Nominal Wyvern:
Employing Semantic Separation for Usability

Yu Xiang Zhu

CMU-CS-19-105

April 2019

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Jonathan Aldrich, Chair
Heather Miller
Alex Potanin, Victoria University of Wellington, NZ

Submitted in partial fulfillment of the requirements
for the degree of Master of Science.

Keywords: Nominality, Wyvern, Dependent Object Types, Subtype Decidability

Abstract

This thesis presents Nominal Wyvern, a nominal type system that emphasizes semantic separation for better usability. Nominal Wyvern is based on the dependent object types (DOT) calculus, which provides greater expressivity than traditional object-oriented languages by incorporating concepts from functional languages. Although DOT is generally perceived to be nominal due to its path-dependent types, it is still a mostly structural system and relies on the free construction of types. This can present usability issues in a subtyping-based system where the semantics of a type are as important as its syntactic structure. Nominal Wyvern overcomes this problem by semantically separating structural type/subtype definitions from ad hoc type refinements and type bound declarations. In doing so, Nominal Wyvern is also able to overcome the subtype undecidability problem of DOT by adopting a semantics-based separation between types responsible for recursive subtype definitions and types that represent concrete data. The result is a more intuitive type system that achieves nominality and decidability while maintaining the expressiveness of F-bounded polymorphism that is used in practice.

Acknowledgments

This research would not have been possible without the support from the following individuals. I would like to thank my thesis advisor Prof. Jonathan Aldrich for giving me this opportunity and providing support over the past year. I have wanted to find a research opportunity in programming languages since before I started my master's program, so I'm very grateful for the opportunity last fall to join Prof. Aldrich for an interesting independent study that eventually led to this thesis. I've learnt a lot from this experience, and, considering I was so close to doing something else entirely instead, I'm glad I was able to do what I did. Thanks to Prof. Alex Potanin and his student Julian Mackay for their trans-Pacific guidance in and out of my weekly meetings with Prof. Aldrich in the past year. Julian's earlier papers have been an especially helpful resource in bringing me up to speed on the current state of the research. Special thanks to Prof. Heather Miller for having faith in this project and joining the committee on a short notice. Finally, thanks to all three committee members for their detailed feedback on my drafts despite the tight schedule.

Contents

List of Figures

Listings

Chapter 1

Introduction

The power of abstraction is recognized as one of the greatest ideas in computer science [Vleck, 2008]. It allows humans to safely segregate pieces of large systems so they can be reasoned about individually. For programming languages, abstraction provides flexibility by allowing for code reuse and modularization. However, while functions are the agreed upon way for abstracting away specific values to focus on the procedure itself, the exact way to achieve polymorphism, the abstraction of types (a.k.a. generic programming), is less agreed upon. On the one end is pure object-oriented (OO) languages, where polymorphism exists in the form of subtyping. Types are generally monomorphic except for the ability for specific types to act like general ones. On the other end is functional languages, where parametric polymorphism allows type variables to stand in for real types, and new types can be constructed from abstract type parameters. The differences also correspond to the encouraged ways of achieving data abstraction in each paradigm. OO languages naturally use objects to encapsulate state and procedures for interaction (a.k.a. accepted messages). Functional languages utilize modules to encapsulate abstract type members and operations on them (i.e. abstract data types). Each paradigm has its own benefits: objects enjoy the flexibility of being dynamically generated and treated as first-class values, while modules allow for more efficient implementations [Cook, 2009] and more flexible type abstraction via parameterization.

Programming language researchers have long wanted to get the best of both worlds. Scala is one of the languages that sit in between the two paradigms by supporting type members in objects and bounded parametric polymorphism (or "bounded quantification"). In contrast to plain parametric polymorphism, the bounded version allows for an OO-style

restriction that restricts the instantiating type to be the subtype of some type. The benefit is one can now additionally require that the instantiating type has certain features by giving a bound on how general it can be. For example, a hash table type can now easily require its key type to be hashable by specifying the instantiating type to be a subtype of the general Hashable type that has a `hash` method.

