defined for Wyv_{core} (and Java), are an inherently nominal concept, and in Wyv_{core}, may not be used in a lower bound. As a result, the definition of `List` above is

$$\begin{array}{lll} \sigma & ::= & \textbf{Declaration Type} \\ & \vdots & \\ & L \preceq \tau & \textit{nominal definition} \end{array}$$

$$\frac{\Gamma \vdash x \ : \ \{L \ \preceq \ \tau\}}{\Gamma \vdash x \ : \ \{L \ \leqslant \tau\}} \quad (\text{T-Nom})$$

$$\frac{\Gamma \vdash \tau_1 \ <: \ \tau_2}{\Gamma \vdash L \ \preceq \ \tau_1 \ <: \ L \ \leqslant \tau_2} \quad (\text{S}_\sigma\text{-Upper}_{\preceq})$$

$$\Gamma \vdash L \ \preceq \ \tau \ <: \ L \ \preceq \ \tau \quad (\text{S}_\sigma\text{-Nominal})$$

Figure 4.2: Concrete Nominal Type Definitions

illegal due to the use of a Shape (`Eq`) in it's lower bound. The solution we introduce is an alternate syntactic form for type members: a concrete, but nominal definition. I define this extension to the syntax and semantics in Figure 4.2.

Selections on nominal types may be treated as selections on upper bounds (T-Nom) in the same way that selections on exact types can, but critically they are not treated as having a lower bound, and thus can't be subtyped except by reflexivity or type refinement. In most cases selections on nominally defined types behave just like upper bounded types. They differ in that subtyping of their definitions is invariant during subtyping (S_σ-Nominal). This means that objects can be initialized with nominally defined types, something that is not true of abstractly defined types. From the perspective of the Material/Shape separation, this means that they may contain shapes in the way that specific types ($L \ = \ \tau$) cannot, but can be used during object initialization in the way that abstract upper bounded types cannot. We are now able to rewrite the earlier example of `MyListAPI`:

```
new ListAPI{
        type Eq = { ... }
        type List ≼ Eq{ self => ... }
}
```

Note the nominal form ($\preceq$) used to define `List`, as opposed to the specific type ($=$) in the code fragment presented earlier in this section.

4.2 Formalizing The Material/Shape Separation in Wyv_{core}

In this section I formalize the Material/Shape separation for Wyv_{core}. To help formally address the distinction between different type usages in Wyv_{core}, I develop a graphical representation of types similar to that of Greenman et al.. The Material/Shape separation designed for Java also uses a graphical representation of types, however those graphs elide much of the type information. In developing a Material/Shape separation into Wyv_{core}, it is useful to elaborate this information for both clarity, and to help make explicit the language features (type definition, type extension, and parametric polymorphism) represented by each.

4.2.1 Type Graphs in Wyv_{core}

In Figure 4.3 I define a syntax for type definition graphs in Wyv_{core}. I define the vertex (Definition 4.2.1) and edge (Definition 4.2.2) sets, along with the type definition graph (Definition 4.2.3) for a specific Java environment.

Definition 4.2.1 (Wyv_{core} Type Definition Vertices)**.** *For a given environment* Γ*, we define the set of vertices as*

$$\begin{array}{lll}
V & ::= & \textbf{Vertex} \\
 & z :: \texttt{type}\ L & \textit{type definition} \\
 & z :: \texttt{val}\ l & \textit{value definition} \\
 & \leqslant\ \tau & \textit{upper bound} \\
 & \geqslant\ \tau & \textit{lower bound} \\
 & =\ \tau & \textit{exact type} \\
 & \preceq\ \tau & \textit{nominal type} \\
 & :\ \tau & \textit{value type} \\
 & \tau & \textit{type usage}
\end{array}
\qquad
\begin{array}{lll}
E & ::= & \textbf{Edge} \\
 & V \rightarrow V & \textit{bound} \\
 & V \overset{\texttt{def}}{\rightarrow} V & \textit{definition} \\
 & V \overset{\{\}}{\rightarrow} V & \textit{member} \\
 & V \overset{\{\}}{\underset{x.L}{\rightarrow}} V & \textit{extension member} \\
 & V \overset{\forall}{\rightarrow} V & \textit{parameter} \\
 & V \overset{\texttt{ret}}{\rightarrow} V & \textit{return} \\
 & V \overset{\leqslant::}{\rightarrow} V & \textit{extension}
\end{array}$$

Figure 4.3: Syntax for *Wyv*$_{core}$ Type Definition Graphs

$$\mathcal{V}_\Gamma \triangleq \left\{ V \middle| \begin{array}{l}
V = z :: \texttt{type}\ L, \exists\ r\ \tau,\ \Gamma \vdash z : \{L\ r\ \tau\}, r \in \{\leqslant, \geqslant, =, \preceq\} \vee \\
V = z :: \texttt{val}\ l,\ \Gamma \vdash z : \{l : \ldots\} \vee \\
V = (r\ \tau), r \in \{\leqslant, \geqslant, =, \preceq\}, \exists z\ L, \Gamma \vdash z\ :\ \{L\ r\ \tau\} \vee \\
V = (:\ \tau), \exists z\ l, \Gamma \vdash z\ :\ \{l : \tau\} \vee \\
V = \tau, (r\ \tau) \in \mathcal{V}_\Gamma\ r \in \{\leqslant, \geqslant, =, \preceq, :\} \vee \\
V = \tau_1, (\forall(x : \tau_1).\tau_2) \in \mathcal{V}_\Gamma \vee \\
V = \tau_2, (\forall(x : \tau_1).\tau_2) \in \mathcal{V}_\Gamma \vee \\
V \in \mathcal{V}_{\Gamma, z:\tau\{z \Rightarrow \overline{\sigma}\}\}}, \tau\{z \Rightarrow \overline{\sigma}\} \in \mathcal{V}_\Gamma \vee \\
V \in \mathcal{V}_{\Gamma, x:\tau_1}, \forall(x : \tau_1).\tau_2 \in \mathcal{V}_\Gamma \vee \\
V = r\ \tau, \tau' \in \mathcal{V}_\Gamma, \Gamma \vdash \tau'\ \leqslant::\ \tau,\ r\ \in \{\leqslant, \geqslant, =, \preceq\} \vee \\
V = \tau, \tau' \in \mathcal{V}_\Gamma, \Gamma \vdash \tau'\ \leqslant::\ \tau
\end{array} \right\}$$

Definition 4.2.2 (*Wyv*$_{core}$ Type Definition Edges)**.** *For a given environment* Γ*, with vertex set* $\mathcal{V}_\Gamma$ *we define the set of edges as*

$$\mathcal{E}_\Gamma \triangleq \left\{ E \;\middle|\; \begin{array}{l} E = z :: \texttt{type}\ L \rightarrow r\ \tau,\ \Gamma \vdash z : \{L\ r\ \tau\} \vee \\ E = r\ x.L\{z \Rightarrow \overline{\sigma}\} \xrightarrow[x.L]{\{\}} \tau, L'\ r'\ \tau\ \in\ \overline{\sigma},\ r\ r' \in \{\leqslant, \geqslant, =, \preceq\} \vee \\ E = r\ x.L\{z \Rightarrow \overline{\sigma}\} \xrightarrow[x.L]{\{\}} \tau, l : \tau\ \in\ \overline{\sigma},\ r \in \{\leqslant, \geqslant, =, \preceq\} \vee \\ E = r\ x.L\{z \Rightarrow \overline{\sigma}\} \overset{\texttt{def}}{\rightarrow} x :: \texttt{type}\ L,\ r\ \in \{\leqslant, \geqslant, =, \preceq\} \vee \\ E = x.L\{z \Rightarrow \overline{\sigma}\} \overset{\{\}}{\rightarrow} \tau, L'\ r'\ \tau\ \in\ \overline{\sigma},\ r\ r' \in \{\leqslant, \geqslant, =, \preceq\}, \\ \quad x.L\{z \Rightarrow \overline{\sigma}\} \in \mathcal{V}_\Gamma \vee \\ E = x.L\{z \Rightarrow \overline{\sigma}\} \overset{\{\}}{\rightarrow} \tau, l : \tau\ \in\ \overline{\sigma},\ r \in \{\leqslant, \geqslant, =, \preceq\}, \\ \quad x.L\{z \Rightarrow \overline{\sigma}\} \in \mathcal{V}_\Gamma \vee \\ E = x.L\{z \Rightarrow \overline{\sigma}\} \overset{\texttt{def}}{\rightarrow} x :: \texttt{type}\ L,\ r\ \in \{\leqslant, \geqslant, =, \preceq\}, \\ \quad x.L\{z \Rightarrow \overline{\sigma}\} \in \mathcal{V}_\Gamma \vee \\ E = \forall(x : \tau_1).\tau_2 \overset{\forall}{\rightarrow} \tau_1, \forall(x : \tau_1).\tau_2 \in \mathcal{V}_\Gamma \vee \\ E = \forall(x : \tau_1).\tau_2 \overset{\texttt{ret}}{\rightarrow} \tau_2, \forall(x : \tau_1).\tau_2 \in \mathcal{V}_\Gamma \vee \\ E = r\ \tau' \overset{\leqslant::}{\rightarrow} \tau, \Gamma \vdash \tau'\ \leqslant::\ \tau,\ r\ \in \{\leqslant, \geqslant, =, \preceq\} \vee \\ E = \tau' \overset{\leqslant::}{\rightarrow} \tau, \Gamma \vdash \tau'\ \leqslant::\ \tau \end{array} \right\}$$

Definition 4.2.3 (Wyv_{core} Type Definition Graph)**.** *For a given environment* Γ*, we define a type definition graph as*

$$\mathcal{G}_\Gamma \triangleq (\mathcal{V}_\Gamma, \mathcal{E}_\Gamma)$$

What definitions 4.2.1, 4.2.2, and 4.4 capture is a graphical representation of the relationships between type definitions in a program. That is, for an environment Γ, type graph $\mathcal{G}_\Gamma$ is the graph whose vertices (V in Figure 4.3) represent types and their definitions, while edges represent relationships between those types and definitions. A vertex (V) is either a type definition (of the form $z :: \texttt{type}\ L$), a value defintion ($z :: \texttt{val}\ l$), an upper bound ($\leqslant\ \tau$), a lower bound ($\geqslant\ \tau$), an exact type ($=\ \tau$), a nominal bound ($\preceq\ \tau$), a value type ($:\ \tau$), or a type usage (τ). An edge (E) between these vertices either connects a type definition (or value definition) to it's bound ($V \rightarrow V$), a selection type to it's definition ($V \overset{\texttt{def}}{\rightarrow} V$), a type refinement to a member within the refinement ($V \overset{\{\}}{\rightarrow} V$), a type refinement on the top level of a bound

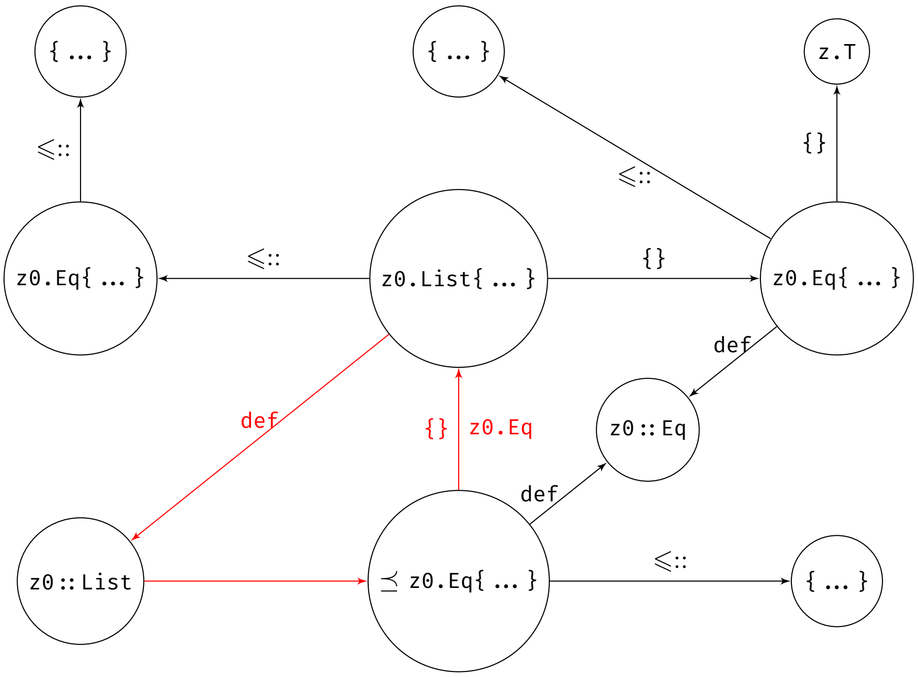

Figure 4.4: Type Graph (Definition) of `List` in *Wyv*$_{core}$

to a member within the refinement $(V \xrightarrow[x.L]{\{\}} V)$, a dependent function type to it's argument type $(V \overset{\forall}{\to} V)$ and it's return type $(V \overset{\texttt{ret}}{\to} V)$, or a type to it's extended type $(V \overset{\leqslant::}{\to} V)$.

Using Definitions 4.2.1, 4.2.2, and 4.2.3, I construct the graph in Figure 4.4, which provides a full representation of the `List` and `Eq` types in the *Wyv*$_{core}$ version (including the syntactic extension of Section 4.1.2) of the `List`/`Eq` example.

```
new ListAPI{ z0
  type Eq = ⊤{
    type E >: ⊥
    def equals : E -> boolean
  }
  type List ≼ z0.Eq{ z =>
```

```
    type T <: ⊤
    type E = z0.List{type T = z0.Eq{type E = z.T}}
  }
}
```

For the purposes of space, I elide the details of some vertices in Figure 4.4 by the use of elipses. Note, as with the graph in Figure 2.23, there is only one cycle (`class List` → ... $\overset{\texttt{def}}{\rightarrow}$ `class List`), with only one edge labeled with a type name (`Eq`). In fact, from the perspective of cycles, these two graphs are equivalent. The differences are apparent in that all of the relevant type relationships are visible in Figure 4.4.

The recursion of types in Wyv_{core} means that their expansion is not finite, but a critical feature of type graphs is that they are finite. Recursion in Wyv_{core} is due to the presence type members. Usages of type members in type graphs point back to the definition of the type member, and do not introduce a new type.

4.2.2 An Extended Syntax for Material/Shape Separated Wyv_{core}

Given the definition of type graphs, we are now able to define the Material/Shape separation for Wyv_{core}. As with the original Material/Shape separation designed for Java, we define Shapes in Definition 4.2.4.

Definition 4.2.4. *Let $\mathcal{T}$ be all types defined as a type member of some object in an environment Γ. We define as shapes, some set $\mathcal{S}$ of types such that removing all edges labeled with an element of $\mathcal{S}$ from the type graph usage of Γ results in an acyclic graph. We define all remaining types, the set $\mathcal{M}$, as materials. $\mathcal{M}$ and $\mathcal{S}$ are disjoint, and $\mathcal{T} = \mathcal{M} \cup \mathcal{S}$.*

The result of Definition 4.2.4 is that every type cycle in Material/Shape separated Wyv_{core} is guaranteed to contain at least one Shape. Thus, while

$$\begin{array}{lll}
L & ::= & \textbf{Type Name} \\
 & M & \mathit{material} \\
 & S & \mathit{shape}
\end{array}$$

Figure 4.5: Separation on Material/Shape Type Names

$$\begin{array}{llr}
\tau & ::= & \textbf{Type} \\
 & \eta\{z \Rightarrow \overline{\sigma}\} & \\
 & x.L\{z \Rightarrow \overline{\delta}\} & \\
 & x.M & \\
 & \forall(x:\tau).\tau & \\
 & \top & \\
 & \bot & \\
 & & \\
\sigma & ::= & \textbf{Decl. Type} \\
 & M \ \leqslant \tau & \\
 & M \ \geqslant \eta & \\
 & M \ = \ \eta & \\
 & L \ \preceq \ \tau & \\
 & l : \eta &
\end{array}
\qquad
\begin{array}{llr}
\eta & ::= & \textbf{Material Type} \\
 & \eta\{z \Rightarrow \overline{\delta}\} & \\
 & x.M & \\
 & \forall(x:\eta).\eta & \\
 & \top & \\
 & \bot & \\
 & & \\
\delta & ::= & \textbf{Mat. Decl. Type} \\
 & M \ \leqslant \eta & \\
 & M \ \geqslant \eta & \\
 & M \ = \ \eta & \\
 & L \ \preceq \ \eta & \\
 & l : \eta &
\end{array}$$

Figure 4.6: Material/Shape Separated Wyv_{core} Syntax

the expansion of a type in Material/Shape separated Wyv_{core} might be infinite, there is guaranteed to be a finite "Shape Depth". It is this measure that I will use to demonstrate termination of subtyping in the variants of Wyv_{core} described in the rest of this thesis.

Finally, Figure 4.6 provides a Material/Shape separated syntax for Wyv_{core}, that defines where Shape may be used, and where they are prohibited. The Material/Shape separation defines where shapes may be used safely. Types are separated into general types (τ) that may contain either Materials or Shapes, and pure material types (η) that do not contain any Shapes, along with general declaration types (σ) and pure material declaration types (δ).

Shapes are restricted to function types ($\forall(x : \tau).\tau$) and upper bounded type definitions only ($L \leqslant \tau$ and $L \preceq \tau$). Shapes are also restricted to refined usages ($x.L\{z \Rightarrow \overline{\delta}\}$). This ensures that any subtype of a shape must use the S-Refine rule, and thus that the restriction on shape extension in Figure 4.1 applies in subtyping all shape usages. Further, all refinements on shapes are restricted to materials ($\overline{\delta}$).

Chapter 5

A General Decidability Argument

In this Chapter I use the Material/Shape separation defined in Chapter 4 to formalise a general argument that I use to prove subtype decidability for type systems defined throughout the rest of the thesis. In Chapter 4, I defined a Material/Shape separation for Wyv_{core}, that ensures that for any type definition, there is a finite "shape depth". That is, for any Wyv_{core} type in some environment, if we recursively unfold that type, there is a finite depth at which a Shape occurs. This "Shape Depth" is the depth measure that is used in proving subtype decidability for the decidable variants presented later in this thesis, and I define such a depth measure measure later in this Chapter.

A Material/Shape separation does not however ensure decidability. There is a remaining problem of environment narrowing. For any type, the "Shape Depth " is dependent on a specific environment. If the environment changes during subtyping, we cannot be sure that the shape depth does not change. This Chapter introduces an intermediate representation of Wyv_{core} types called Wyv_{expand}, that removes environment narrowing. Types in Wyv_{expand} represent the (potentially infinite) expansion of types in Wyv_{core}. This representation, along with terminating subtype algorithm is the basis for all of the decidability proofs found in the rest of this thesis. That is, if it can be shown that subtyping for any variant of Wyv_{core} is reducible to subtyping of this variant, then it follows that subtyping for that variant is decidable.

In Section 5.1 I define $Wyv_{\texttt{expand}}$, an expansion of Wyv_{core} types. In Section 5.2 I define a subtype algorithm (`subtype`) for types in $Wyv_{\texttt{expand}}$. In Section 5.3 I define the shape depth measure, on types in $Wyv_{\texttt{expand}}$, and then use the shape depth measure to prove termination of `subtype`. I subsequently provide some brief examples of how shape depth relies on an environment to be well behaved during subtyping. All variants of Wyv_{core} in later Chapters deal with this problem of environment narrowing in some way. Finally, in Section 5.5.1, I present one such variant of Wyv_{core} called Wyv_{fix}, describe how Wyv_{fix} deals with environment narrowing, and prove subtyping for Wyv_{fix} decidable by demonstrating a sound and complete reduction from subtyping for Wyv_{fix} to subtyping of $Wyv_{\texttt{expand}}$.

5.1 $Wyv_{\texttt{expand}}$: Recursive Expansion of Wyv_{core} Types

As I have already mentioned, environment narrowing in Wyv_{core} is a problem for subtype decidability. All of the variants of Wyv_{core} that are defined in the rest of this thesis introduce some restriction on environment narrowing. In this Section I define a syntax for expanded types in Wyv_{core} called $Wyv_{\texttt{expand}}$. Expanding types by unfolding the type members to their definitions makes the definition of a subtype algorithm much easier, as we are not required to maintain an environment, and don't have to consider environment narrowing. Further, a full expansion allows an easier definition of the finite measure Shape Depth (Section 5.3). In all of the variants of Wyv_{core} that I define in the rest of the thesis include some restriction on environment narrowing, and target $Wyv_{\texttt{expand}}$ as an intermediate language in order to demonstrate subtype decidability. In Figure 5.1 I define a syntax for $Wyv_{\texttt{expand}}$.

Types in $Wyv_{\texttt{expand}}$ largely resemble types in Wyv_{core}, except in those

$$\begin{array}{lll} T & ::= & \qquad\qquad \textbf{Type} \\ & \forall(x:T).T & \\ & \{z \Rightarrow \overline{D}\} & \\ & x.L\{z \Rightarrow \overline{D}\} \mapsto \overline{T} & \\ & x.L \mapsto \overline{D} & \\ & \top & \\ & \bot & \end{array} \qquad \begin{array}{lll} D & ::= & \qquad \textbf{Decl. Type} \\ & L \leqslant T & \\ & L \geqslant T & \\ & L = T & \\ & L \preceq T & \\ & l : T & \end{array}$$

Figure 5.1: Type Syntax for $\mathit{Wyv}_{\mathsf{expand}}$

cases that include a type member selection. This is evident by looking at the syntax defined in Figure 5.1. A syntactic term is either a type (T) or a declaration type (D). Types that are constructed from selection types differ from those of $\mathit{Wyv}_{\mathit{core}}$, in that they include the definitions of the type selection as a syntactic subterm. A selection type ($x.L \mapsto \overline{D}$) includes the set of type member definitions ($\overline{D}$) associated with the type selection ($x.L$) . A type refinement on a selection type ($x.L\{z \Rightarrow \overline{D}\} \mapsto \overline{T}$) includes the set of extended types ($\overline{T}$) represented by the type refinement.

Types in $\mathit{Wyv}_{\mathsf{expand}}$ represent an expansion of types in $\mathit{Wyv}_{\mathit{core}}$, and as such, I define a mapping that captures this in Figure 5.2. The mapping of $\mathit{Wyv}_{\mathit{core}}$ types to $\mathit{Wyv}_{\mathsf{expand}}$ types in Figure 5.2 is fairly self-explanatory. There are two notable properties of Figure 5.2: (i) a type T might be infinite due to the recursion present in $\mathit{Wyv}_{\mathit{core}}$ (I address this directly in Section 5.3), and (ii) Figure 5.2 constitutes an algorithm for expanding $\mathit{Wyv}_{\mathit{core}}$ types into $\mathit{Wyv}_{\mathsf{expand}}$ types since it is syntax directed, and there is no ambiguity of rules. This second property is important because this means it is always possible to generate the expansion of a type.

$$\boxed{\Gamma \vdash \tau \ \longmapsto \ T}$$

$$\Gamma \vdash \top \ \longmapsto \ \top \qquad \Gamma \vdash \bot \ \longmapsto \ \bot \qquad \frac{\Gamma \vdash \overline{\sigma} \ \longmapsto \ \overline{D} \qquad \overline{\sigma} = \{\sigma | \Gamma \vdash x \ : \ \{\sigma\}, \ id(\sigma) = L\}}{\Gamma \vdash x.L \ \longmapsto \ x.L \mapsto \overline{D}}$$

$$\frac{\Gamma, z : x.L\{z \Rightarrow \overline{\sigma}\} \vdash \overline{\sigma} \ \longmapsto \ \overline{D} \qquad \Gamma \vdash \overline{\tau} \ \longmapsto \ \overline{T} \qquad \overline{\tau} = \{\tau | \Gamma \vdash x.L\{z \Rightarrow \overline{\sigma}\} \ <:: \ \tau\}}{\Gamma \vdash x.L\{z \Rightarrow \overline{\sigma}\} \ \longmapsto \ x.L\{z \Rightarrow \overline{D}\} \mapsto \overline{T}}$$

$$\frac{\Gamma, z : \{z \Rightarrow \overline{\sigma}\} \vdash \overline{\sigma} \ \longmapsto \ \overline{D}}{\Gamma \vdash \{z \Rightarrow \overline{\sigma}\} \ \longmapsto \ \{z \Rightarrow \overline{D}\}} \qquad \frac{\Gamma \vdash \tau_1 \ \longmapsto \ T_1 \qquad \Gamma, x : \tau_1 \vdash \tau_2 \ \longmapsto \ T_2}{\Gamma \vdash \forall(x : \tau_1).\tau_2 \ \longmapsto \ \forall(x : T_1).T_2}$$

$$\frac{\Gamma \vdash \tau \ \longmapsto \ T}{\Gamma \vdash L \ \leqslant \tau \ \longmapsto \ L \ \leqslant T} \qquad \frac{\Gamma \vdash \tau \ \longmapsto \ T}{\Gamma \vdash L \ \geqslant \tau \ \longmapsto \ L \ \geqslant T} \qquad \frac{\Gamma \vdash \tau \ \longmapsto \ T}{\Gamma \vdash L \ = \ \tau \ \longmapsto \ L \ = \ T}$$

$$\frac{\Gamma \vdash \tau \ \longmapsto \ T}{\Gamma \vdash L \ \preceq \ \tau \ \longmapsto \ L \ \preceq \ T} \qquad \frac{\Gamma \vdash \tau \ \longmapsto \ T}{\Gamma \vdash l : \tau \ \longmapsto \ l : T}$$

Figure 5.2: Unfolding Types in Wyv_{core} to Types in Wyv_{expand}

5.2 A Subtyping Algorithm for Wyv_{expand}

I now define a subtype algorithm for subtyping between types in Wyv_{expand}: `subtype`. If T_1 and T_2 are types in Wyv_{expand}, then subtyping between them is captured by

$$\mathtt{subtype}(T_1, T_2) = \mathtt{true}$$

Before providing the final subtype algorithm, I define some helper functions in Algorithm 1 that assit in deriving the lower bound of a selection type:

- `ub`: the upper bound function takes a type, and returns a set of defined upper bounds for that type.

- `lb`: the lower bound function takes a type, and returns a set of defined lower bounds for that type.

```
ub($T$)
    switch $T$ do
        case $x.L \mapsto \overline{D}$ do $\bigcup_{D \in \overline{D}}$ ub($D$);
        case $L \leqslant T'$ do $\{T'\}$;
        case $L \geqslant \_$ do $\{\top\}$;
        case $L = T'$ do $\{T'\}$;
        case $L \preceq T'$ do $\{T'\}$;
        otherwise do $\emptyset$;
    end
lb($T$)
    switch $T$ do
        case $x.L \mapsto \overline{D}$ do $\bigcup_{D \in \overline{D}}$ lb($D$);
        case $L \leqslant \_$ do $\{\bot\}$;
        case $L \geqslant T'$ do $\{T'\}$;
        case $L = T'$ do $\{T'\}$;
        case $L \preceq T'$ do $\{T'\}$;
        otherwise do $\emptyset$;
    end
```

Algorithm 1: Wyv_{expand} Subtyping Helper Functions

The definitions for `ub` and `lb` are fairly straightforward, and simply pattern match on the definitions defined for a selection type, returning the set of bounds (either upper or lower bounds).