However, the merging of the two paradigms is also the merging of nominal and structural type systems. Traditionally, OO languages are nominal in that the names of the types are significant in themselves since they are what the types are identified with in the subtype hierarchy, the backbone of OO abstractions. Two types can have the same structure internally, but having different names means they are different. In contrast, functional languages lean on the structural side, where the structure of a type is what defines the type, and the name is a mere convenience in referring to it. This difference is closely related to how abstraction is achieved in each paradigm, so a careful merging of nominality and structurality is also required to get the best of both worlds in terms of usability. The foundational type system for Scala, dependent object types, merges the two paradigms by preserving nominality only for the subtype topology. Path-dependent types are referred to by name since they, combined with type bound declarations, make up the subtype topology. However, the rest of the type system is structural since it allows the construction of new types freely. This meant new structural types can be created without ever being given any names. For large systems where subtype relations play an important role, this may lead to accidental subtyping, or implicit subtyping relations that are unclear to code readers. Heavily relying on structural types may also lead to types whose purposes are less clear and whose problems cannot be easily communicated to the user by compiler and programming tools.

Additionally, while the merging of the paradigms seem to have provided additional expressiveness, it comes at the cost of decidability. While subtyping and parametric polymorphism by themselves are well-studied and easily decidable, the combination of them is not. It is proved very early on that bounded quantification is fundamentally an undecidable problem [Pierce, 1992]. This is due to the ability to define types that subtype a type parameterized by themselves. The languages that conservatively build on top of it also suffer from the same problem. Scala, for instance, is known to have an undecidable subtyping problem [Amin et al., 2014]. Scala also is not the only language that has tried to merge the two concepts. Java and C++ are both object-oriented languages that have parametric

polymorphism added (Java Generics, C++ templates), and these systems are also shown to be problematic as well [Grigore, 2017]. Having an undecidable component in a language can pose real problems to the user. Programmers unfamiliar with the intricacies of their language's type system may end up unable to compile their "well-written" program. The compiler's failure also means no guidance on what the programmer should do to fix the problem.

This thesis presents Nominal Wyvern, a new core type system for the Wyvern programming language [Nistor et al., 2013] based on the dependent object types calculus of Scala [Odersky et al., 2003], that aims to solve the presented problems that arise from the merging of the two paradigms with a clearer semantic separation. In particular, Nominal Wyvern's nominal typing system separates the definition of a structural type from the declaration of a type member's bound so that both typing and subtyping are explicit and nominal. It also separates the structural types that are responsible for recursive definitions from the types that represent concrete data (inspired by Mackay [2019]'s adaptation of Greenman et al. [2014]'s material-shape separation idea) to curb subtype undecidability. The goal is to produce a type system that avoids the usability pitfalls mentioned above in the hope of making it easier to write safe and correct code.

The main contribution of this thesis is the design of a more usable core type system for Wyvern that achieves its usability goals via nominality and decidability. More specifically, this thesis presents 1) the design of a more thoroughly nominal system based on DOT, and 2) an adaptation of material-shape separation to this DOT-based type system.

Chapter 2 discusses in detail the earlier research in DOT, subtype decidability, and nominality that motivated this thesis. Chapter 3 presents the grammar and typing rules of Nominal Wyvern with an example, and explains how this design facilitates usability and decidability. Chapter 4 delves into the subtyping rules and gives a proof of why subtyping is decidable when material-shape separation is observed. Chapter 5 shows the expressiveness of Nominal Wyvern by presenting several examples of common programming patterns in Nominal Wyvern syntax. Chapter 6 concludes the thesis and talks about possible directions for future work.

Chapter 2

Background and Motivation

2.1 DOT and Path-Dependent Types

The dependent object types (DOT) calculus [Amin et al., 2014] was developed as a type-theoretic foundation for Scala. The key distinguishing feature of the DOT calculus is objects with type members. Traditionally, objects included only fields and methods. Modules in ML systems supported type members, but modules do not enjoy the benefits of being first-class values like objects do in object-oriented programming languages. Unifying concepts from objects and modules allows DOT to model types that are dependent on objects. Listing 2.1 presents such an example.

```
1  class Bank {b =>
2    type Card
3    def applyForCard(name: String) : b.Card
4    def payOff(c: b.Card) : Unit = {}
5  }
6
7  val chase : Bank = ...
8  val pnc : Bank = ...
9  val myCard = chase.applyForCard("freedom")
10 chase.payOff(myCard)     // OK
11 pnc.payOff(myCard)       // type mismatch
12                          // found    : chase.Card
13                          // required : pnc.Card
```

Listing 2.1: Path-dependent type

The `Bank` class defines an abstract type member `Card`. This `Card` type is then used in the definition of the class by the `applyForCard` and `payOff` methods (the variable `b` is the self variable with which other members of the class can be accessed). This means if you have a value, `chase`, of type `Bank`, calling `applyForCard` on it would return a value of type `chase.Card`. This is a path-dependent type because the type is not self-contained. It depends on another variable in the environment. Since the exact type of `chase.Card` is unknown (abstracted away in `Bank`), paying it off at any other Bank would not typecheck, even if the underlying `Card` type for the two Libraries are the same. The type system thus allows the code to model the real world restrictions of cards (cannot pay off a card from one bank at another bank) without restricting the number of possible banks that can be dynamically created.

Unlike modules, type members in objects do not have to be either completely opaque or completely transparent. The exposed type member can be specified with a bound on its subtyping relation: A type member `t` can be either upper-bounded (`t` must be a subtype of another type), lower-bounded (`t` must be a supertype of another type), or exact-bounded (`t` is exactly another type, i.e. completely transparent).[1] This provides the language with not only the ability to represent and typecheck traditional object/record or module types, but also more expressive types that are related to each other.

```
1  class CreditCard {}
2  class SecuredCard extends CreditCard {}
3  class AuthorizedUserCard extends CreditCard {}
4
5  class RegionalBank extends Bank {b =>
6    type Card ≤ SecuredCard
7    ...
8  }
9  def giveChildren(card: SecuredCard): Unit = ...
10
11 val veryCautiousBank : RegionalBank = ...
12 val pnc : Bank = ...
13 study(veryCautiousBank.applyForCard("..."))   // OK
14 study(pnc.applyForCard("..."))                // type mismatch
```

Listing 2.2: Bounds on type members

[1] In this system, a completely opaque type is usually defined with an upper bound of the top type ($\top$), which is defined as the supertype of all types. On the contrary, a bottom type ($\bot$) typically exists and is defined as the subtype of all types.

In Listing 2.2, `RegionalBank` is defined to be the kind of bank whose cards are all of an abstract type that is a subtype of `SecuredCard`. This means cards issued by a `RegionalBank` can be passed to `giveChildren()`, while a card issued by a generic `Bank` whose type member `Card` is completely opaque will not typecheck when given to children (it is not guaranteed to be safe to do so).

Finally, type refinements are also supported as a flexible way of specifying more specific types. In Listing 2.3, the `applyAndGiveChildren` function typechecks because it requires its argument to not just be any `Bank`, but specifically a Bank whose `Card` type is "at most" (i.e. no more general than) `SecuredCard`.

```
1 def applyAndGiveChildren(b: Bank{type Card ≤ SecuredCard}): Unit =
2   study(b.applyForCard("..."))
3
4 applyAndGiveChildren(veryCautiousBank)  // OK
5 applyAndGiveChildren(pnc)               // type mismatch
```

Listing 2.3: Type refinement

2.2 Subtyping and Undecidability

The widespread adoption of object-oriented programming languages has made popular the concept of subtyping, a form of declaration-site inclusion polymorphism [Cardelli and Wegner, 1985] that allows one type to masquerade as another. At its core, subtyping is characterized by the substitution principle: if S is a subtype of T (written S <: T), then values of type S can act like values of type T. Clearly, this provides additional expressive power to programmers by enabling a limited form of bounded parametric polymorphism for functions even without traditional parametric polymorphism support. For example, one could write a traversal method that takes in any type of Graph, and not care what particular subtype of Graph (e.g. DAG, Tree) they actually get since the interface would be as expected from Graph.