The Subtyping Algorithm

Subtyping of types in Wyv_{expand} is given in Algorithms 2 and 3. Both algorithms are based on the Wyv_{core} subtype rules defined in Figure 3.14. Unlike types that exist within an environment, the unfolded types of Wyv_{expand} include all type information as syntactic sub-components. Thus, the main algorithm, Algorithm 2 is syntactically driven, and each successive call to `subtype` (or `subtypeDecl`) is made on syntactic components of the previous call. Note: at the moment there is no indication that `subtype` should terminate, as expanded types are potentially infinite, and thus each component is

```
subtype($T_1, T_2$)
    switch $T_1$ do
        case $\top$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x.L \mapsto \overline{D}$ do $\bigvee_{T \in \texttt{lb}(T_2)}$ subtype($T_1$, $T$) ;
                otherwise do false;
            end
        case $\bot$ do true;
        case $x_1.L_1 \mapsto \overline{D}_1$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x_2.L_2 \mapsto \overline{D}_2$ do $(x_2 = x_2 \wedge L_1 = L_2) \vee (\bigvee_{T \in \texttt{ub}(T_1)}$ subtype($T$, $T_2$))$\vee$
                  ($\bigvee_{T \in \texttt{lb}(T_2)}$ subtype($T_1$, $T$)) ;
                otherwise do $\bigvee_{T \in \texttt{ub}(T_1)}$ subtype($T$, $T_2$) ;
            end
        case $\forall(x : S_1).U_1$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x.L \mapsto \overline{D}$ do $\bigvee_{T \in \texttt{lb}(T_2)}$ subtype($T_1$, $T$) ;
                case $\forall(x : S_2).U_2$ do (subtype($S_2$, $S_1$)) $\wedge$ subtype($U_1, U_2$) ;
                otherwise do false;
            end
        case $T_1'\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x.L \mapsto \overline{D}$ do $\bigvee_{T \in \texttt{lb}(T_2)}$ subtype($T_1$, $T$) ;
                case $T_2'\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$ do
                    if $T_1' == T_2'$ then
                        subtypeDecl($\overline{D}_1, \overline{D}_2$)
                    else
                        $\bigvee_{T \in \overline{T}_1}$ subtype($T$, $T_2$)
                    end
                otherwise do false;
            end
    end
```

Algorithm 2: $Wyv_{\texttt{expand}}$ Subtyping Algorithm

```
subtypeDecl($D_1, D_2$)
    switch $D_1, D_2$ do
        case $L_1 \leqslant T_1, L_2 \leqslant T_2$ do
            if $L_1 = L_2$ then subtype($T_1, T_2$);
            else false;
        case $L_1 = T_1, L_2 \leqslant T_2$ do
            if $L_1 = L_2$ then subtype($T_1, T_2$);
            else false;
        case $L_1 \geqslant T_1, L_2 \geqslant T_2$ do
            if $L_1 = L_2$ then subtype($T_2, T_1$);
            else false;
        case $L_1 = T_1, L_2 \geqslant T_2$ do
            if $L_1 = L_2$ then subtype($T_2, T_1$);
            else false;
        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 false;
        case $L_1 \preceq T_1, L_2 \preceq T_2$ do
            $L_1 = L_2 \wedge T_1 = T_2$
        case $l_1 : T_1, l_2 : T_2$ do
            if $l_1 = l_2$ then subtype($T_1, T_2$);
            else false;
        otherwise do false;
    end
```

Algorithm 3: $Wyv_{\texttt{expand}}$ Declaration Subtyping Algorithm

$$
\begin{array}{lcl}
\mathcal{S}(\top) & = & 0 \\
\mathcal{S}(\bot) & = & 0 \\
\mathcal{S}(x.S \;\mapsto\; \overline{D}) & = & 0 \\
\mathcal{S}(x.M \;\mapsto\; \overline{D}) & = & 1 + \mathit{max}(\mathcal{S}(\overline{D})) \\
\mathcal{S}(\forall(x : T_1).T_2) & = & 1 + \mathit{max}(\mathcal{S}(T_1), \mathcal{S}(T_2)) \\
\mathcal{S}(T\{z \Rightarrow \overline{D}\} \;\mapsto\; \overline{T}) & = & \mathit{if}\ T\ \mathit{is\ a\ shape} \\
 & & \mathit{then}\ 0 \\
 & & \mathit{else}\ 1 + \mathit{max}(\mathcal{S}(\overline{T}), \mathcal{S}(\overline{D})) \\
\mathcal{S}(L \;\leqslant T) & = & 1 + \mathcal{S}(T) \\
\mathcal{S}(L \;\geqslant T) & = & 1 + \mathcal{S}(T) \\
\mathcal{S}(L \;=\; T) & = & 1 + \mathcal{S}(T) \\
\mathcal{S}(L \;\preceq\; T) & = & 1 + \mathcal{S}(T) \\
\mathcal{S}(l : T) & = & 1 + \mathcal{S}(T)
\end{array}
$$

Figure 5.3: Shape Depth Measure on Type Graphs

not necessarily smaller than it's parent.

`subtype` takes two types as arguments, and returns either `true` or `false`. `subtypeDecl` takes two declaration types as arguments, and returns either `true` or `false`. Both `subtype` and `subtypeDecl` consist of switch statements that pattern match on the syntactic form of the first argument, and then the second.

As I mentioned at the beginning of this section, Algorithms 2 and 3 are based on the subtype rules of $\mathit{Wyv}_{\mathit{core}}$, and attempt to perform a proof search by inverting the rules. Thus, the premises of the $\mathit{Wyv}_{\mathit{core}}$'s subtype rules become sub-calls to Algorithms 2 and 3. Due to this correlation, it is possible to draw comparisons between rules in Figure 3.14 and cases in Algorithms 2 and 3. In fact, subsequent proofs of decidability for variants of $\mathit{Wyv}_{\mathit{core}}$ aim to demonstrate this exact correlation.

5.3 Shape Depth

In Chapter 4 I defined a Material/Shape separation on types that restricts Shapes from certain uses. In this Section I will use the Material/Shape separation to define a finite measure on types. As I have already noted, expanded types in Wyv_{expand} may be infinite due to the potential for recursive definitions. This makes it difficult to nail down a finite measure that strictly decreases during subtyping.

While types in Wyv_{expand} may be infinite, they do have a finite "Shape Depth". That is, an infinite type in Wyv_{expand} corresponds to a recursive type in Wyv_{core}. A recursively defined type in Wyv_{core} must exhibit a cycle in it's type graph (by Definition 4.2.4). By Definition 4.2.4, that cycle must include a Shape. Thus, for any vertex $\mathcal{V}$ in a type graph $\mathcal{G}$ there does not exist an infinite path that does not include a Shape. Inverting this line of reasoning, for any infinite type in Wyv_{expand}, there must be a finite maximum depth at which a Shape occurs. I define this depth in Figure 5.3

Below I define a set of separation properties on type graphs. These properties all follow directly from Definition 4.2.4 and the syntax defined in Figure 4.6, and can be statically checked apart from typing.

Separation Property 1. *All cycles in type graphs of type definitions contain at least one edge that is labelled with a Shape (by definition 4.2.4).*

Separation Property 2. *All type names are either Materials or Shapes (by Definition 4.2.4).*

Separation Property 3. *Lower bounds do not contain Shapes (by Figure 4.6).*

Separation Property 4. *Refinements on Shapes do not contain Shapes (by Figure 4.6).*

Since types in Wyv_{expand} are derived from types in Wyv_{core}, we could just as easily define $\mathcal{S}$ on Wyv_{core} types. However for ease of defining a subtyping algorithm, we define $\mathcal{S}$ on Wyv_{expand} types.

Theorem 5.3.1 ($\mathcal{S}$ is a Finite Measure)**.** *For all Γ, τ, $\mathcal{G}_\Gamma$, and T, such that $\Gamma \vdash \tau \quad \longmapsto \quad T$, and $\mathcal{G}_\Gamma$ is the type graph for Γ, if $\mathcal{G}_\Gamma$ observes the Material/Shape separation, then $\mathcal{S}(T)$ is finite.*

Proof. T is either

(a) Finite or

(b) Infinite.

If T is finite, then it follows immediately that $\mathcal{S}(T)$ is finite. If T is infinite, then it follows that τ contains a recursive definition, and thus for any type name used in this definition, there is a finite depth at which it occurs. By Separation Property 1 it follows that any recursive definition in τ contains a Shape, and thus there is a finite depth at which a Shape occurs in T. It follows directly that $\mathcal{S}(T)$ is finite. □

5.4 Termination of Subtyping Algorithm

In this Section I demonstrate that all calls to the `subtype` (and `subtypeDecl`) function are guaranteed to terminate if the arguments obey the Material/Shape separation. The proof of termination of `subtype` serves as a general proof of decidability for all forms of subtyping that are equivalent to `subtype`. That is, for all variants of *Wyv*$_{core}$ subtyping, if $\Gamma \vdash \tau_1 <: \tau_2 \iff \texttt{subtype}(T_1, T_2) = \texttt{true},$ *where* $\Gamma \vdash \tau_1 \longmapsto T_1$ *and* $\Gamma \vdash \tau_2 \longmapsto T_2$, subtyping is decidable. Unfortunately this does not hold for the *Wyv*$_{core}$ subtype rules defined in Chapter 3. *Wyv*$_{fix}$ defined in Section 5.5.1 and the variants defined in the following Chapters take different approaches to constructing variants of *Wyv*$_{core}$ subtyping that do observe this equivalence.

Theorem 5.4.1. *For any subtype check,* `subtype`(T_1, T_2) *there exists a maximum depth after which all calls to* `subtype` *have a pure material type on the right-hand side.*

Proof. By induction on $\mathcal{S}(T_1) + \mathcal{S}(T_2)$ it is demonstrable that there is a maximum depth at which either T_1 or T_2 with be a Shape. If either T_1 or T_2 is a Shape, `subtype` can only make progress by traversing the other side of the subtype check, and again by induction on the shape depth of that side, there is a maximum depth at which the other side is also a Shape. The subtype comparison of two shapes can only have two outcomes, failure (trivially satisfying the desired result), or a comparison of two pure material refinements, also satisfying the desired result. □

Theorem 5.4.2. *For any subtype check* `subtype`*(T_1, T_2), where T_2 is a pure material, all subsequent calls to* `subtype` *will also have a pure material on the right-hand side.*

Proof. Proceed by case analysis on T_2.

Case 1 ($T_2 = \top$)**.** Trivial.

Case 2 ($T_2 = \bot$)**.** Case analysis on T_1 demonstrates that any subsequent calls to `subtype` that might arise must have $\bot$ on the right-hand side, satisfying the desired result.

Case 3 ($T_2 = x.L \ \mapsto \ \overline{D}$)**.** Case analysis on T_1 demonstrates that any subsequent calls to `subtype` must either have T_2 on the right-hand side or the lower bound of T_2, another pure material, thus satisfying the desired result.

Case 4 ($T_2 = \forall(x : S_2).U_2$)**.** Case analysis on T_1 demonstrates that any subsequent calls to `subtype` must either have T_2 on the right-hand side, U_2 (a pure material) or the type bound of some other universally quantified type (another pure material), all satisfying the desired result.

Case 5 ($T_2 = T\{z \Rightarrow \overline{D}\} \ \mapsto \ \overline{T}$)**.** Case analysis on T_1 demonstrates that any subsequent calls to `subtype` must either have T_2 on the right-hand side, some $D_i \in \overline{D}$ (a pure material) or the lower bound of some other type member (another pure material), all satisfying the desired result.

□

Theorem 5.4.3. *Any subtype check* `subtype`(T_1, T_2)*, where T_2 is a pure material type, will terminate for all inputs.*

Proof. Our proof is by induction on the measure in Figure 5.3. I have already noted that $\mathcal{S}(T_1) + \mathcal{S}(T_2)$ is finite for all T_1 and T_2. This follows from the Material/Shape separation, specifically Property 1.

Now, the proof proceeds by induction on the size of $\mathcal{S}(T_1) + \mathcal{S}(T_2)$. For any call to `subtype`, it can be shown that any subsequent calls are strictly smaller. By Theorem 5.4.2 no Shapes will occur on the right-hand side of any subsequent calls to `subtype`. Since occurrences of Shapes must be refined, a Shape on the right-hand side may only be compared with another Shape on the left (which will not occur). Thus execution of `subtype` is bound by the finite shape depth on the right-hand side. □

Theorem 5.4.4 (Decidability)**.** *Any call to Algorithm 2,* `subtype`(T_1, T_2) *will terminate for all inputs.*

Proof. Follows directly from Theorems 5.4.1 and 5.4.3. □

5.5 Fixed Environments

I have already discussed environment narrowing as it affects type safety proofs in DOT (see Section 2.2.1), but environment narrowing is the central reason subtyping in *Wyv*$_{core}$ is not equivalent to subtyping of *Wyv*$_{\mathsf{expand}}$ Informally environment narrowing is the process whereby a variable within an environment is mapped to a more specific, or "narrower" type. Formally, preservation by environment narrowing is given in Property 5.5.1.

Property 5.5.1 (Environment Narrowing Preserves Subtyping)**.**

$$\frac{\Gamma \vdash \tau' <: \tau \qquad \Gamma(x) = \tau \qquad \Gamma \vdash \tau_1 <: \tau_2}{\Gamma[x\ : \tau'] \vdash \tau_1 <: \tau_2}$$

Property 5.5.1 states that if τ_1 subtypes τ_2 in Γ, then narrowing the type of some x preserves that subtype relationship.

Environment narrowing can occur in multiple parts of a type system like *Wyv*$_{core}$ or DOT. Environment narrowing during term reduction has been well covered for DOT. In *Wyv*$_{core}$, environment narrowing occurs during subtyping.

In this Section I define *Wyv*$_{fix}$, a decidable variant of *Wyv*$_{core}$ that is free from narrowing during subtyping entirely. As such, *Wyv*$_{fix}$ is not a subset of *Wyv*$_{core}$. I provide a proof of subtype decidability for *Wyv*$_{fix}$, and subsequently discuss the properties of type system.

5.5.1 *Wyv*$_{fix}$

Wyv$_{fix}$ is a variant of *Wyv*$_{core}$ that removes narrowing from the subtyping judgement. The strategy fixes types to the context in which they were defined. In *Wyv*$_{core}$, narrowing occurs in two places during subtyping: function subtyping and refinement subtyping:

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

$$\frac{\Gamma, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2}{\Gamma \vdash \tau\{z \Rightarrow \overline{\sigma}_1\} \ <: \ \tau\{z \Rightarrow \overline{\sigma}_2\}} \quad \text{(S-}\textsc{Rfn}\text{)}$$

In both cases, types are defined in one context and then subtyped in another. In S-All: τ_1' is defined in a context where $x : \tau_1$, but during subtyping, the type of x is narrowed to τ_2. In S-Rfn: declaration types $\overline{\sigma}_2$ are defined in an environment where z, the self variable, has type $\tau\{z \Rightarrow \overline{\sigma}_2\}$, but during subtyping the type of z is narrowed to $\tau\{z \Rightarrow \overline{\sigma}_1\}$. The key modification to the *Wyv*$_{fix}$ subtype rules over those of *Wyv*$_{core}$ is a double headed subtype judgement.

$$\Gamma_1 \vdash \tau_1 <: \tau_2 \dashv \Gamma_2$$

Subtyping takes place in the context of two environments, one for each type. All variable within τ_1 are typed in the context of Γ_1, and similarly all variables within τ_2 are typed in the context of Γ_2. Thus, any variation from the defining environment of a type is prevented during subtyping, as typings always take

$$\Gamma_1 \vdash \tau \ <: \ \top \dashv \Gamma_2 \quad \text{(S-\textsc{Top})} \qquad\qquad \Gamma_1 \vdash \bot \ <: \ \tau \dashv \Gamma_2 \quad \text{(S-\textsc{Bot})}$$

$$\Gamma_1 \vdash x.L \ <: \ x.L \dashv \Gamma_2 \quad \text{(S-\textsc{Rfl})} \qquad\qquad \frac{\begin{array}{c}\Gamma_1 \vdash x \ : \ \{L \ \leqslant \tau'\} \\ \Gamma_1 \vdash \tau' \ <: \ \tau \dashv \Gamma_2\end{array}}{\Gamma_1 \vdash x.L \ <: \ \tau \dashv \Gamma_2} \quad \text{(S-\textsc{Upper})}$$

$$\frac{\begin{array}{c}\Gamma_2 \vdash x \ : \ \{L \ \geqslant \tau'\} \\ \Gamma_1 \vdash \tau \ <: \ \tau' \dashv \Gamma_2\end{array}}{\Gamma_1 \vdash \tau \ <: \ x.L \dashv \Gamma_2} \quad \text{(S-\textsc{Lower})} \qquad \frac{\begin{array}{c}\Gamma_2 \vdash \tau_2 \ <: \ \tau_1 \dashv \Gamma_1 \\ \Gamma_1, x : \tau_1 \vdash \tau_1' \ <: \ \tau_2' \dashv \Gamma_2, x : \tau_2\end{array}}{\Gamma_1 \vdash \forall(x : \tau_1).\tau_1' \ <: \ \forall(x : \tau_2).\tau_2' \dashv \Gamma_2} \quad \text{(S-\textsc{All})}$$

$$\frac{\Gamma_1 \vdash \tau_1 \ \leqslant:: \ \tau \qquad \Gamma_1 \vdash \tau \ <: \ \tau_2 \vdash \Gamma_2}{\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \dashv \Gamma_2} \quad \text{(S-\textsc{Ext})}$$

$$\frac{\Gamma_1, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2 \dashv \Gamma_2, z : \tau\{z \Rightarrow \overline{\sigma}_2\}}{\Gamma_1 \vdash \tau\{z \Rightarrow \overline{\sigma}_1\} \ <: \ \tau\{z \Rightarrow \overline{\sigma}_2\} \dashv \Gamma_2} \quad \text{(S-\textsc{Rfn})}$$

$$\frac{\forall \sigma_2 \ \in \ \overline{\sigma}_2, \exists \sigma_1 \ \in \ \overline{\sigma}_1, \ \Gamma_1 \vdash \sigma_1 \ <: \ \sigma_2 \dashv \Gamma_2}{\Gamma_1 \vdash \sigma_1, \overline{\sigma}_1 \ <: \ \sigma_2, \overline{\sigma}_2 \dashv \Gamma_2} \quad \text{(S-\textsc{Decls})}$$

$$\frac{\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \vdash \Gamma_2}{\Gamma_1 \vdash L \leqslant\!/\!= \tau_1 \ <: \ L \ \leqslant \tau_2 \dashv \Gamma_2} \quad \text{(S}_\sigma\text{-\textsc{Upper})}$$

$$\frac{\Gamma_2 \vdash \tau_2 \ <: \ \tau_1 \vdash \Gamma_1}{\Gamma_1 \vdash L \geqslant\!/\!= \tau_1 \ <: \ L \ \geqslant \tau_2 \dashv \Gamma_2} \quad \text{(S}_\sigma\text{-\textsc{Lower})}$$

$$\frac{\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \vdash \Gamma_2 \qquad \Gamma_2 \vdash \tau_2 \ <: \ \tau_1 \vdash \Gamma_1}{\Gamma_1 \vdash L \ = \ \tau_1 \ <: \ L \ = \ \tau_2 \dashv \Gamma_2} \quad \text{(S}_\sigma\text{-\textsc{Equal})}$$

$$\frac{\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \vdash \Gamma_2 \qquad \Gamma_2 \vdash \tau_2 \ <: \ \tau_1 \vdash \Gamma_1}{\Gamma_1 \vdash L \ \preceq \ \tau_1 \ <: \ L \ \preceq \ \tau_2 \dashv \Gamma_2} \quad \text{(S}_\sigma\text{-\textsc{Nominal})}$$

$$\frac{\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \vdash \Gamma_2}{\Gamma_1 \vdash l : \tau_1 \ <: \ l : \tau_2 \dashv \Gamma_2} \quad \text{(S}_\sigma\text{-\textsc{Value})}$$

Figure 5.4: *Wyv*$_{\mathit{fix}}$ Subtyping

place within the same environment. A double headed subtype relation is not a novel form. Amin et al. [13], used a double headed form of subtyping to push back on narrowing when attempting to derive transitivity. While Amin et al. used a double headed form of subtyping in a specific instance, Wyv_{fix} entirely replaces subtyping with the double headed form.

Wyv_{fix} is syntactically identical to Wyv_{core}. The full subtype semantics are given in Figure 5.4. Apart from the addition of a second environment, subtyping of Wyv_{fix} does not differ from that of Wyv_{core}. Member lookup is performed in the environment related to the appropriate type (S-Upper and S-Lower). Environment updates are performed simultaneously on both environments (S-All and S-Rfn). Comparison of types in contra-variant positions invert the positions of the environments.

5.5.2 Subtype Decidability

The advantage of Wyv_{fix} over other type systems discussed in this thesis is the ease with which subtyping can be reduced to subtyping of Wyv_{expand}. Subtyping in Fixed Wyvern completely removes narrowing and thus the expansion of types in Wyv_{fix} to Wyv_{expand} remains constant during subtyping.

The proof of subtype decidability is achieved from a proof of equivalence between Wyv_{fix} subtyping and subtyping of the Wyv_{expand} types those types expand to. This is provided in Lemmas 5.5.1 and 5.5.2. Subtype decidability follows directly in Theorem 5.5.1.

Lemma 5.5.1 (Subtyping ⇒ `subtype`). *For all* $\mathrm{Wyv}_{\mathit{fix}}$ *types and declaration types,* τ_1*,* σ_1*,* τ_2 *and* σ_2 *in environments* Γ_1 *and* Γ_2*, and their expansions into* $\mathrm{Wyv}_{\mathsf{expand}}$ T_1 *and* T_2*, and* D_1 *and* D_2 *such that*

$$\Gamma_1 \vdash \tau_1 \ \longmapsto\ T_1$$

and

$$\Gamma_2 \vdash \tau_2 \ \longmapsto\ T_2$$

and

$$\Gamma_1 \vdash \sigma_1 \ \longmapsto \ D_1$$

and

$$\Gamma_2 \vdash \sigma_2 \ \longmapsto \ D_2$$

if

$$\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \dashv \Gamma_2 \ \textit{then}\ \texttt{subtype}(T_1, T_2) = \textit{true}$$

and if

$$\Gamma_1 \vdash \sigma_1 \ <: \ \sigma_2 \dashv \Gamma_2 \ \textit{then}\ \texttt{subtypeDecl}(D_1, D_2) = \textit{true}$$

Proof. Proof proceeds by mutual induction on the structure of the proofs of

$$\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \dashv \Gamma_2 \qquad \Gamma_1 \vdash \sigma_1 \ <: \ \sigma_2 \dashv \Gamma_2$$

- *Induction Hypothesis 1:* for any sub-proof in $\Gamma_1 \vdash \tau_1 \ <: \ \tau_2 \dashv \Gamma_2$, `subtype` (or `subtypeDecl`) returns `true`.

- *Induction Hypothesis 2:* for any sub-proof in $\Gamma_1 \vdash \sigma_1 \ <: \ \sigma_2 \dashv \Gamma_2$, `subtype` (or `subtypeDecl`) returns `true`.

Case 1 (Base Case: S-Top)**.**

$$\tau_2 = \top$$

It follows by inversion of the derivation of $\Gamma \vdash \tau_1 \ \longmapsto \ T_2$, that $T_2 = \top$. Thus, after case analysis on T_1, the desired result, $\texttt{subtype}(T_1, \top) = \texttt{true}$ is trivially derived.

Case 2 (Base Case: S-Bot)**.**

$$\tau_1 = \bot$$

It follows by inversion of the derivation of $\Gamma \vdash \tau_1 \ \longmapsto \ T_2$, that $T_1 = \bot$. Thus, the desired result, $\texttt{subtype}(\bot, T_2) = \texttt{true}$ is trivially derived.

Case 3 (Base Case: S-Rfl).

$$\tau_1 = \tau_2 = x.L$$

By inversion on the derivation of $\Gamma_1 \vdash \tau_1 \longmapsto T_1$ and $\Gamma_2 \vdash \tau_2 \longmapsto T_2$, it follows that there exists $\overline{D}_1$ and $\overline{D}_2$ such that

$$T_1 = x.L \mapsto \overline{D_1} \qquad T_2 = x.L \mapsto \overline{D_2}$$

By the definition of `subtype`, it follows trivially that $\texttt{subtype}(x.L \mapsto \overline{D_1}, x.L \mapsto \overline{D_2}) = \texttt{true}$.

Case 4 (S-Upper).

$$\tau_1 = x.L \qquad \Gamma_1 \vdash x \ : \ \{L \ \leqslant \tau'\} \qquad \Gamma_1 \vdash \tau \ <: \ \tau_2 \dashv \Gamma_2$$

By the derivation of $\Gamma \vdash \tau_1 \longmapsto T_1$, it follows that there exists $\overline{D}$ such that:

$$T_1 = x.L \mapsto \overline{D}$$

By the definition of type expansion to $Wyv_{\texttt{expand}}$, it follows that $\exists T_1' \in \texttt{ub}(x.L \mapsto \overline{D})$. By the inductive hypothesis, it follows that $\texttt{subtype}(T_1', T_2) = \texttt{true}$. Thus by case analysis on T_2, it is easy to demonstrate that $\texttt{subtype}(T_1, T_2) = \texttt{true}$, the desired result.

Case 5 (S-Lower).

$$\tau_2 = x.L \qquad \Gamma_2 \vdash x \ : \ \{L \ \geqslant \tau'\} \qquad \Gamma_1 \vdash \tau_1 \ <: \ \tau \dashv \Gamma_2$$

By the derivation of $\Gamma \vdash \tau_2 \longmapsto T_2$, it follows that there exists $\overline{D}_2$ and T_2' such that:

$$T_2 = x.L \mapsto \overline{D}_2 \qquad \Gamma_2 \vdash \tau \longmapsto T_2'$$

By the definition of type expansion, it follows that $T_2' \in \texttt{lb}(x.L \mapsto \overline{D})$. By the inductive hypothesis, it follows that $\texttt{subtype}(T_1, T_2') = \texttt{true}$. Thus by case analysis on T_1, it is easy to demonstrate that $\texttt{subtype}(T_1, T_2) = \texttt{true}$, the desired result.

Case 6 (S-All).

$$\tau_1 = \forall(x : \tau_1').\tau_1'' \qquad \tau_2 = \forall(x : \tau_2').\tau_2'' \qquad \Gamma_2 \vdash \tau_2' \ <: \ \tau_1' \dashv \Gamma_1$$
$$\Gamma_1, x : \tau_1' \vdash \tau_1'' \ <: \ \tau_2'' \dashv \Gamma_2, x : \tau_2''$$

By the definition of $\Gamma_1 \vdash \tau_1 \ \longmapsto \ T_1$, it follows that there exists T_1' and T_1'' such that

$$\Gamma_1 \vdash \tau_1' \ \longmapsto \ T_1' \qquad \Gamma_1, x : \tau_1' \vdash \tau_1'' \ \longmapsto \ T_1'' \qquad T_1 = \forall(x : T_1').T_1''$$

Similarly, there exists T_2' and T_2'' such that

$$\Gamma_2 \vdash \tau_2' \ \longmapsto \ T_2' \qquad \Gamma_2, x : \tau_2' \vdash \tau_2'' \ \longmapsto \ T_2'' \qquad T_2 = \forall(x : T_2').T_2''$$

By the induction hypothesis, it follows that $\texttt{subtype}(T_2', T_1') = \texttt{true}$ and $\texttt{subtype}(T_1'', T_2'') = \texttt{true}$. Thus it is easy to demonstrate that $\texttt{subtype}(T_1, T_2) = \texttt{true}$, the desired result.

Case 7 (S-Ext).

$$\Gamma_1 \vdash \tau_1 \ \leqslant:: \ \tau \qquad \Gamma_1 \vdash \tau \ <: \ \tau_2 \dashv \Gamma_2$$

By the definition of type extension, either τ_1 is some $x.L$, where $\Gamma_1 \vdash x : \{L \ \leqslant \tau\}$, in which case we can use the same reasoning to Case 4 to derive the desired result, or τ_1 is some $\tau'\{z \Rightarrow \overline{\sigma}\}$. In the second case, by inversion on the derivation of $\Gamma_1 \vdash \tau_1 \ \longmapsto \ T_1$ there exists some T', $\overline{D}$, T and $\overline{T}$ such that

$$\Gamma_1 \vdash \tau' \ \longmapsto \ T'$$
$$\Gamma_1 \vdash \overline{\sigma} \ \longmapsto \ \overline{D} \qquad \Gamma_1 \vdash \tau \ \longmapsto \ T \qquad T_1 = T'\{z \Rightarrow \overline{D}\} \mapsto \overline{T} \qquad T' \in \overline{T}$$

By the induction hypothesis, it follows that $\texttt{subtype}(T, T_2') = \texttt{true}$. Thus it is easy to demonstrate that $\texttt{subtype}(T_1, T_2) = \texttt{true}$, the desired result.

Case 8 (S-Rfn).

$$\tau_1 = \tau\{z \Rightarrow \overline{\sigma}_1\} \qquad \tau_2 = \tau\{z \Rightarrow \overline{\sigma}_2\} \qquad \Gamma_1, z : \tau_1 \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2 \dashv \Gamma_2, z : \tau_2$$

By the derivation of $\Gamma_1 \vdash \tau_1 \;\mapsto\; T_1$ there exists T_1' and $\overline{D}_1$ such that

$$\Gamma_1 \vdash \tau \;\mapsto\; T_1' \qquad \Gamma_1, z : \tau_1 \vdash \overline{\sigma}_1 \;\mapsto\; \overline{D}_1$$

Similarly, there exists T_2' and $\overline{D}_2$ such that

$$\Gamma_2 \vdash \tau \;\mapsto\; T_2' \qquad \Gamma_2, z : \tau_2 \vdash \overline{\sigma}_2 \;\mapsto\; \overline{D}_2$$

By the induction hypothesis, it follows that $\texttt{subtypeDecl}(\overline{D}_1, \overline{D}_2) = \texttt{true}$, and thus giving the desired result: $\texttt{subtype}(T_1, T_2) = \texttt{true}$.

The remaining cases related to declaration subtyping are all easily derived by an application of the induction hypothesis to the premises. □

Lemma 5.5.2 (Subtyping $\Leftarrow$ $\texttt{subtype}$)**.** *For all τ_1 and τ_2 in environments Γ_1 and Γ_2 respectively, $\Gamma_1 \vdash \tau_1 \;<:\; \tau_2 \dashv \Gamma_2$ is derivable if* $\texttt{subtype}(T_1, T_2) = \texttt{true}$*, where T_1 and T_2 are the* Wyv$_{\texttt{expand}}$ *expansions of τ_1 and τ_2 such that $\Gamma_1 \vdash \tau_1 \;\longmapsto\; T_1$ and $\Gamma_2 \vdash \tau_2 \;\longmapsto\; T_2$.*

Proof. Since the evaluation of $\texttt{subtype}(T_1, T_2)$ is finite, we induct on its evaluation depth. The evaluation depth of $\texttt{subtype}(T_1, T_2)$ is defined as the number of recursive calls made during it's evaluation. Then by case analysis on those combinations of T_1, T_2 where either $\texttt{subtype}(T_1, T_2) = \texttt{true}$, or results in a recursive call to $\texttt{subtype}$.

Case 1 ($T_1 = \top$, $T_2 = \top$)**.** Trivial.

Case 2 ($T_1 = \top, T_2 = x.L \mapsto \overline{D}$)**.** There is one sub-evaluation $\texttt{subtype}(T_1, \texttt{lb}(x.L))$ which is strictly smaller in terms of evaluation depth than $\texttt{subtype}(T_1, T_2)$. By inversion on the derivation of $\Gamma \vdash \tau_2 \;\longmapsto\; x.L \mapsto \overline{D}$ there exists τ and T such that

$$\Gamma_2 \vdash \tau \;\longmapsto\; T \qquad T \in \texttt{lb}(x.L \mapsto \overline{D}) \qquad \Gamma_2 \vdash x \;:\; \{L \;\geqslant \tau\}$$
$$\texttt{subtype}(T_1, T) = \texttt{true}$$

That is, τ expands to T, and T is a lower bound of $x.L$ in Γ_2, and super types T_1. By the inductive hypothesis, it follows that $\Gamma_1 \vdash \tau_1 \;<:\; \tau \vdash \Gamma_2$. Thus, by S-Lower the desired result is achieved.

Case 3 **($T_1 = \bot$).** Trivial.

Case 4 **($T_1 = x.L$, $T_2 = \top$).** Trivial.

Case 5 **($T_1 = x_1.L_1 \mapsto \overline{D}_1$, $T_2 = x_2.L_2 \mapsto \overline{D}_2$).** $\texttt{subtype}(T_1, T_2)$ reduces to $(x_1 = x_2 \wedge L_1 = L_2) \vee (\texttt{subtype}(\texttt{up}(x_1.L_1 \mapsto \overline{D}_1), T_2)) \vee (\texttt{subtype}(T_1, \texttt{lb}(x_2.L_2 \mapsto \overline{D}_2)))$. Thus, one of these boolean subterms must equal `true`. By case analysis:

Subcase 1 **($x_1 = x_2 \wedge L_1 = L_2$).** By trivial application of S-Rfl.

Subcase 2 **($\texttt{subtype}(\texttt{ub}(x_1.L_1 \mapsto \overline{D}_1), T_2) = \texttt{true}$).** By the equivalence between `upper_bound` and upper bound lookup, and similar reasoning to that of Case 2 the desired result is achieved.

Subcase 3 **($\texttt{subtype}(T_1, \texttt{lb}(x_2.L_2 \mapsto \overline{D}_2)) = \texttt{true}$).** By the same reasoning as the Case 2 the desired result is achieved.

Case 6 **($T_1 = x.L \mapsto \overline{D}$, $T_2 = \_$).** By the equivalence between `ub` and upper bound lookup, and similar reasoning to that of Case 2 the desired result is achieved.

Case 7 **($T_1 = \forall(\_ : \_).\_$, $T_2 = \top$).** Trivial.

Case 8 **($T_1 = \forall(x : S).U$, $T_2 = x.L \mapsto \overline{D}$).** By the same reasoning as that of Case 2 the desired result is achieved.

Case 9 **($T_1 = \forall(x : S_1).U_1$, $T_2 = \forall(x : S_2).U_2$).**

$$\texttt{subtype}(T_1, T_2) = \texttt{subtype}(S_2, S_1) \wedge \texttt{subtype}(U_1, U_2) = \texttt{true}$$

Both of the above evaluations of `subtype` have strictly smaller evaluation depths than $\texttt{subtype}(T_1, T_2)$. By the inversion of the derivation of $\Gamma_1 \vdash \tau_1 \longmapsto \forall(x : S_1).U_1$ and $\Gamma_2 \vdash \tau_2 \longmapsto \forall(x : S_2).U_2$, there exists τ_1', τ_1'', τ_2' and τ_2'' such that

$$\begin{array}{lll} \tau_1 = \forall(x : \tau_1').\tau_1'' & \Gamma_1 \vdash \tau_1' \longmapsto S_1 & \Gamma_1, x : \tau_1' \vdash \tau_1'' \longmapsto U_1 \\ \tau_2 = \forall(x : \tau_2').\tau_2'' & \Gamma_2 \vdash \tau_2' \longmapsto S_2 & \Gamma_2, x : \tau_2' \dashv \tau_2'' \longmapsto U_2 \end{array}$$

Thus by the inductive hypothesis, it follows that

$$\Gamma_2 \vdash \tau_2' \ <: \ \tau_1' \vdash \Gamma_1 \qquad \Gamma_1, x : \tau_1' \vdash \tau_1'' \ <: \ \tau_2'' \vdash \Gamma_2, x : \tau_2'$$

And subsequently by S-All that $\Gamma_1 \vdash \forall(x : \tau_1').\tau_1'' \ <: \ \forall(x : \tau_2').\tau_2' \dashv \Gamma_2$, the desired result.

Case 10 ($T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}$, $T_2 = \top$)**.** Trivial.

Case 11 ($T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}$, $T_2 = x_2.L_2 \mapsto \overline{D}_2$)**.** By the same reasoning as that of Case 2 the desired result is achieved.

Case 12 ($T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$, $T_2 = x_2.L_2\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$)**.** By case analysis on the equality of $x_1.L_1$ and $x_2.L_2$:

Subcase 1 ($x_1.L_1 = x_2.L_2 = x.L$)**.**

$$\texttt{subtype}(T_1, T_2) = \texttt{subtypeDecl}(\overline{D}_1, \overline{D}_2) = \texttt{true}$$

By inversion on the derivation of $\Gamma_1 \vdash \tau_1 \ \longmapsto \ x.L\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$ and $\Gamma_2 \vdash \tau_2 \ \longmapsto \ x.L\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$, there exists $\overline{\sigma}_1$ and $\overline{\sigma}_2$ such that

$$\begin{array}{ll} \tau_1 = x.L\{z \Rightarrow \overline{\sigma}_1\} & \Gamma_1, z : x.L\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \ \longmapsto \ \overline{D}_1 \\ \tau_2 = x.L\{z \Rightarrow \overline{\sigma}_2\} & \Gamma_2, z : x.L\{z \Rightarrow \overline{\sigma}_2\} \vdash \overline{\sigma}_2 \ \longmapsto \ \overline{D}_2 \end{array}$$

For all (D_1, D_2) in $(\overline{D}_1, \overline{D}_2)$, the evaluation depth of $\texttt{subtypeDecl}(D_1, D_2)$ is strictly smaller that that of $\texttt{subtype}(T_1, T_2)$. Thus by the induction hypothesis, it follows that

$$\Gamma_1, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2 \dashv \Gamma_2, z : \tau\{z \Rightarrow \overline{\sigma}_2\}$$

Subsequently by S-Refine the desired result $\Gamma_1 \vdash \tau\{z \Rightarrow \overline{\sigma}_1\} \ <: \ \tau\{z \Rightarrow \overline{\sigma}_2 \dashv \Gamma_2\}$ is achieved.

Subcase 2 ($x_1.L_1 \neq x_2.L_2$)**.**

$$\exists T, T \in \overline{T}_1 \qquad \texttt{subtype}(T_1, T_2) = \texttt{subtype}(T, T_2) = \texttt{true}$$

By inversion on the derivation of $\Gamma_1 \vdash \tau_1 \longmapsto x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$ and $\Gamma_2 \vdash \tau_2 \longmapsto x_2.L_2\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$, there exists τ, $\overline{\sigma}_1$ and $\overline{\sigma}_2$ such that

$$\tau_1 = x_1.L_1\{z \Rightarrow \overline{\sigma}_1\} \qquad \Gamma_1, z : \tau_1'\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \mapsto \overline{D}_1$$
$$\tau_2 = x_2.L_2\{z \Rightarrow \overline{\sigma}_2\} \qquad \Gamma_2, z : \tau_2'\{z \Rightarrow \overline{\sigma}_2\} \vdash \overline{\sigma}_2 \mapsto \overline{D}_2 \qquad \Gamma \vdash \tau \longmapsto T$$
$$\Gamma \vdash x.L\{z \Rightarrow \overline{\sigma}_1\} \leqslant:: \tau$$

Thus by the induction hypothesis, it follows that $\Gamma_1 \vdash \tau <: \tau_2$, and subsequently by S-Extend the desired result that $\Gamma_1 \vdash x_1.L_1\{z \Rightarrow \overline{\sigma}_1\} <: \tau_2 \dashv \Gamma_2$.

The remaining cases relating to the subtyping of *Wyv*$_{\texttt{expand}}$ declaration types are omitted as they are easily derivable by simple application of the induction hypothesis. □

Theorem 5.5.1 (Subtype Decidability)**.** *Subtyping in* Wyv$_{\mathit{fix}}$ *is decidable.*

Proof. From Lemmas 5.5.1 and 5.5.2 it follows that for all τ_1, T_1, Γ_1, τ_2, T_2 and Γ_2 where $\Gamma_1 \vdash \tau_1 \longmapsto T_1$ and $\Gamma_2 \vdash \tau_2 \longmapsto T_2$, $\Gamma_1 \vdash \tau_1 <: \tau_2 \vdash \Gamma_2 \iff \texttt{subtype}(T_1, T_2) = \texttt{true}$

Thus, by Theorem 5.4.4, there exists a terminating algorithm able to decide questions of Fixed Wyvern subtyping. □

5.5.3 Transitivity in *Wyv*$_{\mathit{fix}}$

Subtyping in *Wyv*$_{\mathit{fix}}$ is not transitive. This is not entirely surprising since environment narrowing and transitivity have a long history of being intertwined [13, 75, 10, 73] and tightly interdependent. The absence of transitivity is easiest to understand in the following counter-example.

```
type A = {w⇒type L = Integer
             type L' >: Integer}
type B = {w⇒type L = Integer
             type L' >: w.L}
type C = {w⇒type L <: ⊤
             type L' >: w.L}
```

While $\mathsf{A} <: \mathsf{B}$ and $\mathsf{B} <: \mathsf{C}$, $\mathsf{A} \not<: \mathsf{C}$. 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. 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.

Chapter 6

Non-Recursive Subtyping

Recursive types and their subtyping are at the core of the decidability issues in Wyv_{core}. Using recursive types, programmers are able to construct mutually defined types that feature cycles, recursive type definitions, and encode the subtyping of two type languages that are known to contain undecidable subtyping, System $F_{<:}$ and Java. In this Chapter I present a variant of Wyv_{core} that removes subtyping of recursive types: $Wyv_{non\text{-}\mu}$. This may seem like a significant restriction, however it is in some ways the simplest of available restrictions, and resembles the subtyping of Wadlerfest DOT [14], that also does not include subtyping for recursive types. Further, Scala itself does not include subtyping of recursive structural types [73]. Scala allows for recursively defined types, and recursive inheritance, but not subtyping of recursive type refinements. Both of these languages suggest that valuable expressiveness can still be derived from such a restriction.

6.1 $Wyv_{non\text{-}\mu}$

In this section, I define the subtyping of $Wyv_{non\text{-}\mu}$, a restricted form of the subtyping of Figure 3.14. While the subtyping of $Wyv_{non\text{-}\mu}$ differs from that of Wyv_{core}, the syntax and related relations remain the same, and so are omitted.

$$\Gamma \vdash x.L \;<:\; x.L \quad (\textsc{S-Rfl}) \qquad\qquad \Gamma \vdash \bot <: \tau \quad (\textsc{S-Bot})$$

$$\Gamma \vdash \tau <: \top \quad (\textsc{S-Top}) \qquad \frac{\Gamma \vdash x \;:\; \{L \;\leqslant \tau'\} \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash x.L \;<:\; \tau} \quad (\textsc{S-Upper})$$

$$\frac{\Gamma \vdash x \;:\; \{L \;\geqslant \tau'\} \qquad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash \tau \;<:\; x.L} \quad (\textsc{S-Lower})$$

$$\frac{\Gamma, x : \tau \vdash \tau_1 \;<:\; \tau_2}{\Gamma \vdash \forall(x : \tau).\tau_1 \;<:\; \forall(x : \tau).\tau_2} \quad (\textsc{S-All}) \qquad \frac{\Gamma \vdash \overline{\sigma}_1 <: \overline{\sigma}_2}{\Gamma \vdash \tau\{\overline{\sigma}_1\} <: \tau\{\overline{\sigma}_2\}} \quad (\textsc{S-Rfn})$$

$$\frac{\Gamma \vdash \tau_1 \;\leqslant::\; \tau \qquad \Gamma \vdash \tau \;<:\; \tau_2}{\Gamma \vdash \tau_1 \;<:\; \tau_2} \quad (\textsc{S-Ext})$$

Figure 6.1: $Wyv_{non\text{-}\mu}$ Subtyping

The subtype rules for $Wyv_{non\text{-}\mu}$ are shown in Figure 6.1. I do not include the subtype rules for declaration types, as they are identical to those in Figure 3.14. Subtyping in $Wyv_{non\text{-}\mu}$ differs from Wyv_{core} only in the subtyping of type refinements (S-Rfn), requiring the refinement to be non-recursive, and dependent function types, requiring invariance on the argument type.

6.2 Subtype Decidability

In this section I demonstrate that the subtyping of Figure 6.1 is decidable. Types in $Wyv_{non\text{-}\mu}$ are unchanged from those of Wyv_{core}, and as such the encoding to Wyv_{expand} (see Section 5.1) remains the same. $Wyv_{non\text{-}\mu}$ subtyping semantically differs from that of Figure 3.14, by only allowing subtyping of non-recursive type refinements (S-Rfn). As a consequence, Algorithm

```
$\texttt{subtype}_{non-\mu}(T_1, T_2)$
    switch $T_1$ do
        case $\top$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x.L \mapsto \overline{D}$ do $\bigvee_{T \in \texttt{lb}(x.L \mapsto \overline{D})} \texttt{subtype}_{non-\mu}(T_1, T)$ ;
                otherwise do false;
            end
        case $\bot$ do true;
        case $x_1.L_1 \mapsto \overline{D}_1$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x_2.L_2 \mapsto \overline{D}_2$ do $(x_1 = x_2 \wedge L_1 = L_2) \vee (\bigvee_{T \in \texttt{ub}(x_1.L_1 \mapsto \overline{D}_1)}$
                  $\texttt{subtype}_{non-\mu}(T, T_2)) \vee (\bigvee_{T \in \texttt{lb}(x_2.L_2 \mapsto \overline{D}_2)} \texttt{subtype}_{non-\mu}(T_1, T))$ ;
                otherwise do $\bigvee_{T \in \texttt{ub}(x_1.L_1 \mapsto \overline{D}_1)} \texttt{subtype}_{non-\mu}(T, T_2)$ ;
            end
        case $\forall(x : S_1).U_1$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x_2.L_2 \mapsto \overline{D}_2$ do $\bigvee_{T \in \texttt{lb}(x_2.L_2 \mapsto \overline{D}_2)} \texttt{subtype}_{non-\mu}(T_1, T)$ ;
                case $\forall(x : S_2).U_2$ do
                    $S_1 = S_2 \wedge \texttt{subtype}_{non-\mu}(U_1, U_2)$
                otherwise do false;
            end
        case $x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$ do
            switch $T_2$ do
                case $\top$ do true;
                case $x_2.L_2 \mapsto \overline{D}_2$ do $\bigvee_{T \in \texttt{lb}(x_2.L_2 \mapsto \overline{D}_2)} \texttt{subtype}_{non-\mu}(T_1, T)$ ;
                case $x_2.L_2\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$ do
                    if $x_1.L_1 == x_2.L_2 \wedge z \notin fv(\overline{D}_1) \cup fv(\overline{D}_2)$ then
                        $\texttt{subtypeDecl}_{non-\mu}(\overline{D}_1, \overline{D}_2)$
                    else if $x_1.L_1 == x_2.L_2 \wedge z \in fv(\overline{D}_1) \cup fv(\overline{D}_2)$ then
                        false
                    else
                        $\bigvee_{T \in \overline{T}_1} \texttt{subtype}_{non-\mu}(T, T_2)$
                otherwise do false;
            end
    end
```

Algorithm 4: $Wyv_{non\text{-}\mu}$ Subtyping Algorithm

2 does not capture the subtyping of Figure 6.1, and a modified subtype algorithm must be defined in order to subtype the type graphs of $Wyv_{non\text{-}\mu}$ types. Algorithm 4 defines the subtype algorithm for $Wyv_{non\text{-}\mu}$ expansions into $Wyv_{\texttt{expand}}$.

Algorithm 4 defines $\texttt{subtype}_{non-\mu}$, a subtype algorithm that is identical to `subtype` defined in Algorithm 2 except during subtyping of dependent function type and when checking subtyping between refinements on equivalent types. Argument types are invariant during subtyping of dependent function types and refinements are only compared in the case that they do not have recursive references. This is a minor deviation from Algorithm 2, that allows us to still lean on the decidability result of 5.4. Theorem 6.2.1 proves that any call to $\texttt{subtype}_{non-\mu}$ terminates if an equivalent call to `subtype` terminates. Given Theorem 5.4.4, this is equivalent to demonstrating that $\texttt{subtype}_{non-\mu}$ for all inputs.

Theorem 6.2.1 ($\texttt{subtype}_{non-\mu}$ Termination)**.** *For all T_1 and T_2, if* $\texttt{subtype}(T_1, T_2)$ *terminates, then so does* $\texttt{subtype}_{non-\mu}(T_1, T_2)$.

Proof. Since we know that $\texttt{subtype}(T_1, T_2)$ will terminate, we can induct on the depth of the call tree for $\texttt{subtype}(T_1, T_2)$. Our induction hypothesis being that the termination of any sub-call $\texttt{subtype}(T_1', T_2')$ implies the termination of $\texttt{subtype}_{non-\mu}(T_1, T_2)$. All cases are trivially demonstrated by the induction hypothesis, leaving only the case:

$$T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1 \qquad T_2 = x_2.L_2\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$$

where

```
$\texttt{subtype}(T_1, T_2)$ = if $x_1.L_1 == x_2.L_2$ then
    $\texttt{subtypeDecl}(\overline{D}_1, \overline{D}_2)$
else
    $\bigvee_{T \in \overline{T}_1} \texttt{subtype}(T, T_2)$
```

and

```
$\texttt{subtype}_{non-\mu}(T_1, T_2) =$ if $x_1.L_1 == x_2.L_2 \wedge z \notin fv(\overline{D}_1) \cup fv(\overline{D}_2)$
 then
    $\texttt{subtypeDecl}_{non-\mu}(\overline{D}_1, \overline{D}_2)$
else if $x_1.L_1 == x_2.L_2 \wedge z \in fv(\overline{D}_1) \cup fv(\overline{D}_2)$ then
    false
else
    $\bigvee_{T \in \overline{T}_1} \texttt{subtype}_{non-\mu}(T,\ T_2)$
```

The desired result can then be achieved by case analysis: first on the equality of the base types ($x_1.L_1 == x_2.L_2$), giving the desired result if the base types differ, and subsequently on whether z is in the set of free variables of $\overline{D}_1$ and $\overline{D}_2$.

Subcase 1 ($z \notin fv(\overline{D}_1) \cup fv(\overline{D}_2)$)**.** By the induction hypothesis.

Subcase 2 ($z \in fv(\overline{D}_1) \cup fv(\overline{D}_2)$)**.** Trivial.

□

Finally, equivalence between Algorithm 4 and the declarative subtyping rules is shown in Theorems 6.2.2 and 6.2.3.

Theorem 6.2.2 ($<:_{non-\mu} \Rightarrow \texttt{subtype}_{non-\mu}$)**.** *For all* Γ, τ_1, τ_2, T_1 *and* T_2 *where* $\Gamma \vdash \tau_1 \longmapsto T_1$ *and* $\Gamma \vdash \tau_2 \longmapsto T_2$*, if* $\Gamma \vdash \tau_1 <: \tau_2$ *then it follows that* $\texttt{subtype}_{non-\mu}(T_1, T_2) = \texttt{true}$.

Proof. By induction on the derivation of $\Gamma \vdash \tau_1 <: \tau_2$.

Case 1 (S-Rfl)**.**

$$\tau_1 = x.L \qquad \tau_2 = x.L$$

By inversion on the derivation of $\Gamma \vdash x.L \longmapsto T_2$, it follows that there exists some T such that

$$T_1 = x.L \mapsto T = T_2$$

By the definition of $\texttt{subtype}_{non-\mu}$, we get the desired result:

$$\texttt{subtype}_{non-\mu}(T_1, T_2) = (x == x) \wedge (L == L) = \texttt{true}$$

Case 2 (S-Top)**.**

$$\tau_2 = \top$$

By inversion of the derivation of $\Gamma \vdash \tau_2 \;\longmapsto\; T_2$, it follows that

$$T_2 = \top$$

And subsequently by the definition of $\texttt{subtype}_{non-\mu}$ we easily get the desired result.

Case 3 (S-Bot)**.**

$$\tau_1 = \bot$$

By inversion of the derivation of $\Gamma \vdash \tau_1 \;\longmapsto\; T_1$, it follows that

$$T_1 = \bot$$

And subsequently by the definition of $\texttt{subtype}_{non-\mu}$ we easily get the desired result.

Case 4 (S-Upper)**.**

$$\tau_1 = x.L \qquad \Gamma \vdash x \;:\; \{L \;\leqslant \tau\} \qquad \Gamma \vdash \tau \;<: \tau_2$$

By inversionon the derivation of $\Gamma \vdash x.L \;\longmapsto\; T_1$, it follows that there exists $\overline{D}$ and T such that

$$T_1 = x.L \mapsto \overline{D} \qquad L \;\leqslant T \in \overline{D} \qquad \Gamma \vdash \tau \;\longmapsto\; T$$

By the induction hypothesis it follows that

$$\texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$$

and thus by the definition of $\texttt{subtype}_{non-\mu}$ we get the desired result:

$$\texttt{subtype}_{non-\mu}(T_1, T_2) = \texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$$

Case 5 (S-Lower)**.** By similar reasoning to the case for S-Upper.

Case 6 (S-All)**.**

$$\tau_1 = \forall(x : \tau).\tau_1' \qquad \tau_2 = \forall(x : \tau).\tau_2' \qquad \Gamma, x : \tau \vdash \tau_1' \ <: \ \tau_2'$$

By inversion on the definition of $\Gamma \vdash \tau_1 \ \longmapsto \ T_1$ and $\Gamma \vdash \tau_2 \ \longmapsto \ T_2$, there exists some T, T_1', and T_2' such that

$$\begin{gathered} T_1 = \forall(x : T).T_1' \qquad T_2 = \forall(x : T).T_2' \\ \Gamma \vdash \tau \ \longmapsto \ T \qquad \Gamma, x : \tau \vdash \tau_1' \ \longmapsto \ T_1' \qquad \Gamma x : \tau \vdash \tau_2' \ \longmapsto \ T_2' \end{gathered}$$

By the definition of $\mathtt{subtype}_{non-\mu}$ we have,

$$\mathtt{subtype}_{non-\mu}(T_1, T_2) = (T == T) \wedge (\mathtt{subtype}_{non-\mu}(T_1', T_2'))$$

By the induction hypothesis we have $\mathtt{subtype}_{non-\mu}(T_1', T_2') = \mathtt{true}$, the desired result.

Case 7 (S-Rfn)**.**

$$\tau_1 = x.L\{\overline{\sigma}_1\} \qquad \tau_2 = x.L\{\overline{\sigma}_2\} \qquad \Gamma \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2$$

By inversion on the derivation of $\Gamma \vdash \tau_1 \ \longmapsto \ T_1$ and $\Gamma \vdash \tau_2 \ \longmapsto \ T_2$, for all $\sigma_1 \ \in \ \overline{\sigma}_1$ and $\sigma_1 \ \in \ \overline{\sigma}_2$, such that $\Gamma \vdash \sigma_1 \ <: \ \sigma_2$, there exists $\overline{D}_1$, $\overline{D}_2$, D_1, and D_2 such that

$$\begin{gathered} \Gamma, z : x.L\{\overline{\sigma}_1\} \vdash \sigma_1 \ \longmapsto \ D_1 \qquad \Gamma, z : x.L\{\overline{\sigma}_2\} \vdash \sigma_2 \ \longmapsto \ D_2 \\ \Gamma, z : x.L\{\overline{\sigma}_1\} \vdash \overline{\sigma}_1 \ \longmapsto \ \overline{D}_1 \qquad \Gamma, z : x.L\{\overline{\sigma}_2\} \vdash \overline{\sigma}_2 \ \longmapsto \ \overline{D}_2 \end{gathered}$$

By environment strengthening (as $z \ \notin \ fv(\overline{\sigma}_1)$ and $z \ \notin \ fv(\overline{\sigma}_2)$) we get

$$\begin{gathered} \Gamma \vdash \sigma_1 \ \longmapsto \ D_1 \\ \Gamma \vdash \sigma_2 \ \longmapsto \ D_2 \qquad \Gamma \vdash \overline{\sigma}_1 \ \longmapsto \ \overline{D}_1 \qquad \Gamma \vdash \overline{\sigma}_2 \ \longmapsto \ \overline{D}_2 \end{gathered}$$

By the definition of $\mathtt{subtype}_{non-\mu}$ we have,

$$\mathtt{subtype}_{non-\mu}(T_1, T_2) = (x.L == x.L) \wedge (\mathtt{subtypeDecl}_{non-\mu}(\overline{D}_1, \overline{D}_2))$$

By the induction hypothesis, we get

$$\texttt{subtype}_{non-\mu}(\overline{D}_1, \overline{D}_2) = \texttt{true}$$

giving us the desired result

$$\texttt{subtype}_{non-\mu}(T_1, T_1)$$

Case 8 (S-Ext)**.**

$$\Gamma \vdash \tau_1 \;\leqslant::\; \tau \qquad \Gamma \vdash \tau \;<:\; \tau_2$$

By case analysis on the derivation of $\Gamma \vdash \tau_1 \;\leqslant::\; \tau'$:

Subcase 1 ($\tau_1 = x.L$)**.**

$$\Gamma \vdash x \;:\; \{L \;\leqslant \tau\}$$

By similar reasoning to Case 4 (S-Upper).