Subtype checking is the procedure for checking if one type subtypes another. In the basic scenario above, subtype checking is easily decidable. Since all the types are self-contained names, the predefined subtyping relations (usually defined at the declaration site of the type with keywords such as `extends`) define a partial order on all the type names. Subtype checking is thus checking if the two types are correctly related with respect to the

```
class Bank<Card> {
  Card applyForCard(String name) {
    ...
  }
}
```

(a) Type Parameter in Java

```
class Bank {b =>
  type Card
  def applyForCard(name: String): b.Card =
    ...
}
```

(b) Type Member in Scala

Figure 2.1: Type parameter vs type member

partial order.

However, many modern object-oriented languages also support parametric polymorphism, either in the form of type parameters (e.g. Java generics, C++ templates) or type members (e.g. Scala type members[2]) as shown in Figure 2.1. As a result, not all types are predefined like before. The possibility of constructing new types (by filling in type parameters or refining type members) means subtype checking in these systems must evolve to be structural and recursive. However, A <: B does not necessarily mean each type parameter/member of A is a subtype of the corresponding parameter/member of B. Figure 2.2 illustrates that when A <: B, a type parameter/member can be covariant (i.e. it preserves this subtyping relation) or contravariant (i.e. it reverses this subtyping relation).

```
class ReadStream[+T] {
  def read(): T
}
val rs1 : ReadStream[Int] = ...
val rs2 : ReadStream[Num] = rs1
rs.read()    // return type ≤ Num
```

(a) *Covariant* type parameter (+): Int <: Num
⇒ ReadStream[Int] <: ReadStream[Num]

```
class WriteStream[-T] {
  def write(x: T): Unit
}
val ws1 : WriteStream[Num] = ...
val ws2 : WriteStream[Int] = ws1
ws.write(1)  // input type ≥ Int
```

(b) *Contravariant* type member (-): Int <: Num
⇒ WriteStream[Num] <: WriteStream[Int]

Figure 2.2: Variance of type parameter (in Scala's type parameter syntax)

The difficulty of subtyping arises when a type S is defined as a subtype of some type parameterized with S itself. Such a recursive definition may seem unfamiliar, but recursive subtype definition is heavily used by F-bounded polymorphism [Canning et al., 1989], a generalization of bounded polymorphism where the bounded type can appear in its bound. One common usage, shown in Figure 2.3, is using recursive bounds to specify features of the bounded type. In this case, `String` is defined recursively so that functions ex-

[2]Scala also supports type parameters.

```
trait Cloneable[T] {
  def clone(): T
}
class String extends Cloneable[String] {
  def clone(): String = ...
}
def makeClone[T <: Cloneable[T]](x: T) = x.clone()
```

Figure 2.3: F-bounded polymorphism (in Scala)

Given types:

$$\text{Eq}\langle -T\rangle, \quad \text{List}\langle +T\rangle <: \text{Eq}\langle\text{List}\langle\text{Eq}\langle T\rangle\rangle\rangle, \quad \text{Tree} <: \text{List}\langle\text{Tree}\rangle$$

Query: $\text{Tree} <: \text{Eq}\langle\text{Tree}\rangle$

$$\begin{aligned}
\text{Tree} &<: \text{Eq}\langle\text{Tree}\rangle \\
\text{List}\langle\text{Tree}\rangle &<: \text{Eq}\langle\text{Tree}\rangle \\
\text{Eq}\langle\text{List}\langle\text{Eq}\langle\text{Tree}\rangle\rangle\rangle &<: \text{Eq}\langle\text{Tree}\rangle \\
\text{Tree} &<: \text{List}\langle\text{Eq}\langle\text{Tree}\rangle\rangle \\
\text{List}\langle\text{Tree}\rangle &<: \text{List}\langle\text{Eq}\langle\text{Tree}\rangle\rangle \\
\text{Tree} &<: \text{Eq}\langle\text{Tree}\rangle \\
&\ldots
\end{aligned}$$

Figure 2.4: Infinite derivation example (from Greenman et al. [2014])

pecting Cloneable objects can make more specific inferences about the return type of their `clone()` method.