Subcase 2 ($\tau_1 = x.L\{z \Rightarrow \overline{\sigma}_1\}$)**.**

$$\Gamma \vdash x.L \;\leqslant::\; \tau'' \qquad \tau = flat(\tau'', z, \overline{\sigma}_1)$$

By inversion on the derivation of $\Gamma \vdash \tau_1 \;\longmapsto\; T_1$, there exists some $\overline{D}$, $\overline{T}$, and T such that

$$\Gamma \vdash x.L\{z \Rightarrow \overline{\sigma}_1\} \;\longmapsto\; x.L\{z \Rightarrow \overline{D}\} \mapsto \overline{T} \qquad \Gamma \vdash \tau \;\longmapsto\; T \qquad T \;\in\; \overline{T}$$

By the definition of $\texttt{subtype}_{non-\mu}$ we have

$$\texttt{subtype}_{non-\mu}(T_1, T_2) = \bigvee_{T \;\in\; \overline{T}} \texttt{subtype}_{non-\mu}(T, T_2)$$

By the induction hypothesis, we get

$$\texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$$

and thus by the definition of $\texttt{subtype}_{non-\mu}$ we have the desired result

$$\texttt{subtype}_{non-\mu}(T_1, T_2) = \bigvee_{T' \;\in\; \overline{T}} \texttt{subtype}_{non-\mu}(T', T_2) = \texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$$

□

Theorem 6.2.3 ($<:_{non-\mu} \Leftarrow$ $\texttt{subtype}_{non-\mu}$)**.** *For all* Γ*,* τ_1*,* τ_2*,* T_1 *and* T_2 *where* $\Gamma \vdash \tau_1 \longmapsto T_1$ *and* $\Gamma \vdash \tau_2 \longmapsto T_2$*, if* $\texttt{subtype}_{non-\mu}(T_1, T_2) = \texttt{true}$ *then it follows that* $\Gamma \vdash \tau_1 <: \tau_2$*.*

Proof. We induct on the finite call depth of $\texttt{subtype}_{non-\mu}$:

Case 1 ($T_1 = \top$, $T_2 = \top$)**.** By inversion on the derivation of $\Gamma \vdash \tau_2 \longmapsto T_2$ we have $\tau_2 = \top$, and subsequently the desired result by S-Top.

Case 2 ($T_1 = \top$, $T_2 = x_2.L_2 \mapsto \overline{D}$)**.**

$$\texttt{subtype}_{non-\mu}(T_1, T_2) = \bigvee_{T \in \texttt{lb}(x_2.L_2 \mapsto \overline{D})} \texttt{subtype}_{non-\mu}(\top, T) = \texttt{true}$$

In other words, there exists some $T \in \texttt{lb}(x_2.L_2 \mapsto \overline{D})$ such that $L \geqslant T \in \overline{D}$ and $\texttt{subtype}_{non-\mu}(T_1, T) = \texttt{true}$. By inversion on the derivation of $\Gamma \vdash \tau_2 \longmapsto x_2.L_2 \mapsto \overline{D}$ there exists some τ such that

$$\Gamma \vdash \tau \longmapsto T \qquad \Gamma \vdash x : \{L \geqslant \tau\}$$

Thus, by the induction hypothesis we have

$$\Gamma \vdash \tau_1 <: \tau$$

giving us the desired result by S-Lower.

Case 3 ($T_1 = \bot$)**.** By inversion on the derivation of $\Gamma \vdash \tau_1 \longmapsto \bot$ we have $\tau_1 = \bot$, and subsequently the desired result by S-Bot.

Case 4 ($T_1 = x_1.L_1$, $T_2 = \top$)**.** By inversion on the derivation of $\Gamma \vdash \tau_2 \longmapsto T_2$ we have $\tau_2 = \top$, and subsequently the desired result by S-Top.

Case 5 ($T_1 = x_1.L_1 \mapsto \overline{D}_1$, $T_2 = x_2.L_2 \mapsto \overline{D}_2$)**.**

$$x_1 = x_2 \qquad L_1 = L_2$$

Result is easily obtained by S-Rfl.

Case 6 ($T_1 = x_1.L_1 \mapsto \overline{D}_1$, $T_2 = x_2.L_2 \mapsto \overline{D}_2$)**.**

$$\texttt{subtype}_{non-\mu}(T_1, T_2) = \bigvee_{T \in \texttt{ub}(x_1.L_1 \mapsto \overline{D}_1)} \texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$$

In other words, there exists some $T \in \texttt{ub}(x_1.L_1 \mapsto \overline{D}_1)$ such that $\texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$. By inversion on the derivation of $\Gamma \vdash x_1.L_1 \mapsto \overline{D}_1 \ \longmapsto T_1$ there exists some τ such that

$$\Gamma \vdash \tau \ \mapsto \ T \qquad \Gamma \vdash x_1 \ : \ \{L \ \leqslant \tau\}$$

Thus, by the induction hypothesis we have

$$\Gamma \vdash \tau \ <: \ \tau_2$$

giving us the desired result by S-Upper.

Case 7 ($T_1 = x_1.L_1 \mapsto \overline{D}_1$, $T_2 = x_2.L_2 \mapsto \overline{D}_2$)**.** By similar reasoning to Case 2

Case 8 ($T_1 = x_1.L_1 \mapsto \overline{D}_1$)**.** By similar reasoning to Case 6

Case 9 ($T_1 = \forall(x : S_1).U_1$, $T_2 = \top$)**.** By inversion on the derivation of $\Gamma \vdash \tau_2 \ \longmapsto \ T_2$ we have $\tau_2 = \top$, and subsequently the desired result by S-Top.

Case 10 ($T_1 = \forall(x : S_1).U_1$, $T_2 = x_2.L_2 \mapsto \overline{D}_2$)**.** By similar reasoning to Case 2

Case 11 ($T_1 = \forall(x : S_1).U_1$, $T_2 = \forall(x : S_1).U_1$)**.**

$$\texttt{subtype}_{non-\mu}(T_1, T_2) = (S_1 == S_2) \wedge \texttt{subtype}_{non-\mu}(U_1, U_2) = \texttt{true}$$

By inversion on the derivation of $\Gamma \vdash \tau_1 \ \longmapsto \ \forall(x : S_1).U_1$ and $\Gamma \vdash \tau_2 \ \longmapsto \ \forall(x : S_2).U_2$ there exists some τ_1' and τ_1'' such that

$$\tau_1 = \forall(x : \tau_1').\tau_1'' \qquad \Gamma \vdash \tau_1' \ \longmapsto \ S_1 \qquad \Gamma, x : \tau_1' \vdash \tau_1'' \ \longmapsto \ U_1$$

and, some τ_2' and τ_2'' such that

$$\tau_2 = \forall(x : \tau_2').\tau_2'' \qquad \Gamma \vdash \tau_2' \longmapsto S_2 \qquad \Gamma, x : \tau_2' \vdash \tau_2'' \longmapsto U_2$$

It follows that $S_1 = S_2$, and subsequently that $\tau_1' = \tau_2'$. Thus we get $\Gamma, x : \tau_1' \vdash \tau_2'' \longmapsto U_2$. Subsequently by the induction hypothesis we get

$$\Gamma, x : \tau_1' \vdash \tau_1'' \ <: \ \tau_2''$$

and the desired result by S-All

Case 12 ($T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}$, $T_2 = \top$)**.** By inversion on the derivation of $\Gamma \vdash \tau_2 \longmapsto T_2$ we have $\tau_2 = \top$, and subsequently the desired result by S-Top.

Case 13 ($T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}$, $T_2 = x_2.L_2$)**.** By similar reasoning to Case 2

Case 14 ($T_1 = x.L\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$, $T_2 = x.L\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$)**.**

$$z \notin fv(\overline{D}_1) \cup fv(\overline{D}_2) \qquad \texttt{subtypeDecl}_{non-\mu}(\overline{D}_1, \overline{D}_2) = \texttt{true}$$

By inversion on the derivation of $\Gamma \vdash \tau_1 \longmapsto T_1$ and $\Gamma \vdash \tau_2 \longmapsto T_2$ it follows that there exists some $\overline{\sigma}_1$ such that

$$\Gamma, z : x.L\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \longmapsto \overline{D}_1$$

Similarly, there exists some $\overline{\sigma}_2$ such that

$$\Gamma, z : x.L\{z \Rightarrow \overline{\sigma}_2\} \vdash \overline{\sigma}_1 \longmapsto \overline{D}_2$$

Since $z \notin fv(\overline{D}_1) \cup fv(\overline{D}_2)$, it follows that $z \notin fv(\overline{\sigma}_1) \cup fv(\overline{\sigma}_2)$. Thus, by strengthening we have

$$\Gamma \vdash \overline{\sigma}_1 \mapsto \overline{D}_1 \qquad \Gamma \vdash \overline{\sigma}_2 \mapsto \overline{D}_2$$

By the induction hypothesis and $\texttt{subtypeDecl}_{non-\mu}(\overline{D}_1, \overline{D}_2) = \texttt{true}$, we get

$$\Gamma \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2$$

Finally, we get the desired result by S-Rfn.

Case 15 ($T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$, $T_2 = x_2.L_2\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$)**.**

$$x_1 \neq x_2 \qquad L_1 \neq L_2 \qquad \bigvee_{T \in \overline{T}_1} \texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$$

In other words, there exists some T such that

$$T \;\in\; \overline{T}_1 \qquad \texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$$

By inversion on the derivation of $\Gamma \vdash \tau_1 \;\longmapsto\; x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$ there exists τ and $\overline{\sigma}_1$ such that

$$\begin{array}{cc} \tau_1 = x_1.L_1\{z \Rightarrow \overline{\sigma}_1\} & \Gamma, z : x_1.L_1\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \;\mapsto\; \overline{D}_1 \\ \Gamma \vdash \tau \;\mapsto\; T & \Gamma \vdash x_1.L_1\{z \Rightarrow \overline{\sigma}_1\} \;\leqslant:: \tau \end{array}$$

Now by the induction hypothesis and $\texttt{subtype}_{non-\mu}(T, T_2) = \texttt{true}$, we get $\Gamma \vdash \tau \;<:\; \tau_2$, and subsequently the desired result by S-Ext.

□

6.3 Type Safety

I now present an argument for type safety for *Wyv*$_{non\text{-}\mu}$ for a term syntax and typing that . Unfortunately subtyping in *Wyv*$_{non\text{-}\mu}$, as with that of *Wyv*$_{core}$, is not transitive. Again, this is due to same reasons that transitivity was so difficult to derive in DOT. In this section I construct an encoding for a term syntax to a version of DOT, and demonstrate that it preserves typing. Due to their similarity, the encoding of *Wyv*$_{non\text{-}\mu}$ targets Wadlerfest DOT.

6.3.1 Term Typing and Reduction

The syntax of *Wyv*$_{non\text{-}\mu}$ is defined in Figure 6.2 is only marginally different from that of Wadlerfest DOT. The only difference is in the syntax of new objects, which appears as $\nu(z : \tau)d$ in DOT, and declarations which include the intersection form $d \cap d$. This similarity in syntax is adopted primarily to

$$
\begin{array}{lll}
t & ::= & \textbf{Term} \\
 & \nu & \textit{value} \\
 & x\ y & \textit{application} \\
 & x.l & \textit{selection} \\
 & \mathbf{let}\ x\ = t\ \mathbf{in}\ t : \tau & \textit{let} \\
\nu & ::= & \textbf{Value} \\
 & \mathbf{new}\ \tau\{z \Rightarrow \overline{d}\} & \textit{object} \\
 & \lambda\ x : \tau.t\ \ : \tau & \textit{abstraction}
\end{array}
\qquad
\begin{array}{lll}
d & ::= & \textbf{Declaration} \\
 & L\ =\ \tau & \textit{type} \\
 & l : \tau = t & \textit{value} \\
E & ::= & \textbf{Eval. Context} \\
 & [\,] & \textit{hole} \\
 & \mathbf{let}\ x\ = E\ \mathbf{in}\ t : \tau & \\
 & \mathbf{let}\ x\ = \nu\ \mathbf{in}\ E : \tau &
\end{array}
$$

Figure 6.2: $Wyv_{non\text{-}\mu}$ Term Syntax

aid in the encoding of Wyv_{core} to DOT. As with DOT, the syntax uses Administrative Normal Form [76] for the sake of simplicity. Thus, all reducible operations (function applications and member selections) may only involve variables ($x\ y$ or $x.l$).

While the syntax is fairly similar to that of Wadlerfest DOT, term typing is significantly different. The term typing is defined in Figure 6.3. This more complete term typing subsumes that of Figure 3.10, extending those rules with the typing rules for the terms of Figure 6.2. Thus, typing for variables (T-Var), variables typed with type refinements (T-Rec and T-Rfn), selection types (T-Sel) and variables typed with equal or nominal type definitions (T-Upper and T-Lower) remain unchanged. We extend these rules with type rules for member access, abstractions, applications, new objects and let expressions. A member access on a variable has the type of the member associated with the type of the receiver variable (T-Acc). An abstraction has a dependent function type if the body of the abstraction subtypes the return type (T-Abs). An abstraction application has the return type of the abstraction if the argument subtypes the argument type (T-App). A let expression has the type of the body if the body is appropriately typed with respect to the bound term. New object initializations are typed using

$$\boxed{\Gamma \vdash t \ : \ \tau, \ d \ : \ \sigma}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \ : \ \tau} \ (\text{T-}\textsc{Var}) \qquad \frac{\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\} \qquad \sigma \in \overline{\sigma}}{\Gamma \vdash x \ : \ \{[x/z]\sigma\}} \ (\text{T-}\textsc{Rec})$$

$$\frac{\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\}}{\Gamma \vdash x \ : \ \tau} \ (\text{T-}\textsc{Rfn}) \qquad \frac{\Gamma \vdash x \ : \ y.L \qquad \Gamma \vdash y \ : \ \{L \ \leqslant \tau\}}{\Gamma \vdash x \ : \ \tau} \ (\text{T-}\textsc{Sel})$$

$$\frac{\Gamma \vdash x \ : \ \{L \preceq\!/\!= \tau\}}{\Gamma \vdash x \ : \ \{L \ \leqslant \tau\}} \ (\text{T-}\textsc{Upper}) \qquad \frac{\Gamma \vdash x \ : \ \{L \ = \ \tau\}}{\Gamma \vdash x \ : \ \{L \ \geqslant \tau\}} \ (\text{T-}\textsc{Lower})$$

$$\frac{\Gamma \vdash x \ : \ \{l : \tau\}}{\Gamma \vdash x.l \ : \ \tau} \ (\text{T-}\textsc{Acc})$$

$$\frac{\Gamma, x : \tau_1 \vdash t \ : \ \tau \qquad \Gamma, x : \tau_1 \vdash \tau \ <: \ \tau_2 \qquad x \ \notin \ fv(\tau_1)}{\Gamma \vdash (\lambda \ (x : \tau_1).t : \tau_2) \ : \ \forall(x : \tau_1).\tau_2} \ (\text{T-}\textsc{Abs})$$

$$\frac{\Gamma \vdash x \ : \ \forall(z : \tau').\tau \qquad \Gamma \vdash y \ : \ \tau_2 \qquad \Gamma \vdash \tau_2 \ <: \ \tau'}{\Gamma \vdash x \ y \ : \ [y/z]\tau} \ (\text{T-}\textsc{App})$$

$$\frac{\Gamma \vdash t_1 \ : \ \tau_1 \qquad \Gamma, x : \tau_1 \vdash t_2 \ : \ \tau_2 \qquad \Gamma, x : \tau_1 \vdash \tau_2 \ <: \ \tau \qquad x \ \notin \ fv(\tau)}{\Gamma \vdash \texttt{let} \ x \ = t_1 \ \texttt{in} \ t_2 : \tau \ : \ \tau} \ (\text{T-}\textsc{Let})$$

$$\frac{\Gamma \vdash \tau \ \cong \ \{z \Rightarrow \overline{\sigma}\} \qquad \overset{\overline{d} \ \textit{has distinct labels}}{\Gamma, z : \tau \vdash \overline{d} \ : \ \overline{\sigma}'} \qquad \Gamma, z : \tau \vdash \overline{\sigma}' \ <: \ \overline{\sigma}}{\Gamma \vdash \texttt{new} \ \tau\{z \Rightarrow \overline{d}\} : \ \tau} \ (\text{T-}\textsc{New})$$

$$\Gamma \vdash L \ = \ \tau \ : \ L \ = \ \tau \ \ (\text{T-}\textsc{Type}) \qquad \frac{\Gamma \vdash t \ : \ \tau' \qquad \Gamma \vdash \tau' \ <: \ \tau}{\Gamma \vdash (l : \tau \ = \ t) \ : \ (l : \tau)} \ (\text{T-}\textsc{Value})$$

Figure 6.3: $Wyv_{non\text{-}\mu}$ Term Typing

$$\Gamma \vdash \top \ \cong \ \{z \Rightarrow \emptyset\} \qquad \frac{\Gamma \vdash \tau \ \cong \ \tau'}{\Gamma \vdash \tau\{z \Rightarrow \overline{\sigma}\} \cong flat(\tau', \overline{\sigma}, z)} \qquad \frac{\Gamma \vdash x \ : \ \{L \preceq\!/\!= \tau\} \quad \Gamma \vdash \tau \ \cong \ \tau'}{\Gamma \vdash x.L \cong \tau'}$$

Figure 6.4: $Wyv_{non\text{-}\mu}$ Member Expansion

$$\frac{v \;=\; \lambda(z:\tau).t:\tau'}{\texttt{let}\ x \;= v\ \texttt{in}\ E[x\ y]:\tau \longrightarrow \texttt{let}\ x \;= v\ \texttt{in}\ E[[y/z]t]:\tau}\quad\text{(R-App)}$$

$$\frac{v \;=\; \texttt{new}\ \tau\{z \Rightarrow \overline{d}\} \qquad (l:\tau' \;=\; t) \in \overline{d}}{\texttt{let}\ x \;= v\ \texttt{in}\ E[x.l]:\tau \longrightarrow \texttt{let}\ x \;= v\ \texttt{in}\ E[t]:\tau}\quad\text{(R-Acc)}$$

$$\texttt{let}\ x \;= y\ \texttt{in}\ t:\tau \;\longrightarrow\; [y/x]t \quad\text{(R-Var)}$$

$$\frac{t' = (\texttt{let}\ y \;= t_1\ \texttt{in}\ t_2:\tau')}{\texttt{let}\ x \;= t'\ \texttt{in}\ t:\tau \;\longrightarrow\; \texttt{let}\ y \;= t_1\ \texttt{in}\ (\texttt{let}\ x \;= t_2\ \texttt{in}\ t:\tau):\tau}\quad\text{(R-Let)}$$

$$\frac{t \;\longrightarrow\; t'}{E[t] \;\longrightarrow\; E[t']}\quad\text{(R-Ctx)}$$

Figure 6.5: *Wyv*$_{non\text{-}\mu}$ Operational Semantics

a secondary judgement, member expansion (Figure 7.8), that associates a type with a set of declaration types. If the declarations of a new object initialization have the type of the expansion, then the new object has the declared type (T-New). Finally, type and value definitions are typed by T-Type and T-Value respectively.

The type expansion judgement in Figure 6.4 is necessary in *Wyv*$_{non\text{-}\mu}$ due to the added nominality of type refinements and nominal type definitions when compared to a calculus such as DOT. Expansion extracts the member types of an object type. $\top$ expands into an empty refinement on $\top$, type refinements expand into the expansion of the base type flattened with the refinement. Only nominal or exact selections are expandable, and expand to the expansion of their defined type.

The term reduction is also derived from Wadlerfest DOT, and is defined by the rules in Figure 6.5. An application $x\ y$ within the binding $\texttt{let}\ x \;=$

$\lambda(x : \tau).t : \tau'$ reduces to the body of the abstraction (t), with the argument substituted in (R-App). A member selection $x.l$ within the binding **let** x = **new** $\tau\{z \Rightarrow \overline{d}\}$, reduces to the value bound by the label (R-Acc). Let statements are reduced if either the bound term is bound in some other binding (R-Var), or if the bound term is itself another binding (R-Let). Finally, context reduction is captured by R-Ctx.

6.3.2 Encoding $Wyv_{non\text{-}\mu}$ in Wadlerfest DOT

The argument for type safety of $Wyv_{non\text{-}\mu}$ is derived from that of Wadlerfest DOT. I define a mapping in Figure 6.6 from $Wyv_{non\text{-}\mu}$ types and terms to Wadlerfest DOT terms and types. The subsequent type safety argument is constructed by demonstrating that this mapping preserves subtyping, typing and reduction. The primary differences between the two syntaxes is the manner in which multiple member declarations and member declaration types are captured. In $Wyv_{non\text{-}\mu}$, multiple member declaration types are represented in a set form ($\overline{\sigma}$) as in a type refinement. Similarly, a set of member declarations is included as part of a new object. DOT makes use of intersections of declaration types to capture multiple declarations, and intersections of member declarations to capture multiple member declarations. These differences is visible in Figure 6.6. Most encodings are relatively straight forward and uninteresting, however the divergence between the two calculi is evident in the encodings of type refinements ($\mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\})$) and new object initializations ($\mathcal{D}($**new** $\tau\{z \Rightarrow \overline{d}\}))$

Type safety of $Wyv_{non\text{-}\mu}$ is then constructed in five theorems:

1. Theorem 6.3.1 proves that variable typing in $Wyv_{non\text{-}\mu}$ implies variable typing in DOT.

2. Theorem 6.3.2 proves that subtyping in $Wyv_{non\text{-}\mu}$ implies an equivalent subypting in DOT.

3. Theorem 6.3.3 proves that typing in $Wyv_{non\text{-}\mu}$ implies an equivalent typing in DOT.

$$\begin{array}{lcl}
\mathcal{D}(\top) & = & \top \\
\mathcal{D}(\bot) & = & \bot \\
\mathcal{D}(x) & = & x \\
\mathcal{D}(x.L) & = & x.L \\
\mathcal{D}(\forall(x:\tau_1).\tau_2) & = & \forall(x:\mathcal{D}(\tau_1)).\mathcal{D}(\tau_2) \\
\mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\}) & = & \mu(z:\mathcal{D}(\tau) \wedge (\bigwedge_{\sigma\in\overline{\sigma}} \mathcal{D}(\{\sigma\}))) \\
 & & \\
\mathcal{D}(L \ \leqslant \tau) & = & L \ : \bot \ldots \mathcal{D}(\tau) \\
\mathcal{D}(L \ \geqslant \tau) & = & L \ : \mathcal{D}(\tau) \ldots \top \\
\mathcal{D}(L \ = \ \tau) & = & L \ : \mathcal{D}(\tau) \ldots \mathcal{D}(\tau) \\
\mathcal{D}(L \ \preceq \ \tau) & = & L \ : \mathcal{D}(\tau) \ldots \mathcal{D}(\tau) \\
\mathcal{D}(l:\tau) & = & l:\mathcal{D}(\tau) \\
 & & \\
\mathcal{D}(x) & = & x \\
\mathcal{D}(x \ y) & = & x \ y \\
\mathcal{D}(x.l) & = & x.l \\
\mathcal{D}(\texttt{let } x \ = t_1 \texttt{ in } t_2 :) & = & \texttt{let } x \ = \mathcal{D}(t_1) \texttt{ in } \mathcal{D}(t_2) \\
\mathcal{D}(\texttt{new } \tau\{z \Rightarrow \overline{d}\}) & = & \nu(z:\mathcal{D}(\tau)) \bigwedge_{d\in\overline{d}} \mathcal{D}(d) \\
\mathcal{D}(\lambda(x:\tau_1).t:\tau_2) & = & \lambda x:\mathcal{D}(\tau_1).\mathcal{D}(t) \\
 & & \\
\mathcal{D}(L \ = \ \tau) & = & L \ = \ \mathcal{D}(\tau) \\
\mathcal{D}(l:\tau = t) & = & l = \mathcal{D}(t)
\end{array}$$

Figure 6.6: $Wyv_{non\text{-}\mu}$ to Wadlerfest DOT Encoding

4. Theorem 6.3.4 proves that term reduction in $Wyv_{non\text{-}\mu}$ implies an equivalent term reduction in DOT.

5. Theorem 6.3.5 proves that term reduction in $Wyv_{non\text{-}\mu}$ does not get stuck.

Theorem 6.3.1 ($Wyv_{non\text{-}\mu}$ variable typing implies DOT variable typing)**.** *For all* Γ, x, τ_x, *if* $\Gamma \vdash x \ : \ \tau_x$ *then* $\mathcal{D}(\Gamma) \vdash x \ : \ \mathcal{D}(\tau_x)$.

Proof. We construct the proof by mutual induction on the derivation of $\Gamma \vdash x \ : \ \tau_x$.

Case 1 (T-Var).

$$\Gamma(x) = \tau_x$$

It follows that

$$\mathcal{D}\Gamma(x) = \mathcal{D}(\tau_x)$$

and thus, $\mathcal{D}(\Gamma) \vdash x \ : \ \mathcal{D}(\tau_x)$ (by Var [75]).

Case 2 (T-Rec).

$$\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\} \qquad \sigma \in \overline{\sigma} \qquad \tau_x = \{[x/z]\sigma\}$$

By the induction hypothesis, we have

$$\mathcal{D}(\Gamma) \vdash x : \ \mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\})$$

By the definition of $\mathcal{D}$ we have

$$\begin{array}{lll} \mathcal{D}(\Gamma) \vdash x & : & \mu(z : \mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}} \{\mathcal{D}(\sigma)\})) & \text{(by defn. of } \mathcal{D}) \\ \mathcal{D}(\Gamma) \vdash x & : & [x/z]\mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}} \{\mathcal{D}(\sigma)\}) & \text{(by Rec-E)} \\ \mathcal{D}(\Gamma) \vdash x & : & [x/z]\{\mathcal{D}(\sigma)\} & \text{(by Sub, And}_1\text{-<: and And}_2\text{-<:)} \\ \mathcal{D}(\Gamma) \vdash x & : & \{\mathcal{D}([x/z]\sigma)\} & \text{(by defn. of } \mathcal{D}) \end{array}$$

The desired result.

Case 3 (T-Rfn).

$$\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\} \qquad \tau_x = \tau$$

By the induction hypothesis, we have

$$\mathcal{D}(\Gamma) \vdash x : \ \mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\})$$

By the definition of $\mathcal{D}$ we have

$$\begin{array}{lll} \mathcal{D}(\Gamma) \vdash x & : & \mu(z : \mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}} \{\mathcal{D}(\sigma)\})) & \text{(by defn. of } \mathcal{D}) \\ \mathcal{D}(\Gamma) \vdash x & : & [x/z]\mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}} \{\mathcal{D}(\sigma)\}) & \text{(by Rec-E)} \\ \mathcal{D}(\Gamma) \vdash x & : & [x/z]\{\mathcal{D}(\tau)\} & \text{(by Sub and And}_1\text{-<:)} \\ \mathcal{D}(\Gamma) \vdash x & : & \{\mathcal{D}([x/z]\tau)\} & \text{(by defn. of } \mathcal{D}) \\ \mathcal{D}(\Gamma) \vdash x & : & \{\mathcal{D}(\tau)\} & \text{(as } z \ \notin \ fv(\tau)) \end{array}$$

The desired result.

Case 4 (T-Sel)**.**

$$\Gamma \vdash x \ : \ y.L \qquad \Gamma \vdash y \ : \ \{L \ \leqslant \tau_x\}$$

By the induction hypothesis and the definition of $\mathcal{D}$ we have

$$\mathcal{D}(\Gamma) \vdash x \ : \ y.L \qquad \mathcal{D}(\Gamma) \vdash y \ : \ \{L \ : \bot \ldots \mathcal{D}(\tau_x)\}$$

By Sel-<:, $\mathcal{D}(\Gamma) \vdash y.L \ <: \ \mathcal{D}(\tau_x)$, and then by Sub we get the desired result: $\mathcal{D}(\Gamma) \vdash \ x \ : \ \mathcal{D}(\tau_x)$.

Case 5 (T-Upper)**.**

$$\Gamma \vdash x \ : \ \{L \preceq\!/_{=} \tau\} \qquad \tau_x = \{L \ \leqslant \tau\}$$

By the induction hypothesis:

$$\mathcal{D}(\Gamma) \vdash x \ : \ \{L \ : \mathcal{D}(\tau) \ldots \mathcal{D}(\tau)\}$$

By Typ-<:-Typ and Sub we have the desired result:

$$\mathcal{D}(\Gamma) \vdash x \ : \ \{L \ : \bot \ldots \mathcal{D}(\tau)\}$$

Case 6 (T-Lower)**.** By similar reasoning to T-Upper we get the desired result.