Subtype checking on these constructed types already involves recursively looking into the structure of both types to make sure all members/parameters satisfy the subtyping relation. Thus, recursive bounds are a potential cause for concern since subtype checking can now possibly loop back to a type it has seen already. Prior research shows this is indeed a challenge. For Java, allowing wildcards with contravariant bounds (i.e. `<? super T>`) is shown to lead to an undecidable subtyping problem [Grigore, 2017]. For Scala, it is the ability to encode system $F_{<:}$, a language already shown to have an undecidable subtyping problem [Pierce, 1992], that makes it undecidable [Amin et al., 2014].

Having an undecidable component in a language can pose real problems to the user (e.g. programmer, compiler & tools implementer). Programmers unfamiliar with the intricacies of the type system of their language may end up unable to compile their "well-

written" program. This problem is worsened in that in these situations the compiler is unable to provide any helpful hints as to why it timed-out/crashed (Running the example in Figure 2.4, the javac compiler loops until it runs out of stack space, and the Scala compiler complains the class graph is not finitary), not to mention any guidance on what can be done to the source code to fix the problem (which is a feature expected of modern compilers). By having a decidable system with clearly defined constraints, compilers and other programming tools will be able to much better assist the programmer in expressing what they want.

2.2.1 Getting Back Decidability

Unfortunately, subtype checking in systems like $F_{<:}$ is not a simple case of cycle detection. Figure 2.6 presents a classic example first discovered by Ghelli [1995] encoded in DOT (based on the translation from Mackay [2019]). As the derivation progresses, we constantly loop back to checking the same structural types but with different variable names, and the context grows larger with these new types. In the general case, it is not trivial to identify a looping derivation. In fact, it is not even ideal to implement only a simple looping detector since, similar to just adding a time-out in the compiler, it does not help the programmer in fixing the problem.

Many have since proposed enforcing some sort of subtype dependency restrictions so that infinitely looping derivations never occur. The most notable is the ban on "expansive inheritance" by Kennedy and Pierce [2006] as it is used by the Scala compiler. An "expansive edge" exists when one type parameter of type S appears at a deeper nested level in the supertype of S. However, as the authors themselves acknowledged, this solution is not immediately applicable to Java wildcards. In addition, Greenman et al. [2014] pointed out that this restriction prevents a common pattern for expressing certain "features" of types: Recall in Figure 2.3 the Cloneable type is used by its subtypes to signal they have a `clone()` method. But if we want a generic list to be cloneable we would get the definition in Figure 2.5, which now includes an expansive edge from E to E.

```
class List[E] extends Cloneable[List[E]] {z =>
  def clone(): List[E] = ...
}
```

Figure 2.5: Cloneable list causing expansive inheritance

Given types:

$$N = \{n \Rightarrow L \geq \bot\}, \qquad T = \left\{ z \Rightarrow \begin{array}{l} A \leq \top \\ B \leq N\{n \Rightarrow L \geq \left\{ z_1 \Rightarrow \begin{array}{l} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\}\} \end{array} \right\},$$

$$T_0 = \left\{ z_0 \Rightarrow \begin{array}{l} A \leq N\{n \Rightarrow L \geq T\} \\ B \leq z_0.A \end{array} \right\}$$

Query: $T_0 <: T$?

$$\emptyset \quad \vdash \quad T_0 \;<:\; T$$

[check bounds on type member B]

$$z_0 : T_0 \quad \vdash \quad z_0.A \;<:\; N\{n \Rightarrow L \geq \left\{ z_1 \Rightarrow \begin{array}{l} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\}\}$$