□

Theorem 6.3.2 (DOT subsumes $Wyv_{non\text{-}\mu}$ subtyping)**.** *For all* Γ, τ_1 *and* τ_2, *if* $\Gamma \vdash \tau_1 \ <: \ \tau_2$ *then* $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) \ <: \ \mathcal{D}(\tau_2)$.

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash \tau_1 \ <: \ \tau_2$.

Case 1 (S-Rfl)**.** Trivial.

Case 2 (S-Bot)**.** Trivial.

Case 3 (S-Top)**.** Trivial.

Case 4 (S-Upper)**.**

$$\tau_1 = x.L \qquad \Gamma \vdash x \ : \ \{L \ \leqslant \tau\} \qquad \Gamma \vdash \tau \ <: \ \tau_2$$

By Theorem 7.4.1 it follows that $\mathcal{D}(\Gamma) \vdash x \ : \ \{L \ : \_ \ldots \mathcal{D}(\tau_1\})$. The induction hypothesis gives $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) \ <: \ \mathcal{D}(\tau_2)$. By DOT subtyping (Sel-<: and Trans) we get the desired result :$\mathcal{D}(\Gamma) \vdash x.L \ <: \ \mathcal{D}(\tau_2)$

Case 5 (S-Lower)**.** By similar reasoning to Case 4.

Case 6 (S-All)**.** By simple application of the induction hypothesis to the premises.

Case 7 (S-Refine)**.**

$$\tau_1 = \tau\{z \Rightarrow \overline{\sigma}_1\} \qquad \tau_2 = \tau\{z \Rightarrow \overline{\sigma}_2\} \qquad \Gamma, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2$$
$$\mathcal{D}(\tau_1) = \mu(z : \mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}_1} \{\mathcal{D}(\sigma)\})) \qquad \mathcal{D}(\tau_2) = \mu(z : \mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}_2} \{\mathcal{D}(\sigma)\}))$$

By the induction hypothesis it follows that $\mathcal{D}(\Gamma, z : \tau\{z \Rightarrow \overline{\sigma}_1\}) \vdash \mathcal{D}(\overline{\sigma}_1) \ <: \ \mathcal{D}(\overline{\sigma}_2)$. By intersection subtyping (And$_1$-<:) in DOT we have $\mathcal{D}(\Gamma), z : \mathcal{D}(\tau_1) \vdash \mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}_1} \mathcal{D}(\sigma)) \ <: \ \mathcal{D}(\tau)$. Intersection subtyping (And$_2$-<:) and transitivity (Trans) in DOT also gives $\mathcal{D}(\Gamma), z \ : \ \mathcal{D}(\tau_1) \ \vdash \ \mathcal{D}(\tau) \wedge (\bigwedge_{\sigma \in \overline{\sigma}_1} \mathcal{D}(\sigma)) \ <: \ \bigwedge_{\sigma \in \overline{\sigma}_2} \mathcal{D}(\sigma)$. Thus the desired result is derived in DOT by <:-And.

Case 8 (S-Extend)**.**

$$\frac{\Gamma \vdash \tau_1 \ \leqslant:: \ \tau \qquad \Gamma \vdash \tau \ <: \ \tau_2}{\Gamma \vdash \tau_1 \ <: \ \tau_2}$$

By case analysis on the derivation of $\Gamma \vdash \tau_1 \leqslant:: \tau$:

Subcase 1 (E-Upper)**.** This is essentially upper bound subtyping, and thus the reasoning from Case 4 applies.

Subcase 2 (E-Refine).

$$\tau_1 \;= \tau'\{z \Rightarrow \overline{\sigma}\} \qquad \Gamma \vdash \tau' \;\leqslant\!:: \;\tau'' \qquad \tau = flat(\tau'', \overline{\sigma}, z)$$

By the induction hypothesis we have $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau) \;<:\; \mathcal{D}(\tau_2)$. Now we demonstrate that $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) \;<:\; \mathcal{D}(\tau)$: By case analysis on τ'' for valid applictions of $flat$ we have either: $\tau'' = \top$, $x.L$ or $\tau'''\{z \Rightarrow \overline{\sigma}'''\}$. The first two cases are easily demonstrated by the DOT subtyping rule And$_1$-<:. Otherwise: $\tau'' = \tau'''\{z \Rightarrow \overline{\sigma}'''\}$ where $\overline{\sigma}''' = \overline{\sigma}_0 \cup \overline{\sigma}$. Again $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) \;<:\; \mathcal{D}(\tau)$ is achieved by successive application of <:-And, And$_1$-<: and And$_2$-<:. Now by the DOT transitivity rule we have: $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) \;<:\; \mathcal{D}(\tau_2)$, the desired result.

□

Lemma 6.3.1. *For all Γ, τ and τ' such that $\Gamma \vdash \tau \;\cong\; \tau'$, then $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau') \;<:\; \mathcal{D}(\tau)$.*

Proof. By induction on the derivation of $\Gamma \vdash \tau \;\cong\; \tau'$:

Case 1.

$$\tau = \top \qquad \tau' = \{z \Rightarrow \emptyset\}$$

Trivial by Top.

Case 2.

$$\tau = \tau''\{z \Rightarrow \overline{\sigma}\} \qquad \Gamma \vdash \tau'' \;\cong\; \{z \Rightarrow \overline{\sigma}''\} \qquad \tau' = flat(\{z \Rightarrow \overline{\sigma}''\}, z, \overline{\sigma})$$

By the induction hypothesis:

$$\mathcal{D}(\Gamma) \vdash \mathcal{D}(\{z \Rightarrow \overline{\sigma}''\}) \;<:\; \mathcal{D}(\tau'')$$

By the definition of $flat$ and $\mathcal{D}$ we get

$$\mathcal{D}(\tau) = \mu(z : \mathcal{D}(\tau'') \wedge (\bigwedge_{\sigma_i \in \overline{\sigma}} \{\sigma_i\}))$$
$$\mathcal{D}(\tau') = \mu(z : \top \wedge (\bigwedge_{\sigma_j \in \overline{\sigma}''} \{\sigma_j\}) \wedge (\bigwedge_{\sigma_i \in \overline{\sigma}} \{\sigma_i\}))$$

The result is then derived by

□

Theorem 6.3.3 (DOT subsumes Wyv_{core} typing)**.** *For all* Γ*,* t *and* τ*, if* $\Gamma \vdash t \ : \ \tau$ *then* $\mathcal{D}(\Gamma) \vdash \mathcal{D}(t) \ : \ \mathcal{D}(\tau)$

Proof. This is the more general form of Theorem 7.4.1. By induction on the derivation of $\Gamma \vdash t \ : \ \tau$: For cases T-Var, T-Rec, T-Rfn, T-Sel, T-Upper and T-Lower, the desired result can be demonstrated by Theorem 6.3.1.

Case 1 (T-Acc)**.**

$$\Gamma \vdash x \ : \ \{l : \tau\}$$

By the induction hypothesis and definition of $\mathcal{D}$ we have:

$$\mathcal{D}(\Gamma) \vdash x \ : \ \top \wedge \{l : \mathcal{D}(\tau)\}$$

Thus, by Sub and And$_2$-<: we have:

$$\mathcal{D}(\Gamma) \vdash x \ : \ \{l : \mathcal{D}(\tau)\}$$

And finally, by {}-E we have the desired result:

$$\mathcal{D}(\Gamma) \vdash x.l \ : \ \mathcal{D}(\tau)$$

Case 2 (T-App)**.**

$$\Gamma \vdash x \ : \ \forall(z : \tau_1).\tau_2 \qquad \Gamma \vdash y \ : \ \tau_1' \qquad \Gamma \vdash \tau_1' \ <: \ \tau_1 \qquad \tau = [y/z]\tau_2$$

By the induction hypothesis and Theorem 6.3.2, we have

$$\begin{gathered}\mathcal{D}(\Gamma) \vdash x \ : \ \forall(z : \mathcal{D}(\tau_1)).\mathcal{D}(\tau_2) \\ \mathcal{D}(\Gamma) \vdash y \ : \ \mathcal{D}(\tau_1') \qquad \mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1') \ <: \ \mathcal{D}(\tau_1)\end{gathered}$$

By Sub, we get

$$\mathcal{D}(\Gamma) \vdash y \ : \ \mathcal{D}(\tau_1)$$

Then, by All-E we get the desired result:

$$\mathcal{D}(\Gamma) \vdash x\ y \ : \ \mathcal{D}([y/z]\tau_2)$$

Case 3 (T-Abs).

$$\Gamma, x : \tau_1 \vdash t \ : \ \tau \qquad \Gamma, x : \tau_1 \vdash \tau \ <: \ \tau_2$$
$$x \notin \ fv(\tau_1) \qquad t = \lambda(x : \tau_1).t : \tau_2 \qquad \tau = \forall(x : \tau_1).\tau_2$$

By the induction hypothesis and Theorem 7.4.2, we get

$$\mathcal{D}(\Gamma), x : \mathcal{D}(\tau_1) \vdash \mathcal{D}(t) \ : \ \mathcal{D}(\tau) \qquad \mathcal{D}(\Gamma), x : \mathcal{D}(\tau_1) \vdash \mathcal{D}(\tau) \ <: \ \mathcal{D}(\tau_2)$$

Also, by the application of $\mathcal{D}$ we get $x \notin \ fv(\mathcal{D}(\tau_1))$. Subsequently, by Sub we get:

$$\mathcal{D}(\Gamma), x : \mathcal{D}(\tau_1) \vdash \mathcal{D}(t) \ : \ \mathcal{D}(\tau_2)$$

And finally, by All-I we get the desired result:

$$\mathcal{D}(\Gamma) \vdash \lambda(x : \tau_1).t \ : \ \forall(x : \tau_1).\mathcal{D}(\tau_2)$$

Case 4 (T-Let).

$$\Gamma \vdash t_1 \ : \ \tau_1 \qquad \Gamma, x : \tau_1 \vdash t_2 \ : \ \tau_2 \qquad \Gamma, x : \tau_1 \vdash \tau_2 \ <: \ \tau \qquad x \notin \ fv(\tau)$$
$$t = (\texttt{let}\ x \ = t_1 \ \texttt{in}\ t_2 : \tau) \qquad \mathcal{D}(t) = \texttt{let}\ x = \mathcal{D}(t_1) \ \texttt{in}\ \mathcal{D}(t_2)$$

By the induction hypothesis and Theorem 6.3.2 we get:

$$\mathcal{D}(\Gamma) \vdash \mathcal{D}(t_1) \ : \ \mathcal{D}(\tau_1)$$
$$\mathcal{D}(\Gamma), x : \mathcal{D}(\tau_1) \vdash \mathcal{D}(t_2) \ : \ \mathcal{D}(\tau_2) \qquad \mathcal{D}(\Gamma), x : \mathcal{D}(\tau_1) \vdash \mathcal{D}(\tau_2) \ <: \ \mathcal{D}(\tau)$$

By Sub we get

$$\mathcal{D}(\Gamma), x : \mathcal{D}(\tau_1) \vdash \mathcal{D}(t_2) \ : \ \mathcal{D}(\tau)$$

By the definition of $\mathcal{D}$ we have $x \notin \ fv(\mathcal{D}(\tau))$. Thus, by Let we get get the the desired result:

$$\mathcal{D}(\Gamma) \vdash (\texttt{let}\ x = \mathcal{D}(t_1) \ \texttt{in}\ \mathcal{D}(t_2)) \ : \ \mathcal{D}(\tau)$$

Case 5 (T-New).

$$\Gamma \vdash \tau \ \cong \ \{z \Rightarrow \overline{\sigma}\}$$
$$\Gamma, z : \{z \Rightarrow \overline{\sigma}'\} \vdash \overline{d} \ : \ \overline{\sigma}' \qquad \Gamma, z : \{z \Rightarrow \overline{\sigma}'\} \vdash \overline{\sigma}' \ <: \ \overline{\sigma}$$

Case 6 (T-Type).

Case 7 (T-Value).

□

Theorem 6.3.4 (DOT Term Reduction is equivalent to $Wyv_{non\text{-}\mu}$ Term Reduction)**.** *For all* t *and* t'*, if* $t \longrightarrow t'$ *then* $\mathcal{D}(t) \longrightarrow \mathcal{D}(t')$*.*

Proof. The result is achieved easily by induction on the derivation of $t \longrightarrow t'$. □

Theorem 6.3.5 ($Wyv_{non\text{-}\mu}$ is Type Safe)**.** *For all* t *and* τ*, if* $\emptyset \vdash t : \tau$*, then term reduction of* t *does not get stuck.*

Proof. The result follows directly from Theorems 6.3.3 and 6.3.4. □

6.4 Expressiveness

$Wyv_{non\text{-}\mu}$ does not allow for subtyping of recursive types, and while this is a significant loss in expressiveness, the fact that Scala also does not allow for subtyping for recursive type refinements suggests there is degree of expressiveness that is valuable.

While Scala does not allow for subtyping of recursive type refinements, it does allow for subtyping of recursively defined nominal types, that is recursively defined classes. Further, at the beginning of this chapter I briefly made the comparison between $Wyv_{non\text{-}\mu}$ and Wadlerfest DOT. This is not an entirely apt comparison. While it is certainly true that Wadlerfest DOT does not feature subtyping of recursive types, it should be noted that Wadlerfest DOT has a subsumption rule, along with an introduction rule for values with recursive types that somewhat make up for this. Consider the following example:

```
def recArg(x : {z => type E <: ⊤, type T = z.E}) = { ... }
```

In $\mathit{Wyv}_{\mathit{non}\text{-}\mu}$, there is no way to use the `recArg` function, because doing so would require subtyping of a recursive type. In Wadlerfest DOT, `recArg` can be called on the value `v : { z ⇒ type E = Int, type T = z.E }` since the following typing can be derived:

$$\dfrac{\dfrac{\dfrac{\mathtt{v\ :\ \{z\ \Rightarrow\ type\ E\ =\ Int,\ type\ T\ =\ z.E\}}}{\mathtt{v\ :\ \{type\ E\ =\ Int,\ type\ T\ =\ v.E\}}}\ \text{(by }\textsc{Rec-E}\text{)}}{\mathtt{v\ :\ \{type\ E\ <:\ \top,\ type\ T\ =\ v.E\}}}\ \text{(by }\textsc{Sub}\text{)}}{\mathtt{v\ :\ \{z\ \Rightarrow\ type\ E\ <:\ \top,\ type\ T\ =\ z.E\}}}\ \text{(by }\textsc{Rec-I}\text{)}$$

General subsumption is not possible in $\mathit{Wyv}_{\mathit{non}\text{-}\mu}$ typing for reasons covered in Section 3.1.2.

Chapter 7

Recursive Types

As discussed in Chapter 3, undecidability of subtyping in Wyv_{core} derives from the confluence of contra-variance and environment narrowing. Since neither one of these is inherently undecidable, in this Chapter, I define a variant that severs the link between the two by addressing recursive types, the central cause of environment narrowing in Wyv_{core}. In this Chapter I start by discussing the expressiveness of recursive types in Wyv_{core} (Section 7.1). I then present Wyv_{μ}, a variant of Wyv_{core} that places a syntactic restriction on where recursive types may be used, along with a proof of subtype decidability (Section 7.2). I subsequently demonstrate that Wyv_{μ} retains a high degree of expressiveness, in particular the ability to encode the subtyping of Java Generics (Section 7.3) along with the specific examples of expressiveness identified in Section 7.1. I finally demonstrate that both Wyv_{core} and Wyv_{μ} are type safe (Section 7.4).

7.1 Examples of Expressiveness with Recursive Types

In the opening remarks of Chapter 3, I mention that the design of any variations on Wyv_{core} must take into account not only subtype decidability, but expressiveness and type safety too. I leave the discussion of type safety for Section 7.4, and issues of decidability for Section 7.2 but I start by discussing several instances of expressiveness for recursive types that programmers might want.

While the following is not a full accounting of the expressiveness of recursive types in $\mathit{Wyv}_{\mathit{core}}$, it identifies core uses of recursive types. I will return to these examples in Section 7.3.

Polymorphic Data Types

As has already been discussed, bounded polymorphism as seen in not only functional languages such as System $F_{<:}$, but also object oriented languages Java and C$^\sharp$, is subsumed in expressiveness by type members in Wyvern and DOT. Chapter 3 has demonstrated that subtyping in $\mathit{Wyv}_{\mathit{core}}$ subsumes that of Java Generics. What was perhaps not explicitly noted in Chapter 3 was the role that recursive types play in the encoding of Java. The following example of a `Node` class in Java can be used to illustrate this.

```
class Map<K, V>
class Node<E, V> extends Map<E, Node>
```

The class `Node` extends the `Map` class, encoding a `Node` as a map where the edges of type `E` map to adjacent nodes. While it may not be immediately obvious, this encoding can only be constructed using recursive types. The use of `E` in the instantiation of `Map<E, Node>` is a usage of a self reference. The Wyvern encoding is given below.

```
type Map[K <: ⊤, V <: ⊤]
type Node[E <: ⊤, V <: ⊤] = Map[self.E, Node]
```

Recursive types are thus necessary for capturing such critical instances of expressiveness.

Family Polymorphism

One of the strengths of calculi like Wyv_{core} and DOT is the ability to define bespoke subtype lattices, specifying the relationships of abstract types without specifying the exact types themselves. The type below demonstrates how type relationships can be defined without explicitly defining the types themselves.

$$\{z \Rightarrow A \ \leqslant \top,\ B \ \geqslant z.A, B \ \leqslant z.C,\ C \ \leqslant \top\}$$

Very little has been said about what the types A, B and C are, but the relationship between them is defined, and we know that whatever the types eventually resolve to, $A \ <: \ B \ <: \ C$ will hold. A more concrete example can be seen in an alternate version of the `Node` example.

```
type Graph = { z ⇒
        type Node <: {val neighbors : Map[Edge, Node]}
        type Edge <: {val origin, destination : Node}
}
```

A limited amount of type information has been defined for `Node` or `Edge`, but we know that they are interdependent. Such a pattern is reliant on the presence of recursive types. Since the usage of `Node` and `Edge` are actually calls on the self variable, `z`, the definition of `Graph` requires recursion.

This manner of defining types is reminiscent of Family Polymorphism, a language feature that allows families of related types to be defined. Polymorphism is captured at the family level and relationship level, ensuring reuse across families while simultaneously ensuring type safety. We adapt the `gbeta` example from [36] to Wyvern:

```
type Graph = { z_1 ⇒
  type Node <: {def touches (e : z_2.Edge) : bool}
  type Edge <: {var n1 : z_1.Node
                var n2 : z_1.Node}
}

val graph : z_1.Graph = new Graph{ z_1 ⇒
  def newNode : z_1.Node =
    new { z_2 ⇒
      def touches (e : z_1.Edge) : bool =
        z_2 == e.n1 || z_2 == e.n2
    }
  def newEdge : z_1.Edge = new {var n1, n2 := null}
}
val onOffGraph : Graph = new Graph{ z_1 ⇒
  type Edge = {var enabled : bool
               var n1 : z_1.Node
               var n2 : z_1.Node}
  def newNode : z_1.Node =
    new { z_2 ⇒
      def touches (e : z_1.Edge) : bool =
        if(e.enabled){z_2 == e.n1 || z_2 == e.n2} else {false}
    }
  def newEdge : z_1.Edge = new {var enabled := false
                               var n1, n2 := null}
}
def build(g : Graph, n : g.Node, e : g.Edge, b : bool) =
  e.n1 := n
  e.n2 := n
  if(b == n.touches(e))
    print(''OK'')

def main =
  build(graph, graph.newNode,
        graph.newEdge, true)
  build(onOffGraph, onOffGraph.newNode,
        onOffGraph.newEdge, false)
  build(graph, graph.newNode,
        onOffGraph.newEdge, false) \\ compile-time error
```

Again, recursive types are required in the definition of `Graph`, `Node` and `Edge`. Such expressiveness arises naturally from path dependent types in the presence of recursive types, and demonstrates the importance of recursive types in Wyv_{core}.

7.2 Wyv_{μ}

In this section I define a syntactic restriction on recursive types that ensures decidability. The advantage of a syntactic restriction is that the semantics remain unchanged, ensuring the behaviour of code that observes the syntactic restriction remains unchanged.

7.2.1 Preservation of the Material/Shape Separation

Recursive types and environment narrowing have more implications than just added expressiveness or environmental expansion, they also have implications for the application of a Material/Shape separation. The Material/Shape separation relies on the detection of cycles as part of an initial step in type checking. Any proof of termination of subtyping relies on existence of a finite shape depth during subtyping, that is for any problem of the form $\Gamma \vdash \tau_1 \ <: \ \tau_2$, there is some finite depth at which a Shape occurs. Cycle detection is performed on a per-environment basis, that is for any specific environment there is some specific Shape depth. In a subtyping that includes environment narrowing, the environment is modified, and thus while the Shape depth might always be finite for any particular subtype question, the measure might not always be well-founded and strictly decreasing with every derivation step. Consider the derivation of $\Gamma \vdash \tau\{z \Rightarrow L \ \leqslant \tau_1\} \ <: \ \tau\{z \Rightarrow$

$L \leqslant \tau_2\}$:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma_1 \vdash \tau_1 \ <: \ \tau_2}}{\Gamma_1 \vdash L \leqslant \tau_1 \ <: \ L \leqslant \tau_2} \qquad \Gamma_1 = \Gamma, z : \tau\{z \Rightarrow L \leqslant \tau_1\}}{\Gamma \vdash \tau\{z \Rightarrow L \leqslant \tau_1\} \ <: \ \tau\{z \Rightarrow L \leqslant \tau_2\}}$$

The shape depth of $L \leqslant \tau_2$ in Γ_1 is not strictly smaller than the shape depth of $\tau\{z \Rightarrow L \leqslant \tau_2\}$ in Γ, as we are only able to statically measure the shape depth of $L \leqslant \tau_2$ in the environment $\Gamma, z : \tau\{z \Rightarrow L \leqslant \tau_2\}$. Thus, the Material/Shape separation is not in fact applicable for Wyv_{core}. Narrowing does not preserve shape depth on the left hand side. Narrowing does however preserve shape depth on the left hand side of the subtype derivation. That is, the shape depth of $L \leqslant \tau_1$ is strictly smaller than the shape depth of $\tau\{z \Rightarrow L \leqslant \tau_1\}$. Unfortunately, our ability to derive useful conclusions from this is limited as contra-variance means that narrowing can occur not only on the right hand side of a subtype proof search, but on the left hand side too. Wyv_{μ}, the variant of Wyv_{core} defined in this section addresses this issue directly.

7.2.2 $Wyv_{core} \longrightarrow Wyv_{\mu}$

The combination of both contra-variance and recursive types is required in the encoding of System $F_{<:}$. While the removal of recursive types would remove valuable expressiveness, the removal of contra-variance would also preclude all of the value of lower bounds. In this section I place a syntactic restriction on the Wyv_{core} type system to restrict the usage of recursive types in contra-variant positions.

Figure 7.1 provides modifications to the Wyv_{core} Syntax and Subtype Semantics from Chapter 3. Wyv_{μ} restricts recursive types from appearing in lower bounds. For convenience we attach this restriction to the existing Material/Shape separation. This requires a relatively simple restriction: pure

$$\begin{array}{lll} \eta & ::= & \textbf{Material Type} \\ & \vdots & \\ & \eta\{\overline{\delta}\} & \end{array}$$

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

Figure 7.1: Wyv_{μ} Syntax/Semantic Extension

material types are restricted from using recursive types. Originally only Shapes were restricted from usage in the lower bounds of type members, and in refinements on shapes, but this restriction is now extended to include refinements with self references. Note: the extension of the Material/Shape separation does not imply a prohibition of recursive types involving Materials, only that for a type to be considered "pure material", it must not contain any recursive types.

The semantic restriction is also fairly simple, subtyping of dependent function types requires invariant argument types. This is in line with the restricted semantics that Kernel $F_{<:}$ placed on System $F_{<:}$.

7.2.3 Subtype Decidability

As with both Wyv_{fix} and $Wyv_{\mathit{non}\text{-}\mu}$, the argument for subtype decidability for Wyv_{μ} is based on an encoding of types into the intermediate language of Wyv_{expand}, as described in Chapter 5. While in Wyv_{μ}, there is no equivalence between subtyping of two arbitrary types, and their respective expansions (for the reasons already discussed in Section 7.2.1), there does exist an equivalence for subtyping of the form $\Gamma \vdash \tau \ <: \ \eta$. That is, an equivalence exists between subtyping of Wyv_{μ} types and their equivalent type graphs if the type on the right-hand side is a material. The reason for this is that environment narrowing is guaranteed not to occur in subtyping of the form

$\Gamma \vdash \tau \ <: \ \eta$.

An algorithm for constructing a reduction from Wyv_μ subtyping to subtyping of only materials is given in Algorithms 2 and 3. Thus, the argument for subtype decidability of Wyv_μ is three part: (*i*) demonstrate that subtype questions of the form $\Gamma \vdash \tau \ <: \ \eta$ are reducible to subtyping of the expansion into $Wyv_{\texttt{expand}}$, (*ii*) demonstrate that Algorithm 2 is able to decide the subtyping defined in Figure 7.1 and (*iii*) demonstrate that Algorithm 2 terminates for all inputs.

Part (*i*) is demonstrated in Theorem 7.2.1. Part (*ii*) is demonstrated in Theorem 7.2.2. Part (*iii*) is demonstrated in Theorem 7.2.3. The proof uses the depth measure in Figure 7.2, whose finiteness is ensured by the Material/Shape separation.

Theorem 7.2.1 (Material Subtyping in Wyv_μ Wyvern)**.** *For all* Γ, τ, η, T_1 *and* T_2*, if* $\Gamma \vdash \tau \mapsto T_1$ *and* $\Gamma \vdash \eta \mapsto T_2$*, then* $\Gamma \vdash \tau \ <: \ \eta \iff$ $\texttt{subtype}(T_1, T_2)$.

Proof. Proof proceeds by case analysis on both directions of $\iff$

Case 1 ($\Rightarrow$)**.** Proceed by induction on the derivation of $\Gamma \vdash \tau \ <: \ \eta$

Subcase 1 (S-Top: $\Gamma \vdash \tau \ <: \ \top$)**.** By inversion on the derivation of $\Gamma \vdash \top \longmapsto T_2$ we have $T_2 = \top$, and we easily get the desired result

$$\texttt{subtype}(T_1, \top) = \texttt{true}$$

Subcase 2 (S-Bottom: $\Gamma \vdash \bot \ <: \ \eta$)**.** By inversion on the derivation of $\Gamma \vdash \bot \longmapsto T_1$ we have $T_1 = \bot$, and we easily get the desired result

$$\texttt{subtype}(\bot, T_2) = \texttt{true}$$

Subcase 3 (S-All: $\tau = \forall(x : \eta').\eta_1$, $\eta = \forall(x : \eta').\eta_2$ and $\Gamma \vdash \forall(x : \eta).\eta_1 \ <: \ \forall(x : \eta).\eta_2$)**.** By inversion on the derivation of $\Gamma \vdash \forall(x : \eta').\eta_1 \longmapsto T_1$, there exists T and T_1' such that

$$\Gamma \vdash \eta' \longmapsto T \qquad \Gamma, x : \eta' \vdash \eta_1 \longmapsto T_1'$$

```
subtype$_\eta$($\Gamma, \tau_1, \tau_2$)
    if $\tau_2$ is a material then
        subtype($T_1, T_2$) where $\Gamma \vdash \tau_1 \mapsto T_1$ and $\Gamma \vdash \tau_2 \mapsto T_2$
    else
        switch $\tau_1$ do
            case $\top$ do
                switch $\tau_2$ do
                    case $\top$ do true;
                    case $x.L$ do subtype$_\eta$($\Gamma, \tau_1$, lower_bound($\Gamma, x.L$)) ;
                    otherwise do false;
                end
            case $\bot$ do true;
            case $x_1.L_1$ do
                switch $\tau_2$ do
                    case $\top$ do true;
                    case $x_2.L_2$ do $(x_2 = x_2 \wedge L_1 = L_2) \vee$ subtype$_\eta$($\Gamma$, upper_bound($\Gamma, x_1.L_1$),
                      $\tau_2$)$\vee$ subtype$_\eta$($\Gamma, \tau_1$, lower_bound($\Gamma, x.L$)) ;
                    otherwise do subtype$_\eta$(upper_bound($\Gamma, x_1.L_1$), $\tau_2$) ;
                end
            case $\forall(x : \tau).\tau_1'$ do
                switch $\tau_2$ do
                    case $\top$ do true;
                    case $x.L$ do subtype$_\eta$($\Gamma, \tau_1$, lower_bound($x.L$)) ;
                    case $\forall(x : \tau').\tau_2'$ do (subtype$_\eta$($\Gamma; x : \tau$, $\tau_1'$, $\tau_2'$)) ;
                    otherwise do false;
                end
            case $\tau_1'\{z \Rightarrow \overline{\sigma}_1\}$ do
                switch $\tau_2$ do
                    case $\top$ do true;
                    case $x.L$ do subtype$_\eta$($\tau_1$, lower_bound($\Gamma, x.L$)) ;
                    case $\tau_2'\{z \Rightarrow \overline{\sigma}_2\}$ do
                        if $\tau_1' = \tau_2'$ then
                            (subtypeDecls$_\eta$($\Gamma, z : \tau_1, \overline{\sigma}_1, \overline{\sigma}_2$))
                        else
                            switch extends($\tau_1, \tau_2, \emptyset$) do
                                case Some($\overline{\sigma}$) do (subtypeDecls$_\eta$($\Gamma, z : \tau_1, \overline{\sigma}, \overline{\sigma}_2$));
                                otherwise do false;
                            end
                        end
                    otherwise do false;
                end
        end
    end
```

Algorithm 5: *Wyv_μ* Subtyping Reduction Algorithm

```
subtypeDecl$_\eta$($\Gamma, \sigma_1, \sigma_2$)
    switch $\sigma_1, \sigma_2$ do
        case $L_1 \leqslant \tau_1, L_2 \leqslant \tau_2$ do
            if $L_1 = L_2$ then subtype$_\eta$($\Gamma, \tau_1, \tau_2$);
            else false;
        case $L_1 = \tau_1, L_2 \leqslant \tau_2$ do
            if $L_1 = L_2$ then subtype$_\eta$($\Gamma, \tau_1, \tau_2$);
            else false;
        case $L_1 \geqslant \tau_1, L_2 \geqslant \tau_2$ do
            if $L_1 = L_2$ then subtype$_\eta$($\Gamma, \tau_2, \tau_1$);
            else false;
        case $L_1 = \tau_1, L_2 \geqslant \tau_2$ do
            if $L_1 = L_2$ then subtype$_\eta$($\Gamma, \tau_2, \tau_1$);
            else false;
        case $\tau_1 = \tau_1, L_2 = \tau_2$ do
            if $L_1 = L_2$ then subtype$_\eta$($\Gamma, \tau_2, \tau_1$) $\wedge$ subtype$_\eta$($\Gamma, \tau_1, \tau_2$);
            else false;
        case $L_1 \preceq \tau_1, L_2 \preceq \tau_2$ do
            if $L_1 = L_2$ then subtype$_\eta$($\Gamma, \tau_2, \tau_1$) $\wedge$ subtype$_\eta$($\Gamma, \tau_1, \tau_2$);
            else false;
        case $l_1 : \tau_1, l_2 : \tau_2$ do
            if $l_1 = l_2$ then subtype$_\eta$($\tau_1, \tau_2$);
            else false;
        otherwise do false;
    end
subtypeDecls$_\eta$($\Gamma, \overline{\sigma}_1, \overline{\sigma}_2$)
    switch $\overline{\sigma}_2$ do
        case $\emptyset$ do true;
        case $\{\sigma_2\} \cup \overline{\sigma}'_2$ do
            switch $\overline{\sigma}_1$ do
                case $\{\sigma_1\} \cup \overline{\sigma}'_1$ do
                    subtypeDecl$_\eta$($\Gamma, \sigma_1, \sigma_2$) $\vee$ subtypeDecls$_\eta$($\Gamma, \overline{\sigma}'_1, \overline{\sigma}'_2$)
                otherwise do false;
            end
    end
```

Algorithm 6: *Wyv*$_\mu$ Declaration Subtyping Reduction Algorithm

Similarly, there exists T_2' such that

$$\Gamma, x : \eta' \vdash \eta_2 \longmapsto T_2'$$

By reflexivity of the `subtype` algorithm, it follows that

$$\texttt{subtype}(T, T) = \texttt{True}$$

and by the induction hypothesis, it follows that

$$\texttt{subtype}(T_1', T_2') = \texttt{True}$$

Thus we have the desired result

$$\texttt{subtype}(T_1, T_2) = \texttt{subtype}(T, T) \wedge \texttt{subtype}(T_1', T_2') = \texttt{True}$$

Subcase 4 (S-Upper: $\tau = x.L, \Gamma \vdash x : \{L \leqslant \tau'\}, \Gamma \vdash \tau' <: \eta$)**.** By inversion on the derivation of $\Gamma \vdash x.L \mapsto T_1$, there exists $\overline{D}$ and T' such that

$$\Gamma \vdash x.L \longmapsto x.L \mapsto \overline{D} \qquad \Gamma \vdash \tau' \longmapsto T' \qquad (L \leqslant T') \in \overline{D}$$

by the definition of `ub` we have

$$T' \in \texttt{ub}(x.L \mapsto \overline{D})$$

By the induction hypothesis, we get

$$\texttt{subtype}(T', T_2) = \texttt{true}$$

thus by the definition of `subtype` we get the desired result

$$\texttt{subtype}(T_1, T_2) = \bigvee_{T \in \texttt{ub}(x.L \mapsto \overline{D})} \texttt{subtype}(T, T_2) = \texttt{True}$$

Subcase 5 (S-Lower: $\eta = x.M, \Gamma \vdash x \ni M \geqslant \tau', \Gamma \vdash \tau <: \tau'$)**.** By inversion on the derivation of $\Gamma \vdash x.M \mapsto T_2$, there exists $\overline{D}$ and T' such that

$$\Gamma \vdash x.M \longmapsto x.M \mapsto \overline{D} \qquad \Gamma \vdash \tau' \longmapsto T' \qquad (M \geqslant T') \in \overline{D}$$

by the definition of `lb` we have

$$T' \in \texttt{lb}(x.M \mapsto \overline{D})$$

Further, since lower bounds are restricted to materials, we know that τ' is a material. Now by the induction hypothesis, we get

$$\texttt{subtype}(T_1, T') = \texttt{true}$$

thus by the definition of `subtype` we get the desired result

$$\texttt{subtype}(T_1, T_2) = \bigvee_{T \in \texttt{lb}(x.M \mapsto \overline{D})} \texttt{subtype}(T_1, T) = \texttt{True}$$

Subcase 6 (S-Refine: $\tau = x.M\{z \Rightarrow \overline{\sigma}\}, \eta = x.M\{\overline{\delta}\}$, $\Gamma, z : x.M\{z \Rightarrow \overline{\sigma}\} \vdash \overline{\sigma} \ <: \ \overline{\delta}$)**.** By inversion on the derivation of $\Gamma \vdash x.M\{z \Rightarrow \overline{\sigma}\} \ \longmapsto \ T_1$ and $\Gamma \vdash x.M\{\overline{\delta}\} \ \longmapsto \ T_2$ there exists $\overline{D}_1$, $\overline{T}_1$, $\overline{D}_2$, and $\overline{T}_2$ such that

$$T_1 = x.M\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1 \quad \text{(a)} \qquad \Gamma, z : x.M\{z \Rightarrow \overline{\sigma}\} \vdash \overline{\sigma} \ \mapsto \overline{D}_1 \quad \text{(b)}$$

$$T_2 = x.M\{\overline{D}_2\} \mapsto \overline{T}_2 \quad \text{(c)} \qquad \Gamma, z : x.M\{\overline{\delta}\} \vdash \overline{\delta} \ \mapsto \overline{D}_2 \quad \text{(d)}$$

Since $z \notin fv(\overline{\delta})$, by applying environment strenghtening, and then weakening to (d) we have

$$\Gamma, z : x.M\{z \Rightarrow \overline{\sigma}\} \vdash \overline{\delta} \ \mapsto \overline{D}_2$$

Thus, by the induction hypothesis, we have

$$\texttt{subtype}(\overline{D}_1, \overline{D}_2) = \texttt{true}$$

giving us the desired result.

Subcase 7 (S-Extend: $\Gamma \vdash \tau \ \leqslant:: \ \tau', \Gamma \vdash \tau' \ <: \ \eta$)**.** By inversion on the derivation of $\Gamma \vdash \tau \ \leqslant:: \ \tau'$, either

1. $\tau = x.L$ and $\Gamma \vdash x \ : \{L \ \leqslant \tau'\}$ or

2. $\tau = x.L\{z \Rightarrow \overline{\sigma}\}$ and $\Gamma \vdash x \ \leqslant:: \ \tau'$

In the first case we proceed in the same fashion as Subcase 4.
In the second case:

By inversion on the derivation or $\Gamma \vdash x.L\{z \Rightarrow \overline{\sigma}\} \ \longmapsto \ T_1$, there exists $\overline{D}$, $\overline{T}$, T' such that

$$T_1 = x.L\{z \Rightarrow \overline{D}\} \ \mapsto \ \overline{T} \qquad T' \in \overline{T} \qquad \Gamma \vdash \tau' \ \longmapsto \ T'$$

Thus, by the induction hypothesis, we have

$$\texttt{subtype}(T', T_2)$$

and by the definition of `subtype` we easily arrive at the desired result.

Case 2 ($\Leftarrow$)**.** By induction on the call depth of $\texttt{subtype}(T_1, T_2)$:
Induction Hypothesis: for every subcall to `subtype`, if $\texttt{subtype}(T_1', T_2') = \texttt{true}$, then it follows that for all Γ', τ' and η' such that $\Gamma' \vdash \tau' \ \longmapsto \ T_1'$ and $\Gamma' \vdash \eta' \ \longmapsto \ T_2'$, then $\Gamma' \vdash \tau' \ <: \ \eta'$.

Subcase 1 (Base Case: $T_1 = \top, T_2 = \top$)**.** Trivial.

Subcase 2 (Inductive Case: $T_1 = \top, T_2 = x.M \ \mapsto \overline{D}$)**.**

$$\exists T_2' \in \texttt{lb}(x.M \mapsto \overline{D}) \qquad \texttt{subtype}(T_1, T_2') = \texttt{true}$$

By inversion on the derivation of $\Gamma \vdash \eta \ \longmapsto \ x.M \mapsto \overline{D}$, there exists some τ_2' such that

$$\Gamma \vdash \tau_2' \ \longmapsto \ T_2' \qquad \Gamma \vdash x \ : \ \{M \ \geqslant \tau_2'\}$$

Since τ_2' is a lower bound, it follows that it must be some pure material type η_2'. Thus, by the induction hypothesis, it follows that

$$\Gamma \vdash \tau \ <: \ \eta_2'$$

And then by S-Lower we get the desired result:

$$\Gamma \vdash \tau \ <: \ \eta$$

Subcase 3 (Base Case: $T_1 = x.L \mapsto \overline{D}, T_2 = \top$)**.** Trivial.

Subcase 4 (Base Case: $T_2 = x.M \mapsto \overline{D}_1, T_2 = x.M \mapsto \overline{D}_2$)**.** Trivially acheived by S-Rfl.

Subcase 5 (Inductive Case: $T_1 = x_1.L \mapsto \overline{D}_1, T_2 = x_2.M \mapsto \overline{D}_2$)**.** By similar reasoning to subcase 2, but on the upper bound instead of the lower bound.

Subcase 6 (Inductive Case: $T_1 = x_1.L \mapsto \overline{D}_1, T_2 = x_2.M \mapsto \overline{D}_2$)**.** By similar reasoning to subcase 2.

Subcase 7 (Inductive Case: $T_1 = \_, T_2 = x_2.M \mapsto \overline{D}_2$)**.** By similar reasoning to subcase 2.

Subcase 8 (Base Case: $T_1 = \forall(x : S_1).U_1, T_2 = \top$)**.** Trivial.

Subcase 9 (Inductive Case: $T_1 = \forall(x : S_1).U_1, T_2 = x_2.M \mapsto \overline{D}_2$)**.** By similar reasoning to subcase 2.

Subcase 10 (Inductive Case: $T_1 = \forall(x : S_1).U_1, T_2 = \forall(x : S_2).U_2$)**.** By simple application of the induction hypothesis.

Subcase 11 (Base Case: $T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1, T_2 = \top$)**.** Trivial.

Subcase 12 (Inductive Case: $T_1 = x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1, T_2 = x_2.M \mapsto \overline{D}_2$)**.** By similar reasoning to subcase 2.

Subcase 13 (Inductive Case: $T_1 = x_1.L\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$,
$T_2 = x_2.M\{z\}\overline{D}_2 \mapsto \overline{T}_2$)**.**

$$x_1 = x_2 \qquad L = M$$

By inversion on the derivation of $\Gamma \vdash T_1 \longmapsto x_1.L_1\{z \Rightarrow \overline{D}_1\} \mapsto \overline{T}_1$ and $\Gamma \vdash T_2 \longmapsto x_2.L_2\{z \Rightarrow \overline{D}_2\} \mapsto \overline{T}_2$ there exists some $\overline{\sigma}$ and $\overline{\delta}$.

$$\tau = x_1.L\{z \Rightarrow \overline{\sigma}\} \quad (1) \qquad\qquad \eta = x_2.M\{\overline{\delta}\} \quad (2)$$

$$\Gamma, z : x_1.L\{z \Rightarrow \overline{\sigma}\} \vdash \overline{\sigma} \longmapsto \overline{D}_1 \quad (3) \qquad \Gamma, z : x_1.M\{\overline{\delta}\} \vdash \overline{\delta} \longmapsto \overline{D}_2 \quad (4)$$

$$\mathcal{D}(\Gamma, \top) = 0 \qquad \mathcal{D}(\Gamma, \bot) = 0 \qquad \frac{\Gamma \vdash x \ni L \ \leqslant \tau}{\mathcal{D}(\Gamma, x.L) = 1 + \mathcal{D}(\Gamma, \tau)} \qquad \frac{\Gamma \vdash x \ni L \ \geqslant \tau}{\mathcal{D}(\Gamma, x.L) = 1 + \mathcal{D}(\Gamma, \tau)}$$

$$\frac{\Gamma \vdash x \ni L \ = \ \tau}{\mathcal{D}(\Gamma, x.L) = 1 + \mathcal{D}(\Gamma, \tau)} \qquad \frac{\Gamma \vdash x \ni L \ \preceq \ \tau}{\mathcal{D}(\Gamma, x.L) = 1 + \mathcal{D}(\Gamma, \tau)}$$

$$\mathcal{D}(\Gamma, \forall(x : \tau_1).\tau_2) = 1 + max(\mathcal{D}(\Gamma, \tau_1), \mathcal{D}(\Gamma; x : \tau_1, \tau_2)) \qquad \mathcal{D}(\Gamma, \tau\{z \Rightarrow \overline{\sigma}\}) = 0$$

Figure 7.2: Refinement Depth Measure on Wyv_{μ} Types

Since $z \notin fv(\overline{\delta})$, by application of environment strengthening followed by environement weakening to (4) we get

$$\Gamma, z : x_1.L\{z \Rightarrow \overline{\sigma}\} \vdash \overline{\delta} \longmapsto \ \overline{D}_2$$

Finally we get the desired result by application of the induction hypothesis.

Subcase 14 (Inductive Case: $T_1 = T_1'\{z \Rightarrow \overline{D}_1\} \ \mapsto \overline{T}_1$, $T_2 = T_2'\{z \Rightarrow \overline{D}_2\} \ \mapsto \overline{T}_2$)**.** The only possible sub-call has T_2 (and η) itself on the right-hand side, fulfilling the requirements for the induction hypothesis. Finally, the desired result is achieved by S-Ext.

□

Theorem 7.2.2 (Equivalence of Wyv_{μ} Subtyping)**.** *For all* Γ, τ_1, τ_2 *and* $\Gamma \vdash \tau_1 \ <: \ \tau_2 \iff \texttt{subtype}_{\eta}(\Gamma, \tau_1, \tau_2)$.

Proof. Algorithm 5 reduces questions of Wyv_{μ} subtyping between types to questions of subtyping between their expansions in Wyv_{expand}. As I noted in Section 5.1, given an environment Γ and a type τ, it is always possible to generate the expansion of that type. If τ_2 is a pure material, then the expansion is generated, and the `subtype` function is called. In all other cases, the types are syntactically matched upon. The notable point is that the only way a cycle may arise on the right, is through a selection type. Since all the lower bounds of all selection types are pure materials, such a case is sound by Theorem 7.2.1. In all other cases, types are syntactically bound, that is, all subsequent calls are to arguments that are strictly smaller syntactically. □

Theorem 7.2.3 (Wyv_μ Subtyping is Decidable)**.**

Proof. The result follows directly from Theorem and 7.2.2. □

7.3 Expressiveness of Wyv_μ

Polymorphic Data Types

Restricting recursive types from the lower bound of type definitions conflicts with several instances of expressiveness, most importantly those that use recursive types in type definitions of the form $L = \tau$. This restriction is implicit since $L = \tau$ specifies both a lower and an upper bound. This is a common form for type members and thus the restriction has wide ranging implications. Among those excluded are both of the examples mentioned in Section 7.1 as both use type members to specify exact types.

While the specific formulation of the examples of Parametric Polymorphism and Family Polymorphism in Section 7.1 are no longer expressible in Wyv_μ, the examples can still be expressed using the nominal form of type members introduced in Chapter 3 ($L \preceq \tau$). Types defined using a nominal definition may only be subtyped by explicit extension, and are not treated as having a lower bound in the same way that other type members are. The `Node` and type definitions are rewritten using the nominal form below.

```
type Node[E <: ⊤, V <: ⊤] ⪯ Map[self.E, Node]
```

As a result of using the nominal form for type definition, `Node` cannot be structurally from below inspected during subtyping. This represents a rather large conceptual split between nominal subtyping where recursive types may be used, and structural subtyping where recursive types are restricted to only upper bounds of type definitions. This is implies a potentially significant restriction on the expressiveness allowable by Scala, but not on a language such as Java where all subtyping is nominal.

Encoding Java in Wyv_{μ}

Chapter 3 already demonstrated that Java Generics represents a subset of Wyv_{core}'s subytyping, but it is valuable that Wyv_{μ} subsumes the subtyping of Java, in particular the version of Java subtyping present in Greenman et al. original Material/Shape separation. It should be noted that while Wyv_{μ} is able to express the types and subtyping of Java, it is not necessarily able to capture the dynamic semantics, in particular Wyvern does not contain inheritance.

A syntax and subtype semantics for a Java-like type system are given in Figures 7.3 and 7.4, while the encoding is given in Figure 7.5. The syntax and semantics of Java provided by Greenman et al. is in fact a superset of Java in that type parameters may be defined with both an upper and a lower bound while Java allows either an upper bound or a lower bound, but not both [38]. The encoding in Figures 7.3 and 7.4 only allows for single bound on type parameters, and are thus in-line with the expressiveness of Java.

Theorem 7.3.1. *The subtype sematics of Figure 7.4 are a subset of those in Chapter 3. That is, if* $\vdash \varsigma_1 \;<:\; \varsigma_2 \;\Rightarrow\; x : \{Wyv(CT)\} \vdash^{Wyv} Wyv(\varsigma_1) \;<:\; Wyv(\varsigma)$.

Proof. Letting $x : \{Wyv(CT)\} = \Gamma_{Wyv}$, proceed by induction on the derivation of $\vdash \varsigma_1 \;<:\; \varsigma_2$. SJ-Top: Trivial. SJ-Bot: Trivial. SJ-Class^{+}($\vdash C\langle +\varsigma_1\rangle \;<:\; C\langle +\varsigma_1\rangle$): By the induction hypothesis, we have $\Gamma_{Wyv} \vdash Wyv(\varsigma_1) \;<:\; Wyv(\varsigma_2)$, or more specifically we have $\Gamma_{Wyv} \vdash L \leqslant Wyv(\varsigma_1) \;<:\; L \leqslant Wyv(\varsigma_2)$. Since Java does not allow self references (α_C) in class type parameters, we can safely weaken the context to $\Gamma_{Wyv}, \alpha_C : \{\alpha_C \Rightarrow L \leqslant Wyv(\varsigma_1)\}$, giving us the desired result by S-Refine. SJ-Class^{-}: Following similar reasoning to SJ-Class^{+}. SJ-Extend: The substitution in Figure 7.3 is captured as a refinement on C' in the Wyv_{μ} encoding of class declaration of

$$
\begin{array}{llr@{\qquad}lll}
\varsigma & ::= & \textbf{Java Type} & & & \\
 & \bot & \textit{bottom} & \pm & ::= & \textbf{Variance} \\
 & \top & \textit{top} & & + & \textit{negative} \\
 & \alpha & \textit{variable} & & - & \textit{positive} \\
 & C\langle \pm\varsigma \rangle & \textit{class} & & &
\end{array}
$$

$$
\begin{array}{lllr}
CT & ::= & & \textbf{Class Table} \\
 & \emptyset & & \\
 & C\langle \pm \rangle \;\; <:: \;\; D\langle \pm\varsigma \rangle ; CT & &
\end{array}
$$

$$
\begin{array}{lcl}
[\alpha \mapsto {}^{\pm_1}/{\pm_2}\varsigma] + \varsigma' & = & +[\alpha \mapsto {}^{\pm_1}/{\pm_2}\varsigma]\varsigma' \\
[\alpha \mapsto {}^{\pm_1}/{\pm_2}\varsigma] - \varsigma' & = & -[\alpha \mapsto \pm_1^{-1}/{\pm_2}\varsigma]\varsigma' \\
[\alpha \mapsto {}^{\pm}/{\pm}\varsigma]\alpha & = & \varsigma \\
[\alpha \mapsto {}^{-}/{+}\varsigma]\alpha & = & \bot \\
[\alpha \mapsto {}^{+}/{-}\varsigma]\alpha & = & \top \\
[\alpha \mapsto {}^{\pm_1}/{\pm_2}\varsigma]C\langle \pm'\varsigma' \rangle & = & C\langle [\alpha \mapsto {}^{\pm_1}/{\pm_2}\varsigma] \pm' \varsigma \rangle \\
 & & \textit{where } +^{-1} = - \textit{ and } -^{-1} = +
\end{array}
$$

Figure 7.3: Java-like Syntax and Substitution

C in Figure 7.5. Since all substitutions that could possibly be applied are distinct, they can be safely extracted to a context that stores substitutions as bounds on each distinct α. Substitution replaces α with the relevant bound (positive types have $\bot$ as the lower bound, and negative types have $\top$ as the upper bound). During Wyvern subtyping, these bounds are looked up as they are needed using S-Upper and S-Lower. Thus, following this logic we can derive a relatively simple substitution lemma. Finally, we get the desired result using this lemma, the induction hypothesis and S-Extend. □

7.4 Type Safety

I now present a proof of type safety for Wyv_{core} for a term syntax and typing that extends that of Wyv_{core}. Unfortunately subtyping in Wyv_{core} (and thus Wyv_{μ}) is not transitive, for the same reasons that transitivity was such a

$$\vdash \varsigma \;<:\; \top \quad (\text{SJ-Top}) \qquad\qquad \vdash \bot \;<:\varsigma \quad (\text{SJ-Bot})$$

$$\frac{\vdash \varsigma_1 \;<:\; \varsigma_2}{\vdash C\langle +\varsigma_1\rangle \;<:\; C\langle +\varsigma_2\rangle} \quad (\text{SJ-Class}^{+}) \qquad\qquad \frac{\vdash \varsigma_2 \;<:\; \varsigma_1}{\vdash C\langle -\varsigma_1\rangle \;<:\; C\langle -\varsigma_2\rangle} \quad (\text{SJ-Class}^{-})$$

$$\frac{C\langle \pm_1\rangle <:: C'\langle \pm_1'\varsigma_1'\rangle \qquad \vdash [\alpha \mapsto {}^{+}/_{\pm_1}\varsigma_1]C'\langle \pm_1'\varsigma_1'\rangle \;<:\; D\langle \pm_2\varsigma_2\rangle}{\vdash C\langle \pm_1\varsigma_1\rangle \;<:\; D\langle \pm_2\varsigma_2\rangle}$$

Figure 7.4: Java-like Subtyping

$$\begin{array}{lcl}
Wyv(\top) & = & \top \\
Wyv(\bot) & = & \bot \\
Wyv(\alpha_C) & = & \alpha_C.L \\
Wyv(C\langle \pm\varsigma\rangle) & = & x.C\{Wyv(L_C, \pm\varsigma)\} \\
 & & \\
Wyv(L, +\varsigma) & = & L \;\leqslant Wyv(\varsigma) \\
Wyv(L, -\varsigma) & = & L \;\geqslant Wyv(\varsigma) \\
 & & \\
Wyv(\emptyset) & = & \emptyset \\
Wyv(C\langle \pm\varsigma_1\rangle \;<::\; D\langle \pm\varsigma_2\rangle; CT) & = & C \;\preceq\; D\{\alpha_C \Rightarrow Wyv(L, \pm\varsigma_1), Wyv(L, \pm\varsigma_2)\}; Wyv(CT) \\
Wyv(\alpha_C \mapsto \pm_1\varsigma_1; \Gamma) & = & \alpha_C : \{\alpha_C \Rightarrow Wyv(L, \pm_1\varsigma_1)\}, Wyv(\Gamma)
\end{array}$$

Figure 7.5: Encoding Java in Wyvern

difficult goal for DOT: the potential for bad bounds. As with DOT, this does not necessarily mean that typing is is unsafe. While types with ill-formed bounds can be defined in Wyv_{core}, they are implicitly uninhabited as their definition must be satisfied by a member of some real object. The type safety argument for Wyv_{core} leans on that of DOT by demonstrating Wyv_{core} a subset of DOT rather taking the traditional route of Progress and Preservation. In this section I define the term typing and reduction of Wyv_{core}. I then define an encoding of Wyv_{core} into DOT. I subsequently prove that a well typed Wyv_{core} term is in fact a well-typed DOT term.

$$\begin{array}{lllr}
t & ::= & & \textbf{Term} \\
 & x & & \textit{variable} \\
 & \texttt{new}\ \tau\{z \Rightarrow \overline{d}\} & & \textit{object} \\
 & t.m(t) & & \textit{method call} \\
x & ::= & & \textbf{variable} \\
 & y & & \textit{concrete var} \\
 & z & & \textit{abstract var}
\end{array}
\qquad
\begin{array}{lllr}
d & ::= & & \textbf{Declaration} \\
 & L \ = \ \tau & & \textit{type} \\
 & m : \forall(x : \tau).\tau = t & & \textit{method} \\
v & ::= & & \textbf{value} \\
 & y & &
\end{array}$$

Figure 7.6: Wyv_{core} Term Syntax

7.4.1 Term Typing and Operational Semantics

I define the term syntax, typing and reduction for Wyv_{core}. As Wyv_{μ} is a subset of Wyv_{core}, results based on this typing are applicable to Wyv_{μ} too. The term syntax is defined in Figure 7.6. A **Term** is either variable (x), a new object initialisation ($\texttt{new}\ \tau\{z \Rightarrow \overline{d}\}$), or a method invocation ($t.m(t)$). New objects contain a set of member declarations. A declaration d is either a type declaration ($L \ = \ \tau$), or a method declaration ($m : \forall(z : \tau).\tau \ = \ t$).

The syntax of Figure 7.6 is only superficially different from that of DOT 2016. The similarity in syntax between the two is intentionally adopted to aid in the encoding of Wyv_{core} to DOT. DOT 2016 is a purely object oriented language, and does not contain the anonymous dependent functions of Wadlerfest DOT. Method declarations in DOT 2016 are however dependent functions, allowing DOT 2016 to capture the same functionality by wrapping functions in objects. In designing the term syntax for Wyv_{core}, we restrict the type of all members to be dependent function types in order to mirror DOT 2016.

While the syntax is fairly similar to that of DOT, term typing is significantly different. The term typing is defined in Figure 7.7. This is a more complete term typing that subsumes the typing of Figure 3.10, extending

$$\boxed{\Gamma \vdash t \ : \ \tau, \ d \ : \ \sigma}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \ : \ \tau} \ (\text{T-}\textsc{Var}) \qquad \frac{\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\} \qquad \sigma \in \overline{\sigma}}{\Gamma \vdash x \ : \ \{[x/z]\sigma\}} \ (\text{T-}\textsc{Rec})$$

$$\frac{\Gamma \vdash t \ : \ \tau\{\overline{\sigma}\} \qquad \sigma \in \overline{\sigma}}{\Gamma \vdash t \ : \ \{\sigma\}} \ (\text{T-}\textsc{Non-Rec}) \qquad \frac{\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\}}{\Gamma \vdash x \ : \ \tau} \ (\text{T-}\textsc{Rfn}_1)$$

$$\frac{\Gamma \vdash t \ : \ \tau\{\overline{\sigma}\}}{\Gamma \vdash t \ : \ \tau} \ (\text{T-}\textsc{Rfn}_2) \qquad \frac{\Gamma \vdash t \ : \ x.L \qquad \Gamma \vdash x \ : \ \{L \ \leqslant \tau\}}{\Gamma \vdash t \ : \ \tau} \ (\text{T-}\textsc{Sel})$$

$$\frac{\Gamma \vdash t \ : \ \{L \preceq/= \tau\}}{\Gamma \vdash t \ : \ \{L \ \leqslant \tau\}} \ (\text{T-}\textsc{Upper}) \qquad \frac{\Gamma \vdash t \ : \ \{L \ = \ \tau\}}{\Gamma \vdash t \ : \ \{L \ \geqslant \tau\}} \ (\text{T-}\textsc{Lower})$$

$$\frac{\Gamma \vdash t_0 \ : \ \{m : \forall(z : \tau_1).\tau_2\} \qquad \Gamma \vdash y \ : \ \tau \qquad \Gamma \vdash \tau \ <: \ \tau_1}{\Gamma \vdash t_0.m(y) \ : \ [y/z]\tau_2} \ (\text{T-}\textsc{Invk}_1)$$

$$\frac{\Gamma \vdash t_0 \ : \ \{m : \forall(z : \tau').\tau\} \qquad \Gamma \vdash t \ : \ \tau_2 \qquad \Gamma \vdash \tau_2 \ <: \ \tau' \qquad x \ \notin \ fv(\tau)}{\Gamma \vdash t_0.m(t) \ : \ \tau} \ (\text{T-}\textsc{Invk}_2)$$

$$\frac{\Gamma \vdash \tau \ \cong \ \{z \Rightarrow \overline{\sigma}\} \qquad \begin{array}{c} \overline{d} \ \textit{has distinct labels} \\ \Gamma, z : \tau \vdash \overline{d} \ : \ \overline{\sigma}' \end{array} \qquad \Gamma, z : \tau \vdash \overline{\sigma}' \ <: \ \overline{\sigma}}{\Gamma \vdash \textbf{new} \ \tau\{z \Rightarrow \overline{d}\} : \ \tau} \ (\text{T-}\textsc{New})$$

$$\Gamma \vdash L \ = \ \tau \ : \ L \ = \ \tau \quad (\text{T-}\textsc{Type})$$

$$\frac{\Gamma, x : \tau_1 \vdash t \ : \ \tau_2}{\Gamma \vdash (m : \forall(x : \tau_1).\tau_2 \ = \ t) \ : \ (m : \forall(: \tau_1).\tau_2)} \ (\text{T-}\textsc{Meth})$$

Figure 7.7: Wyv_{core} Term Typing

$$\Gamma \vdash \top \ \cong \ \{z \Rightarrow \emptyset\} \qquad \frac{\Gamma \vdash \tau \ \cong \ \tau'}{\Gamma \vdash \tau\{z \Rightarrow \overline{\sigma}\} \cong \mathit{flat}(\tau', \overline{\sigma}, z)} \qquad \frac{\begin{array}{c} \Gamma \vdash x \ : \ \{L \preceq/= \tau\} \\ \Gamma \vdash \tau \ \cong \ \tau' \end{array}}{\Gamma \vdash x.L \cong \tau'}$$

Figure 7.8: Wyv_{core} Member Expansion

those rules with type rules for the terms of Figure 7.6, and generalizing some of the existing rules for all terms rather than just variables. Thus, typing for variables (T-Var) and unfolding of recursive types for variables (T-Rec) remain unchanged. Type rules for type refinements (T-Rfn), selection types (T-Sel), and limited subsumption for upper and lower bounded types (T-Upper and T-Lower) are generalized for all terms t rather than just variables (x). We extend these rules with type rules for method invocation (T-Invk$_1$ and T-Invk$_2$), and new objects (T-New). A method invocation $t_0.m(y)$ has type $[y/z]\tau_2$ if the receiver t_0 has type $\{m : \forall(z : \tau_1).\tau_2\}$, and the argument y is has a type that subtypes τ_1. A new object initialisation **new** $\tau\{z \Rightarrow \overline{d}\}$ has type τ if τ if the types of object members $\overline{d}$ subtype the expansion of τ ($\cong$). Expansion of a type is defined in Figure 7.8, and is necessary in Wyv_{core} due to the added nominality of type refinements and nominal type definitions when compared to a calculus such as DOT. Expansion extracts the member types of an object type. $\top$ expands into an empty refinement on $\top$, type refinements expand into the expansion of the base type flattened with the refinement. Only nominal or exact selections are expandable, and expand to the expansion of their defined type.

I now define a term reduction, also derived from DOT 2016, in Figure 7.10. Evaluation contexts and location stores, used in reduction, are defined in Figure 7.9 and are standard and uninteresting. A method invocation (R-Invk) reduces to the body of the method retrieved from the store, substituting out the self and method variables for the receiver and argument respectively. A new object initialization reduces to a newly created location in the store that contains the object. Context reduction (R-Ctx) is standard.

7.4.2 Encoding Wyv_{core} in DOT 2016

As I have already mentioned, the argument for type safety of Wyv_{core} is derived from that of DOT 2016. Toward this end I define a mapping in Figure 7.11 from Wyv_{core} types and terms to DOT 2016 types and terms.

$$
\begin{array}{lll}
E & ::= & \textbf{Eval. Context} \\
 & [\,] & \textit{hole} \\
 & E.m(t) & \\
 & y.m(E) &
\end{array}
\qquad
\begin{array}{lll}
\gamma & ::= & \textbf{Store} \\
 & \emptyset & \\
 & y \mapsto \textbf{new}\ \tau\{z \Rightarrow \overline{d}\}, \gamma &
\end{array}
$$

Figure 7.9: Wyv_{core} Evaluation Contexts and Store

$$
\frac{\gamma(y) = \textbf{new}\ \tau\{z \Rightarrow \overline{d}\} \qquad (m : \forall(x : \tau_1).\tau_2 \ = \ t) \ \in \ \overline{d}}{y.m(y') \longrightarrow \gamma \mid [y/z][y'/x]t} \quad (\textsc{R-Invk})
$$

$$
\frac{y \textit{ fresh in } \gamma \qquad \gamma' = \gamma[y \ \mapsto \ \textbf{new}\ \tau\{x \Rightarrow \overline{d}\}]}{\gamma \mid \textbf{new}\ \tau\{z \Rightarrow \overline{d}\} \longrightarrow \gamma' \mid y} \quad (\textsc{R-New})
$$

$$
\frac{t \longrightarrow t'}{E[t] \longrightarrow E[t']} \quad (\textsc{R-Ctx})
$$

Figure 7.10: Wyv_{core} Operational Semantics

The subsequent type safety argument is constructed by demonstrating that this mapping preserves subtyping, typing and reduction. The primary differences between the two syntaxes is the manner in which multiple member declarations and member declaration types are captured. In Wyv_{core}, multiple member declaration types are represented as a set ($\overline{\sigma}$) in type refinements. Similarly, a set of member declarations ($\overline{d}$) is included as part of a new object. DOT 2016 makes use of intersections of declaration types to capture multiple declarations, and intersections of member declarations to capture multiple member declarations. These differences are visible in Figure 7.11. Most encodings are relatively straight forward and uninteresting, however the divergence between the two calculi is evident in the encodings of type refinements ($\mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\})$), new object initializations ($\mathcal{D}(\textbf{new}\ \tau\{z \Rightarrow \overline{d}\})$)

$$
\begin{array}{lcl}
\mathcal{D}(\top) & = & \top \\
\mathcal{D}(\bot) & = & \bot \\
\mathcal{D}(x) & = & x \\
\mathcal{D}(x.L) & = & x.L \\
\mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\}) & = & \{z \Rightarrow \mathcal{D}(\tau) \wedge (\bigwedge\limits_{\sigma\in\overline{\sigma}} \mathcal{D}(\sigma)^z)\} \\
 & & \\
\mathcal{D}(\tau\{\overline{\sigma}\}) & = & \mathcal{D}(\tau) \wedge (\bigwedge\limits_{\sigma\in\overline{\sigma}} \mathcal{D}(\sigma)) \\
 & & \\
\mathcal{D}(L \ \leqslant \tau) & = & L \ : \bot \ldots \mathcal{D}(\tau) \\
\mathcal{D}(L \ \geqslant \tau) & = & L \ : \mathcal{D}(\tau) \ldots \top \\
\mathcal{D}(L \ = \ \tau) & = & L \ : \mathcal{D}(\tau) \ldots \mathcal{D}(\tau) \\
\mathcal{D}(L \ \preceq \ \tau) & = & L \ : \mathcal{D}(\tau) \ldots \mathcal{D}(\tau) \\
\mathcal{D}(m : \forall(x : \tau_1).\tau_2) & = & m(x : \mathcal{D}(\tau_1)) \ : \ \mathcal{D}(\tau_2) \\
 & & \\
\mathcal{D}(x) & = & x \\
\mathcal{D}(x\ y) & = & x\ y \\
\mathcal{D}(x.l) & = & x.l \\
\mathcal{D}(\textbf{new}\ \tau\{z \Rightarrow \overline{d}\}) & = & \{z \Rightarrow \bigwedge\limits_{d\in\overline{d}} \mathcal{D}(d)\} \\
 & & \\
\mathcal{D}(L \ = \ \tau) & = & L \ = \ \mathcal{D}(\tau) \\
\mathcal{D}(m : \forall(z : \tau_1).\tau_2 \ = \ t) & = & m(x) \ = \ \mathcal{D}(t)
\end{array}
$$

Figure 7.11: Wyv_{core} to DOT 2016 Encoding

and method members ($\mathcal{D}(m : \forall(x : \tau_1).\tau_2 \ = \ t)$).

Type safety of Wyv_{core} is constructed in five theorems:

1. Theorem 7.4.1 proves that variable typing in Wyv_{core} implies variable typing in DOT.

2. Theorem 7.4.2 proves that subtyping in Wyv_{core} implies an equivalent subypting in DOT.

3. Theorem 7.4.3 proves that typing in Wyv_{core} implies an equivalent typing in DOT.

4. Theorem 7.4.4 proves that term reduction in Wyv_{core} implies an equivalent term reduction in DOT.

5. Theorem 7.4.5 proves that term reduction in Wyv_{core} does not get stuck.

Theorem 7.4.1 (Wyv_{core} variable typing implies DOT 2016 variable typing)**.** *For all* Γ, x, τ_x, *if* $\Gamma \vdash x \ : \ \tau_x$ *then* $\mathcal{D}(\Gamma) \vdash x \ : \ \mathcal{D}(\tau_x)$.

Proof. We construct the proof by mutual induction on the derivation of $\Gamma \vdash x \ : \ \tau_x$.

Case 1 (T-VAR)**.**

$$\Gamma(x) = \tau_x$$

It follows that

$$\mathcal{D}(\Gamma)(x) = \mathcal{D}(\tau_x)$$

and thus, $\mathcal{D}(\Gamma) \vdash x \ : \ \mathcal{D}(\tau_x)$ (by VAR [75]).

Case 2 (T-REC)**.**

$$\Gamma \vdash x \ : \ \tau\{z \Rightarrow \overline{\sigma}\} \qquad \sigma \in \overline{\sigma} \qquad \tau_x = \{[x/z]\sigma\}$$

By the induction hypothesis, we have

$$\mathcal{D}(\Gamma) \vdash x : \ \mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\})$$

By the definition of $\mathcal{D}$ we have

$$\begin{array}{ll}
\mathcal{D}(\Gamma) \vdash x \ : \ \{z \Rightarrow \mathcal{D}(\tau) \wedge \bigwedge\limits_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z\} & \text{(by defn. of } \mathcal{D}) \\
\mathcal{D}(\Gamma) \vdash x \ : \ \mathcal{D}(\tau) \wedge \bigwedge\limits_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^x & \text{(by R}\textsc{ec}\text{-E)} \\
\mathcal{D}(\Gamma) \vdash x \ : \ \{\mathcal{D}(\sigma)\}^x & \text{(by S}\textsc{ub}\text{, A}\textsc{nd}_1\text{1 and A}\textsc{nd}_1\text{2)} \\
\mathcal{D}(\Gamma) \vdash x \ : \ \{\mathcal{D}([x/z]\sigma)\} & \text{(by defn. of } \mathcal{D})
\end{array}$$

The desired result.

Case 3 (T-NON-REC)**.** By similar reasoning to Case 2, without the unfolding of the recursive type.

Case 4 (T-Rfn)**.**

$$\Gamma \vdash x \;:\; \tau\{z \Rightarrow \overline{\sigma}\} \qquad \tau_x = \tau$$

By the induction hypothesis, we have

$$\mathcal{D}(\Gamma) \vdash x : \; \mathcal{D}(\tau\{z \Rightarrow \overline{\sigma}\})$$

By the definition of $\mathcal{D}$ we have

$$\begin{array}{lll} \mathcal{D}(\Gamma) \vdash x \;:\; \{z \Rightarrow \mathcal{D}(\tau) \wedge \bigwedge\limits_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z\} & & \text{(by defn. of } \mathcal{D}\text{)} \\ \mathcal{D}(\Gamma) \vdash x \;:\; \mathcal{D}(\tau) \wedge \bigwedge\limits_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^x & & \text{(by VarUnpack)} \\ \mathcal{D}(\Gamma) \vdash x \;:\; \{\mathcal{D}(\tau)\} & & \text{(by Sub and And}_{11}\text{)} \end{array}$$

The desired result.

Case 5 (T-Rfn$_2$)**.** By similar reasoning to Case 4, without the unfolding of the recursive type.

Case 6 (T-Sel)**.**

$$\Gamma \vdash x \;:\; y.L \qquad \Gamma \vdash y \;:\; \{L \;\leqslant \tau_x\}$$

By the induction hypothesis and the definition of $\mathcal{D}$ we have

$$\mathcal{D}(\Gamma) \vdash x \;:\; y.L \qquad \mathcal{D}(\Gamma) \vdash y \;:\; \{L \;: \bot \ldots \mathcal{D}(\tau_x)\}$$

By Sel$_1$, $\mathcal{D}(\Gamma) \vdash y.L \;<:\; \mathcal{D}(\tau_x)$, and then by Sub we get the desired result: $\mathcal{D}(\Gamma) \vdash \; x \;:\; \mathcal{D}(\tau_x)$.

Case 7 (T-Upper)**.**

$$\Gamma \vdash x \;:\; \{L \preceq\!/\!= \tau\} \qquad \tau_x = \{L \;\leqslant \tau\}$$

By the induction hypothesis:

$$\mathcal{D}(\Gamma) \vdash x \;:\; \{L \;: \mathcal{D}(\tau) \ldots \mathcal{D}(\tau)\}$$

By DTyp-<:-DTyp and Sub we have the desired result:

$$\mathcal{D}(\Gamma) \vdash x \;:\; \{L \;: \bot \ldots \mathcal{D}(\tau)\}$$

Case 8 (T-Lower)**.** By similar reasoning to T-Upper we get the desired result.

□

Before proving subtyping is preserved by $\mathcal{D}$, I first demonstrate that that a type's extension (Figure 3.11) is a super type of the original type.

Lemma 7.4.1 ($\leqslant{::} \Rightarrow <:$ in DOT 2016)**.** *For all* Γ*,* τ *and* τ' *such that* $\Gamma \vdash \tau \ \leqslant{::}\ \tau'$ *then* $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau) \ <:\ \mathcal{D}(\tau')$*.*

Proof. By induction on the derivation of $\Gamma \vdash \tau \ \leqslant{::}\ \tau'$:

Case 1 (E-Sel)**.**

$$\tau = x.L \qquad \Gamma \vdash x \ :\ \{L \ \leqslant \tau'\}$$

By Theorem 7.4.1 and the definition of $\mathcal{D}$, we get

$$\mathcal{D}(\Gamma) \vdash x \ :\ \{L \ : \bot \ldots \mathcal{D}(\tau')\}$$

The desired result is then achieved by $\textsc{Sel}_1$

Case 2 (E-Rfn)**.**

$$\tau = \tau_1\{z \Rightarrow \overline{\sigma}\}$$
$$\Gamma \vdash \tau_1 \ \leqslant{::}\ \tau_2 \qquad \tau' = flat(\tau_2, \overline{\sigma}, z) \qquad \mathcal{D}(\tau) = \{z \Rightarrow \mathcal{D}(\tau_1) \wedge \bigwedge_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z\}$$

By case analysis on the $flat$ function, it is easy to demonstrate that

$$\mathcal{D}(flat(\tau_2, \overline{\sigma}, z)) = \{z \Rightarrow \mathcal{D}(\tau_2)^z \wedge \bigwedge_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z\}$$

By the induction hypothesis we get:

$$\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) \ <:\ \mathcal{D}(\tau_2)$$

Thus we get:

$$\begin{array}{ll}
\mathcal{D}(\Gamma), z : \mathcal{D}(\tau) \vdash \mathcal{D}(\tau_1) \ <: \ \mathcal{D}(\tau_2) & \text{(by weakening)} \\
\mathcal{D}(\Gamma), z : \mathcal{D}(\tau) \vdash \mathcal{D}(\tau_1) \wedge \bigwedge_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z \ <: \ \mathcal{D}(\tau_2) \wedge \bigwedge_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z & \text{(by Reflexivity} \\
 & \text{and } \textsc{And}_{11}) \\
\mathcal{D}(\Gamma) \vdash \{z \Rightarrow \mathcal{D}(\tau_1) \wedge \bigwedge_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z\} \ <: \ \{z \Rightarrow \mathcal{D}(\tau_2) \wedge \bigwedge_{\sigma \in \overline{\sigma}} \mathcal{D}(\sigma)^z\} & \text{(by } \textsc{BindX})
\end{array}$$

The desired result. □

Theorem 7.4.2 (DOT 2016 subsumes Wyv_{core} subtyping)**.** *For all* Γ*,* τ_1 *and* τ_2*, if* $\Gamma \vdash \tau_1 \ <: \ \tau_2$ *then* $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau_1) \ <: \ \mathcal{D}(\tau_2)$*.*

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash \tau_1 \ <: \ \tau_2$.

Case 1 (S-Rfl)**.** Trivial.

Case 2 (S-Bot)**.** Trivial.

Case 3 (S-Top)**.** Trivial.

Case 4 (S-Upper)**.**

$$\tau_1 = x.L \qquad \Gamma \vdash x \ : \ \{L \ \leqslant \tau\} \qquad \Gamma \vdash \tau \ <: \ \tau_2$$

By Theorem 7.4.1 it follows that $\mathcal{D}(\Gamma) \vdash x \ : \ \{L \ : \_ \ldots \mathcal{D}(\tau)\}$. The induction hypothesis gives $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau) \ <: \ \mathcal{D}(\tau_2)$. By DOT 2016 subtyping ($\textsc{Sel}_1$ and Trans) we get the desired result :$\mathcal{D}(\Gamma) \vdash x.L \ <: \ \mathcal{D}(\tau_2)$

Case 5 (S-Lower)**.** By similar reasoning to Case 4.

Case 6 (S-All)**.** $\mathcal{D}$ does not encode dependent function types, thus $\mathcal{D}(\tau)$ does not exist, and the result is trivial.

Case 7 (S-Refine)**.**

$$\begin{array}{ccc}
\tau_1 = \tau\{z \Rightarrow \overline{\sigma}_1\} & \tau_2 = \tau\{z \Rightarrow \overline{\sigma}_2\} & \Gamma, z : \tau\{z \Rightarrow \overline{\sigma}_1\} \vdash \overline{\sigma}_1 \ <: \ \overline{\sigma}_2
\end{array}$$
$$\mathcal{D}(\tau_1) = \{z \Rightarrow \mathcal{D}(\tau) \wedge \bigwedge_{\sigma \in \overline{\sigma}_1} \mathcal{D}(\sigma)^z\} \qquad \mathcal{D}(\tau_2) = \{z \Rightarrow \mathcal{D}(\tau) \wedge \bigwedge_{\sigma \in \overline{\sigma}_2} \mathcal{D}(\sigma)^z\}$$

By the S-Decls, it follows that

$$\forall \sigma_2 \in \overline{\sigma}_2,\ \exists \sigma_1,\ \textit{such that}\ \Gamma, \tau_1 \vdash \sigma_1 \ <:\ \sigma_2$$

and subsequently by the induction hypothesis and the definition of $\mathcal{D}$:

$$\mathcal{D}(\Gamma), \mathcal{D}(\tau_1) \vdash \mathcal{D}(\sigma_1) \ <:\ \mathcal{D}(\sigma_1)$$

It then follows that for every T_2 in $\bigwedge\limits_{\sigma \in \overline{\sigma}_2} \mathcal{D}(\sigma)^z$, there exists some T_1 in $\bigwedge\limits_{\sigma \in \overline{\sigma}_1} \mathcal{D}(\sigma)^z$ such that

$$\mathcal{D}(\Gamma), \mathcal{D}(\tau_1) \vdash T_1 \ <:\ T_2$$

Since by reflexivity of subtyping in DOT 2016 we have: $\mathcal{D}(\Gamma), \mathcal{D}(\tau_1) \vdash \mathcal{D}(\tau) \ <:\ \mathcal{D}(\tau)$, the desired result can be reached by a combination of And$_{11}$, And$_{12}$ and And$_2$.

Case 8 (S-Extend)**.**

$$\frac{\Gamma \vdash \tau_1 \ \leqslant::\ \tau \qquad \Gamma \vdash \tau \ <:\ \tau_2}{\Gamma \vdash \tau_1 \ <:\ \tau_2}$$

The result follows from a combination of Lemma 7.4.1 and subtype transitivity (Trans) in DOT 2016.

□

Lemma 7.4.2. *For all* Γ*,* τ *and* τ' *such that* $\Gamma \vdash \tau \ \cong\ \tau'$*, then* $\mathcal{D}(\Gamma) \vdash \mathcal{D}(\tau') \ <:\ \mathcal{D}(\tau)$*.*

Proof. By induction on the derivation of $\Gamma \vdash \tau \ \cong\ \tau'$:

Case 1.

$$\tau = \top \qquad \tau' = \{z \Rightarrow \emptyset\}$$

Trivial by Top.

Case 2.

$$\tau = \tau''\{z \Rightarrow \overline{\sigma}\} \qquad \Gamma \vdash \tau'' \;\cong\; \{z \Rightarrow \overline{\sigma}''\} \qquad \tau' = flat(\{z \Rightarrow \overline{\sigma}''\}, z, \overline{\sigma})$$

By the defintion of $flat$ and $\mathcal{D}$ we get

$$\begin{aligned} \mathcal{D}(\tau) &= \{z \Rightarrow \mathcal{D}(\tau'') \wedge \bigwedge_{\sigma_i \in \overline{\sigma}} \mathcal{D}(\sigma_i)^z\} \\ \mathcal{D}(\tau') &= \{z \Rightarrow \top \wedge (\bigwedge_{\sigma_j \in \overline{\sigma}''} \{\sigma_j\}) \wedge (\bigwedge_{\sigma_i \in \overline{\sigma}} \{\sigma_i\})\} \end{aligned}$$

By the induction hypothesis:

$$\begin{array}{lll}
\mathcal{D}(\Gamma) \vdash & \mathcal{D}(\{z \Rightarrow \overline{\sigma}''\}) \;<:\; \mathcal{D}(\tau'') & \\
\mathcal{D}(\Gamma) \vdash & \{z \Rightarrow \top \wedge (\bigwedge_{\sigma_j \in \overline{\sigma}''} \{\mathcal{D}(\sigma_j)\})\} \;<:\; \mathcal{D}(\tau'') & \text{(by the defn. of } \mathcal{D}) \\
\mathcal{D}(\Gamma), z : \mathcal{D}(\{z \Rightarrow \overline{\sigma}''\}) \vdash & (\top \wedge (\bigwedge_{\sigma_j \in \overline{\sigma}''} \{\mathcal{D}(\sigma_j)\}))^z \;<:\; \mathcal{D}(\tau'') & \text{(by } \textsc{BindI}) \\
\mathcal{D}(\Gamma), z : \mathcal{D}(\{z \Rightarrow \overline{\sigma}''\}) \wedge (\bigwedge_{\sigma_i \in \overline{\sigma}} \{\sigma_i\}) \vdash & (\top \wedge (\bigwedge_{\sigma_j \in \overline{\sigma}''} \{\mathcal{D}(\sigma_j)\}))^z \;<:\; \mathcal{D}(\tau'') & \text{(by } \textsc{Narrowing}) \\
\mathcal{D}(\Gamma), z : \mathcal{D}(\{z \Rightarrow \overline{\sigma}''\}) \wedge (\bigwedge_{\sigma_i \in \overline{\sigma}} \{\sigma_i\}) \vdash & (\top \wedge (\bigwedge_{\sigma_j \in \overline{\sigma}''} \{\mathcal{D}(\sigma_j)\}))^z \wedge (\bigwedge_{\sigma_i \in \overline{\sigma}} \{\sigma_i\}) & \text{(by } \textsc{And}_{11}, \\
 & <:\; \mathcal{D}(\tau'') \wedge (\bigwedge_{\sigma_i \in \overline{\sigma}} \{\sigma_i\}) & \textsc{And}_{12} \text{ and } \textsc{And}_{2})
\end{array}$$

Finally, the result is derived through BindX.

□

Theorem 7.4.3 (DOT subsumes Wyv_{core} typing)**.** *For all* Γ*,* t *and* τ*, if* $\Gamma \vdash t \;:\; \tau$ *then* $\mathcal{D}(\Gamma) \vdash \mathcal{D}(t) \;:\; \mathcal{D}(\tau)$

Proof. This is the more general form of Theorem 7.4.1. By induction on the derivation of $\Gamma \vdash t \;:\; \tau$: For cases T-Var and T-Rec, the desired result can be demonstrated by Theorem 7.4.1. Most of the remaining cases are solved by either similar reasoning to Theorem 7.4.1, or direct application of either the induction hypothesis or Theorem 7.4.2. The only case that varies much from the type rule for DOT 2016 is the rule for new object initialization:

Case 1 (T-New)**.**

$$\begin{gathered} \Gamma \vdash \tau \;\cong\; \{z \Rightarrow \overline{\sigma}\} \\ \Gamma, z : \{z \Rightarrow \overline{\sigma}'\} \vdash \overline{d} \;:\; \overline{\sigma}' \qquad \Gamma, z : \{z \Rightarrow \overline{\sigma}'\} \vdash \overline{\sigma}' \;<:\; \overline{\sigma} \end{gathered}$$

By the induction hypothesis we have

$$\mathcal{D}(\Gamma, z : \{z \Rightarrow \overline{\sigma}'\}) \vdash \mathcal{D}(\overline{d}) \ : \ \mathcal{D}(\overline{\sigma}')$$

By the Theorem 7.4.2 we have

$$\mathcal{D}(\Gamma, z : \{z \Rightarrow \overline{\sigma}'\}) \vdash \mathcal{D}(\overline{\sigma}') \ \ <: \ \mathcal{D}(\overline{\sigma})$$

By the Theorem 7.4.2 we have

$$\mathcal{D}(\Gamma) \vdash \mathcal{D}(\textbf{new}\ \tau\{z \Rightarrow \overline{d}\}) : \ \mathcal{D}(\tau)$$

Thus by subsumption, we get the desired result:

□

Theorem 7.4.4 (DOT Term Reduction is equivalent to *Wyv*$_{core}$ Term Reduction)**.** *For all t and t', if $t \longrightarrow t'$ then $\mathcal{D}(t) \longrightarrow \mathcal{D}(t')$.*

Proof. Term reduction is unaffected by either subtyping or typing, the major differences between *Wyv*$_{core}$ and DOT 2016. Reduction is in fact equivalent, and the proof derives easily from this. □

Theorem 7.4.5 (*Wyv*$_{core}$ is Type Safe)**.** *For all t and τ, if $\emptyset \vdash t \ : \ \tau$, then term reduction of t does not get stuck.*

Proof. The result follows directly from Theorems 7.4.3 and 7.4.4. □

Chapter 8

Conclusion

In this thesis I have identified the different sources of undecidability in typing languages with both path dependent and recursive types. I have subsequently designed three different approaches to tackling these problems. In this Chapter I summarize the contributions of this thesis. I subsequently go on to describe potential future work, and questions that remain unanswered.

8.1 Contributions of this Thesis

In this section I detail the contributions of this thesis. The contributions fall in to two categories: (i) the extension to nominality in Wyvern, and (ii) the development of three distinct approaches to achieving decidability.

8.1.1 Nominality in Wyvern

The construction of the Material/Shape separation of Chapter 3 relies on the nominal properties of Wyv_{core}. This is not surprising as the original Material/Shape separation of Java is based on the nominality of Java. What is interesting, is that Wyv_{core} (and in fact DOT) does not possess the full nominality of Java. There are two important ways nominality in Wyv_{core} differs from that of Java:

1. *Nominal Concrete Types*: In Java all subtyping is nominal, even subtyping between concrete types. In Wyv_{core} (and DOT), nominal subtyping only exists as type refinements on abstractly defined types.

2. *Nominal Supertyping*: In Java the nominal type hierarchy defines not only which types may subtype a specific type, but also which types may supertype a type. Abstract type members in Wyv_{core} do not provide this second restriction. A type $x.L$ defined using the form $L \leqslant \tau$ may only be subtyped by types that refine $x.L$, whereas, $x.L$ may be super-typed by any type that super-types τ.

Both of these differences in nominality have implications for the way that a Material/Shape separation can be defined for Wyv_{core}. The lack of a concrete nominal form limits the expressiveness of Material/Shape separated Wyv_{core}, and the lack of strict nominality on supertyping of Shapes subverts the restrictions enforced by the Material/Shape separation. I describe both of these issues at length in Chapter 4.

To address these issues, I define two extensions to Wyv_{core} that facilitate the Material/Shape separation:

1. I define a concrete nominal form for type members ($L \preceq \tau$) that exhibits nominality in subtyping like abstract, upper bounded type members, but is concrete and invariant in the way specific type members are.

2. I limit type extension of Shapes so as to ensure that Shapes may not be structurally supertyped.

8.1.2 Decidable Variants of Wyv_{core}

In this thesis I have developed three decidable variants on Wyv_{core}.

Wyv_{fix}

Wyv_{fix} uses two environments for subtyping, and allows for the simplest proof of decidability using the algorithm described in Chapter 5. Further, there is no syntactic restriction on Wyv_{fix} beyond the Material/Shape separation. There are considerable downsides however:

- Transitivity is lost, not only in the ways that one might expect from type systems with path dependent types and environment narrowing, but in cases that would seem to work (see 5.5.3).
- Type Safety there is no proof of type safety for Wyv_{fix}. It seems possible (perhaps even likely) that the language would be type safe, but the existence of two environments complicates the derivation of such a proof. Type safety for other variants was derived by demonstrating a sound encoding existed to some form of DOT. Such an encoding is more difficult to demonstrate for Wyv_{fix}.

$Wyv_{non\text{-}\mu}$

$Wyv_{non\text{-}\mu}$ does not include subtyping for recursive types, and subsequently removes a critical source of environment narrowing during subtyping. $Wyv_{non\text{-}\mu}$ is comparable to both Wadlerfest DOT and Scala in its absence of subtyping of recursive types, and so does not represent too large of a loss in expressiveness from Wyv_{core}. I provide a proof of type safety for $Wyv_{non\text{-}\mu}$ through an encoding of $Wyv_{non\text{-}\mu}$ to Wadlerfest DOT that is both sound and complete.

Wyv_{μ}

Wyv_{μ} prohibits recursive types from lower bounds, ensuring a maximal depth after which environment narrowing does not occur. I prove Wyv_{μ} type safe by defining an encoding to DOT 2016 and proving that encoding to be sound and complete. There are some drawbacks in expressiveness, particularly around the encoding of family polymorphism, but they can be mitigated by using the nominal form for type members introduced in Section 4.1.2 (Figure 4.2).

8.2 Future Work

There are still many interesting avenues for research in decidability for subtyping with path dependent types. In this section I will discuss some of these areas that might be fruitful for extending Wyvern with added expressiveness.

8.2.1 Type Safety for Wyv_{fix}

While subyping in Wyv_{fix} is decidable, it is not clear that typing for an acceptable term syntax and operational semantics would be sound. The presence of two environments in subtyping complicates any attempt to define a sound and complete encoding to either of the main DOT variants (Wadlerfest DOT or DOT 2016), the proof strategy used to demonstrate type safety for both $Wyv_{non\text{-}\mu}$ and Wyv_{μ}.

One potential strategy might take inspiration from the type safety proof of Wadlerfest DOT and DOT 2016, and defined some super set of Wyv_{fix}, $Wyv_{fix}\,'$ that explicitly introduces subtype transitivity as a rule. If it could be demonstrated that such an extension could be proven type safe, it would follow that Wyv_{fix} itself would be type safe too.

8.2.2 Transitivity

Unfortunately none of the variants of Wyv_{core} (or indeed Wyv_{core} itself) feature subtype transitivity. This is due to the same reason that transitivity proved so elusive in DOT, the so called “bad bound” problem (see 2.2.1). The lack of transitivity means that it is difficult to construct a proof of type safety for any of the variants without referring to some version of DOT.

Ideally the metatheory of Wyv_{core} would be self contained within Wyv_{core}, allowing for programmers to know that every partially evaluated program of an originally well-typed program is itself well-typed.

This does not seem to be an simple target, given the nature of transitivity in DOT, and the role that it plays in type safety. As has been discussed previously (see 2.2.1), the ability to define type members with arbitrary bounds, coupled with a transitivity axiom, means that a type member represents a subtype assumption, that introduces new connections in the subtype lattice. In other words, given the variable $x : \{L : \texttt{Int} \ldots \texttt{String}\}$ in some context allows for the following transitive subtype relationship: $\texttt{Int} <: x.L <: \texttt{String}$. Transitivity then allows for the derivation of $\texttt{Int} <: \texttt{String}$. The type safety of DOT demonstrates that such a relationship is counter-intuitively still type safe.

Generally the difficulty with defining a transitive subtyping algorithm is the question of which middle type to use. This is especially problematic if there are infinite possible types. One way to possibly achieve transitivity in Wyv_{core} is to limit the search for the middle type to the finite set of path dependent types defined within any particular context. This might be possible to do, but is likely infeasible computationally given the potentially large number of type definitions any program of even modest size might define.

Full nominality, like that of Java might provide an solution to this problem by further constraining the subtype lattice. In a nominal language such as Java, subtyping is explicitly constructed by declarations of subtyping. In Java for example, this is done by class inheritance declarations:

```
class C<E> extends D<E>
```

`C< ... >` may only subtype either some `D< ... >`, or some super type of `D< ... >`. The subtype lattice is thus constrained by explicit subtype declarations. Such nominality in Wyv_{core} might be able to treat type member definition $L : \texttt{Int} \ldots \texttt{String}$ in the same way as nominal subtype declarations, allowing for an ad-hoc way to introduce connections to the subtype lattice.

8.2.3 Intersection and Union Types

The most noticeable features missing from Wyv_{core} when compared to a language such as DOT, are *intersection* and *union* types. Intersection and union types provide a range of useful expressiveness that has already been detailed in Section 2.2.1.

Wyv_{core} is already empowered with a constrained form of intersection types in the form of type refinements (see 3.1), but critically lacks the kinds of multiple inheritance style type hierarchies afforded by full intersection types, along with the ability to define aggregate types in an ad-hoc manner.

Wyv_{core} lacks any support for union types, a language feature that DOT 2016 does have. Wadlerfest DOT does not include union types, and as such they are not needed to encode Scala types.

In this section I discuss some possible extensions to Wyv_{core} that include either some form of intersection types or both intersection and union types.

Multiple Nominal Member Definitions

In Wyv_{core} (and DOT), a type may contain multiple member types of the same name. This is not unsound, so long as at runtime, any values that inhabit that type adheres to the specification of each of those member types. Members of objects in Wyv_{core} (and DOT) must have unique names (see the typing and operational semantics in Sections 6.3.1 and 7.4.1). The reason for this is to avoid potentially unsound types being defined. Soundness in DOT (and thus Wyv_{core}) depends on objects at runtime being sound, and members within objects being well behaved. Multiple members of the same name could violate this property. Consider the following object:

$$\texttt{val}\ x\ =\ \texttt{new}\left\{ z \Rightarrow \begin{array}{lcl} L & = & \texttt{String} \\ L & = & \texttt{Int} \\ \texttt{cast} : \forall(y : z.L).\texttt{String} & = & \lambda(y : z.L).y : \texttt{String} \end{array} \right\}$$

If the definition of x were allowed, the `cast` function could be called on a value of type `Int`, something that is clearly unsound. The unsoundness derives from the fact that one definition ($L = \texttt{Int}$) can used to type arguments supplied to `cast`, and the other definition ($L = \texttt{String}$) can be used to type the body of `cast`. Put another way, we are able to type programs using the lower bound of one definition and the upper bound of another. This is not true if nominal type definitions are used:

$$\texttt{val}\ x = \texttt{new}\left\{ z \Rightarrow \begin{array}{l} L \preceq \texttt{String} \\ L \preceq \texttt{Int} \\ \texttt{cast} : \forall(y : z.L).\texttt{String} = \lambda(y : z.L).y : \texttt{String} \end{array} \right\}$$

The above program still includes multiple definitions of L, but the nominal definition of L means that it has no lower bound, and subsequently any argument supplied to `cast` must explicitly extend $x.L$, meaning it must adhere to the specification both `Int` and `String`. Such a type definition is fairly similar to multiple inheritance style type definitions. It seems likely that a relatively simple extension to Wyv_{core} could be defined that introduced such multiple nominal type definitions in a decidable and type safe manner.

Intersection and Union Types

Full intersection and union types are already known to co-exist with both path dependent and recursive types in DOT 2016 in a type safe manner. It seems reasonable that they might be introduced on top of Wyv_{core} in a way that is both safe and decidable. Intersection types in DOT are an integral part of the type system, and are used to model a variety of type patterns, including type refinements (see 2.2.1). Type refinements are however an integral part of the Material/Shape separation defined in Chapter 3, and intersection types are not immediately amenable to the application of the same Material/Shape separation.

Likely the simplest extension does not integrate the Material/Shape separation into intersection types, defines a simple extension on top of Wyv_{core} that does not seek to replace type refinements. In Figure 8.1 I define a syntactic extension on top of Wyv_{core} that does not remove type refinements,

$$\begin{array}{lll} \tau & ::= & \textbf{Type} \\ & \vdots & \\ & \tau \cap \tau & \textit{intersection} \\ & \tau \cup \tau & \textit{union} \end{array}$$

Figure 8.1: *Wyv*$_{\textit{full}}$ Syntax Extension

$$\boxed{\Gamma \vdash x \ : \ \tau}$$

$$\frac{\Gamma \vdash x \ : \ \tau_1 \cap \tau_2}{\Gamma \vdash x \ : \ \tau_1} \ (\text{T-Int}_1)$$

$$\frac{\Gamma \vdash x \ : \ \tau_1 \cap \tau_2}{\Gamma \vdash x \ : \ \tau_2} \ (\text{T-Int}_2)$$

Figure 8.2: *Wyv*$_{\textit{full}}$ Typing Extension

$$\boxed{\Gamma \vdash \tau_1 \ <: \ \tau_2}$$

$$\frac{\Gamma \vdash \tau \ <: \ \tau_1 \qquad \Gamma \vdash \tau \ <: \ \tau_2}{\Gamma \vdash \tau \ <: \ \tau_1 \cap \tau_2} \ (\text{S-And}) \qquad \frac{\Gamma \vdash \tau_1 \ <: \ \tau}{\Gamma \vdash \tau_1 \cap \tau_2 \ <: \ \tau} \ (\text{S-And}_1)$$

$$\frac{\Gamma \vdash \tau_2 \ <: \ \tau}{\Gamma \vdash \tau_1 \cap \tau_2 \ <: \ \tau} \ (\text{S-And}_2) \qquad \frac{\Gamma \vdash \tau_1 \ <: \ \tau \qquad \Gamma \vdash \tau_2 \ <: \ \tau}{\Gamma \vdash \tau_1 \cup \tau_2 \ <: \ \tau} \ (\text{S-Or})$$

$$\frac{\Gamma \vdash \tau \ <: \ \tau_1}{\Gamma \vdash \tau \ <: \ \tau_1 \cup \tau_2} \ (\text{S-Or}_1) \qquad \frac{\Gamma \vdash \tau \ <: \ \tau_2}{\Gamma \vdash \tau \ <: \ \tau_1 \cup \tau_2} \ (\text{S-Or}_2)$$

Figure 8.3: *Wyv*$_{\textit{full}}$ Subtyping Extension

but simply adds intersection and union types. I also provide extensions to *Wyv*$_{\textit{core}}$ typing (Figure 8.2) and subtyping (Figure 8.3). Such an extension could use the same Material/Shape separation with very little modification. While I have not demonstrated decidability for any variant of *Wyv*$_{\textit{full}}$, in the same manner as *Wyv*$_{\textit{non-}\mu}$ or *Wyv*$_{\mu}$, it seems likely that either approach could be taken to achieve such a variant.

Integrated Subtyping for Intersection and Union Types

Muehlboeck and Tate [60] defined a framework for extending existing languages with intersection and union types in a decidable manner, while also providing useful, intuitive, and expressive properties such as distributivity.

Such an extension would have obvious benefits to either $Wyv_{non\text{-}\mu}$ or Wyv_{μ}. Muehlboeck and Tate do however require several properties of the subtyping of the underlying language, among them subtype transitivity, a property that is missing from all variants of Wyv_{core}.

Decidable Scala

The type systems in this thesis have wider implications than just the Wyvern programming language, they also imply a potential subset of Scala that is still reasonably expressive. Scala, as the most prominent language featuring the both recursive types and path members, is a natural target for testing the ideas in this thesis on large scale industry code bases.

Both $Wyv_{non\text{-}\mu}$ and Wyv_{μ} represent potential paths for developing a decidable variant of Scala. While $Wyv_{non\text{-}\mu}$ lacks subtyping of recursive types, so does the subtyping of both Wadlerfest DOT and Scala. Wyv_{μ}, while it includes subtyping of recursive types, it does not allow for the use of recursive types in the lower bounds of type members. For Wyv_{μ} to adequately capture Scala, researchers would have to evaluate to what degree such a limitation would have on existing Scala code bases.

Building such an extension to Scala would have other advantages, most notably it would allow for a similar survey to that which Greenman et al. conducted on the separation of Materials and Shapes. Whether Sclaa programmers using type members follow the same latent distinction as Java programmers using type parameters is still an unanswered question.

Finally, there are two potential targets for extension; the current Scala 2.XX compiler, and the new Dotty formulation of Scala [4], intended as the basis of Scala 3 [6]. Both targets have their strengths as targets:

- Scala 2 represents a relatively mature language, with a sizeable code base to serve as a corpus, thus providing better intuition for the way Scala programmers use type members over type parameters.
- Dotty is intended as the basis of Scala 3, and ultimately the future of

the Scala language. Scala 3 is yet to be released (intended for a release date of early 2020 [6]), and provides a good basis for constructing decidable subtyping for future editions of Scala. Unfortunately Dotty is not intended to be backwards compatible with all language features of Scala 2, and ultimately some existing Scala code bases will not be compatible with Scala 3.

Bibliography

[1] The Wyvern Programming Language. http://wyvernlang.github.io/. Accessed: 2019-04-21.

[2] Decidable wyvern coq sources. http://tiny.cc/ozomdz, 2019. Accessed: 2019-09-30.

[3] C# language specification. https://bit.ly/30UG9qC, 2019. Accessed: 2019-04-21.

[4] Dotty. http://dotty.epfl.ch/, 2019. Accessed: 2019-04-21.

[5] Ecmascript language specification. www.ecma-international.org/, 2019. Accessed: 2019-04-21.

[6] Towards scala 3. https://www.scala-lang.org/blog/2018/04/19/scala-3.html, 2019. Accessed: 2019-09-29.

[7] Typescript language specification. https://bit.ly/2WiID3E, 2019. Accessed: 2019-04-21.

[8] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1996. ISBN 0387947752.

[9] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 49–65, New York, NY,

USA, 1997. ACM. ISBN 0-89791-908-4. doi: 10.1145/263698.263720. URL http://doi.acm.org/10.1145/263698.263720.

[10] Nada Amin. Dependent object types. page 134, 2016. doi: 10.5075/epfl-thesis-7156. URL http://infoscience.epfl.ch/record/223518.

[11] Nada Amin and Ross Tate. Java and scala's type systems are unsound: The existential crisis of null pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 838–848, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984004. URL http://doi.acm.org/10.1145/2983990.2984004.

[12] Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types. In *19th International Workshop on Foundations of Object-Oriented Languages*, number EPFL-CONF-183030, 2012.

[13] Nada Amin, Tiark Rompf, and Martin Odersky. Foundations of path-dependent types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 233–249, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660216. URL http://doi.acm.org/10.1145/2660193.2660216.

[14] Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 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*, pages 249–272, 2016.

[15] Henk Barendregt. Introduction to generalized type systems. *Journal of functional programming*, 1(2):125–154, 1991.

[16] Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 2(2):125–154, 1991. doi: 10.1017/S0956796800020025.

[17] Yves Bertot and Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642058809, 9783642058806.

[18] Edoardo Biagioni, Robert Harper, and Peter Lee. A network protocol stack in standard ml. *Higher-Order and Symbolic Computation*, 14(4):309–356, 2001. ISSN 1573-0557. doi: 10.1023/A:1014403914699. URL http://dx.doi.org/10.1023/A:1014403914699.

[19] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. ISSN 1469-7653. doi: 10.1017/S095679681300018X. URL http://journals.cambridge.org/article_S095679681300018X.

[20] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. pages 523–549. Springer-Verlag, 1998.

[21] Nicholas Cameron, Sophia Drossopoulou, and Erik Ernst. A model for java with wildcards. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming*, pages 2–26, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70592-5.

[22] L. Cardelli, J. Donahue, M. Jordan, B. Kalsow, and G. Nelson. The modula-3 type system. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 202–212, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75295. URL http://doi.acm.org/10.1145/75277.75295.

[23] Luca Cardelli. The amber machine. In Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet, editors, *Combinators and Functional Programming Languages*, pages 48–70, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg. ISBN 978-3-540-47253-7.

[24] Luca Cardelli. Basic polymorphic typechecking. *Sci. Comput. Program.*, 8(2):147–172, April 1987. ISSN 0167-6423. doi: 10.1016/0167-6423(87) 90019-0. URL http://dx.doi.org/10.1016/0167-6423(87)90019-0.

[25] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system f with subtyping. In Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Software*, pages 750–770, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-47617-7.

[26] Felice Cardone and Mario Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48 – 80, 1991. ISSN 0890-5401. doi: https://doi.org/10.1016/0890-5401(91) 90020-3. URL http://www.sciencedirect.com/science/article/pii/0890540191900203.

[27] Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 151–162, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0. doi: 10.1145/174675.177844. URL http://doi.acm.org/10.1145/174675.177844.

[28] Giuseppe Castagna and Benjamin C. Pierce. Corrigendum: Decidable bounded quantification. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 408–, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1. doi: 10.1145/199448.199539. URL http://doi.acm.org/10.1145/199448.199539.

[29] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press, 2013. ISBN 0262026651, 9780262026659.

[30] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. ISSN 00224812. URL http://www.jstor.org/stable/2266170.

[31] Mads Torgersen Computer and Mads Torgersen. Virtual types are statically safe. In *In 5th Workshop on Foundations of Object-Oriented Languages*, 1998.

[32] William R. Cook. On understanding data abstraction, revisited. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 557–572, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640133. URL http://doi.acm.org/10.1145/1640089.1640133.

[33] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988. ISSN 0890-5401. doi: 10.1016/0890-5401(88)90005-3. URL http://dx.doi.org/10.1016/0890-5401(88)90005-3.

[34] Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11): 584, 1934.

[35] Jacques Garrigue Didier Remy Damien Doligez, Alain Frisch and Jerome Vouillon. The ocaml system release. caml.inria.fr/pub/docs/manual-ocaml/, 2019. Accessed: 2019.

[36] Erik Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 303–

326, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4. URL http://dl.acm.org/citation.cfm?id=646158.680013.

[37] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989. ISBN 0-521-37181-3.

[38] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390069X, 9780133900699.

[39] Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting f-bounded polymorphism into shape. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 89–99, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594308. URL http://doi.acm.org/10.1145/2594291.2594308.

[40] Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009871. URL http://doi.acm.org/10.1145/3009837.3009871.

[41] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2012. ISBN 1107029570, 9781107029576.

[42] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge University Press, New York, NY, USA, 1997. ISBN 0-521-46518-4.

[43] Tony Hoare. The billion dollar mistake. infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare, 2000.

[44] Jaemin Hong, Jihyeok Park, and Sukyoung Ryu. Path dependent types with path-equality. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, pages 35–39, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5836-1. doi: 10.1145/3241653.3241657. URL http://doi.acm.org/10.1145/3241653.3241657.

[45] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[46] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. *Information and Computation*, 175(1):34 – 49, 2002. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.2001.2942. URL http://www.sciencedirect.com/science/article/pii/S0890540101929426.

[47] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001. ISSN 0164-0925. doi: 10.1145/503502.503505. URL http://doi.acm.org/10.1145/503502.503505.

[48] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance, September 2006. FOOL-WOOD ’07.

[49] Gavin King. The ceylon language specification. ceylon-lang.org/documentation/1.0/spec, 2013.

[50] Bent Kristensen, Ole Madsen, Birger Møller-Pedersen, and Kristen Nygaard. The BETA programming language. *DAIMI Report Series*, 16(229), 1987. ISSN 2245-9316. doi: 10.7146/dpb.v16i229.7578. URL http://ojs.statsbiblioteket.dk/index.php/daimipb/article/view/7578.

[51] Joseph Lee, Jonathan Aldrich, Troy Shaw, and Alex Potanin. A theory of tagged objects. In *LIPIcs-Leibniz International Proceedings in Infor-*

matics, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[52] Du Li, Alex Potanin, and Jonathan Aldrich. Delegation vs inheritance for typestate analysis. 2015.

[53] Barbara Liskov. Keynote address - data abstraction and hierarchy. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages and Applications (Addendum)*, OOPSLA '87, pages 17–34, New York, NY, USA, 1987. ACM. ISBN 0-89791-266-7. doi: 10.1145/62138.62141. URL http://doi.acm.org/10.1145/62138.62141.

[54] Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding featherweight java with assignment and immutability using the coq proof assistant. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*, FTfJP '12, pages 11–19, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1272-1. doi: 10.1145/2318202.2318206. URL http://doi.acm.org/10.1145/2318202.2318206.

[55] Julian Mackay, Alex Potanin, Jonathan Aldrich, and Lindsay Groves. Decidable subtyping for path dependent types. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi: 10.1145/3371134. URL https://doi.org/10.1145/3371134.

[56] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 397–406, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. doi: 10.1145/74877.74919. URL http://doi.acm.org/10.1145/74877.74919.

[57] Simon Marlow. Haskell 2010 language report.

[58] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. *The Definition of Standard ML, Revised Edition*. MIT Press, 1997.

[59] James H. Morris, Jr. Types are not sets. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 120–124, New York, NY, USA, 1973. ACM. doi: 10.1145/512927.512938. URL http://doi.acm.org/10.1145/512927.512938.

[60] Fabian Muehlboeck and Ross Tate. Empowering union and intersection types with integrated subtyping. *Proc. ACM Program. Lang.*, 2 (OOPSLA):112:1–112:29, October 2018. ISSN 2475-1421. doi: 10.1145/3276482. URL http://doi.acm.org/10.1145/3276482.

[61] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991. ISBN 0-13-590464-1.

[62] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7.

[63] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '13, pages 9–16, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2046-7. doi: 10.1145/2489828.2489830. URL http://doi.acm.org/10.1145/2489828.2489830.

[64] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005.

ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094815. URL http://doi.acm.org/10.1145/1094811.1094815.

[65] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In Luca Cardelli, editor, *ECOOP 2003 – Object-Oriented Programming*, pages 201–224, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45070-2.

[66] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[67] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *Proceedings of the 28th European Conference on ECOOP 2014 — Object-Oriented Programming - Volume 8586*, pages 105–130, New York, NY, USA, 2014. Springer-Verlag New York, Inc. ISBN 978-3-662-44201-2. doi: 10.1007/978-3-662-44202-9_5. URL http://dx.doi.org/10.1007/978-3-662-44202-9_5.

[68] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In *Automated Deduction — CADE-16*, pages 202–206, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-48660-2.

[69] Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 305–315, New York, NY, USA, 1992. ACM. ISBN 0-89791-453-8. doi: 10.1145/143165.143228. URL http://doi.acm.org/10.1145/143165.143228.

[70] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098.

[71] Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. Software foundations. *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, 2010.

[72] Marianna Rapoport and Ondrej Lhoták. A path to DOT: formalizing fully-path-dependent types. *CoRR*, abs/1904.07298, 2019. URL http://arxiv.org/abs/1904.07298.

[73] Marianna Rapoport, Ifaz Kabir, Paul He, and Ondřej Lhoták. A simple soundness proof for dependent object types. *Proc. ACM Program. Lang.*, 1(OOPSLA):46:1–46:27, October 2017. ISSN 2475-1421. doi: 10.1145/3133870. URL http://doi.acm.org/10.1145/3133870.

[74] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 408–423, Berlin, Heidelberg, 1974. Springer-Verlag. ISBN 3-540-06859-7. URL http://dl.acm.org/citation.cfm?id=647323.721503.

[75] Tiark Rompf and Nada Amin. 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, pages 624–641, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984008. URL http://doi.acm.org/10.1145/2983990.2984008.

[76] AMR Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3):289–360, Nov 1993. ISSN 1573-0557. doi: 10.1007/BF01019462. URL https://doi.org/10.1007/BF01019462.

[77] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. The billion-dollar fix. In Giuseppe Castagna, editor, *ECOOP 2013 –*

Object-Oriented Programming, pages 205–229, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39038-8.

[78] Aaron Stump. *Verified Functional Programming in Agda*. Association for Computing Machinery and Morgan &#38; Claypool, New York, NY, USA, 2016. ISBN 978-1-97000-127-3.

[79] Kresten Krab Thorup. Genericity in java with virtual types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 444–471, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69127-3.

[80] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1):230–265, 1936. doi: 10.2307/2268810.

[81] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015. ISSN 0001-0782. doi: 10.1145/2699407. URL http://doi.acm.org/10.1145/2699407.

[82] Philip Wadler and Wen Kokke. *Programming Language Foundations in Agda*. 2019. Available at http://plfa.inf.ed.ac.uk/.

[83] Stefan Wehr and Peter Thiemann. On the decidability of subtyping with bounded existential types. In Zhenjiang Hu, editor, *Programming Languages and Systems*, pages 111–127, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10672-9.

[84] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. Javagi: Generalized interfaces for java. In Erik Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 347–372, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73589-2.

[85] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38 – 94, 1994. ISSN

0890-5401. doi: https://doi.org/10.1006/inco.1994.1093. URL http://www.sciencedirect.com/science/article/pii/S0890540184710935.

[86] Artem Yushkovskiy and Stavros Tripakis. Comparison of two theorem provers: Isabelle/hol and coq. *CoRR*, abs/1808.09701, 2018. URL http://arxiv.org/abs/1808.09701.

[87] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, flexible object-oriented generics. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 436–445, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2738008. URL http://doi.acm.org/10.1145/2737924.2738008.