[follow upper bound of A in T_0]

$$z_0 : T_0 \quad \vdash \quad N\{n \Rightarrow L \geq T\} \;<:\; N\{n \Rightarrow L \geq \left\{ z_1 \Rightarrow \begin{array}{l} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\}\}$$

[check bounds on type member L (swapped sides due to contravariant bound)]

$$z_0 : T_0 \quad \vdash \quad \left\{ z_1 \Rightarrow \begin{array}{l} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\} \;<:\; T$$

[check bounds on type member B]

$$\begin{array}{l} z_0 : T_0 \\ z_1 : \left\{ z_1 \Rightarrow \begin{array}{l} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\} \end{array} \quad \vdash \quad z_1.A \;<:\; N\{n \Rightarrow L \geq \left\{ z_2 \Rightarrow \begin{array}{l} A \leq z_1.A \\ B \leq z_2.A \end{array} \right\}\}$$

To check if a type subtypes another type, we look inside the type structure and compare the bounds on each member. Only the comparison on the type bounds of B is shown above for brevity.

Figure 2.6: Diverging context example

2.2.2 Material-Shape Separation

The solution of Nominal Wyvern is adapted from the "material-shape separation" idea proposed by Greenman et al. [2014] for Java-like languages (instead of DOT, which has type members). Material-shape separation is a conservative way of separating all types in a program into two camps: materials and shapes: A material type represents concrete types that actually represent data, and are passed around in a program. A shape type, on the other hand, are only used to bound other types, typically parameterized with these other types as well (thus creating loops).

The restriction the authors enforced on top of this dichotomy is that all type cycles must go through at least one shape (shapes enable loops), and that shapes cannot be used as type arguments for inherited types. The reasoning behind the feasibility of such a split is that the problematic dependencies in real world programs are not arbitrary, and are usually not representative of the theoretical types that cause the subtyping to loop forever. Indeed, after studying a large corpus of existing code (13.5 million lines of Java), the authors found that current coding practices already mostly follow this separation, and that the rare cases that do not conform can be easily made so.

The benefit of this solution is twofold: 1) The restriction is already compatible with industry programming standards, meaning it would not require any major shift in programming practices for its adoption. 2) The restriction is easy to understand and identify due to a limited number of intuitive uses of shapes. In fact, Greenman et al. identified the two ways in which shapes are used by programmers that are corroborated by their study of existing code: A shape is either used as a bound for the "self" type, as is the case in the aforementioned Equatable example, or used as a bound for the "self" type as part of a type family. This means that instead of arbitrarily restricting what programmers can write (and thus forcing them to adopt an esoteric rule), material-shape separation can serve as a useful tool in helping them structure their code to be more modular, and in a way that most programmers are already familiar with.

2.2.3 Material-Shape Separation for DOT

Getting a similar separation for DOT-based languages requires adapting the original solution from relying on type parameters to rely instead on type members. The most straightforward translation from a type parameter-based system like Java to a type member-based

system like DOT is using type members to represent type parameters (e.g. Figure 2.7). However, if we directly translate Greenman et al.'s restriction that shapes cannot be type parameters to this system, we end up disallowing the use of shapes when defining type member bounds. This is a notably wider restriction than it was originally meant for in Java since type members have more uses than specifying type parameters. For example, given a shape type `Equatable`, if we wanted to write a type storing a pair of equatable objects, we would have to use equatable to define the type members, even if the parent type `Pair` is not part of any subtype chain.

Nominal Wyvern's adaptation of material-shape separation is inspired by work by Mackay [2019] on Decidable Wyvern, in which the authors proposed an adaptation of material-shape separation to DOT-based systems. Decidable Wyvern's solution is a combination of semantic and syntactic restrictions. Shapes are still defined as the enabler of cycles in the subtyping dependency graph, but additionally all cycles must pass through structural types (to avoid meaningless cross inter-member dependencies). On the syntactic side, shapes can only serve as upper bounds for materials; shapes can only be refined with purely materials; shapes can only be defined with purely materials; and additionally shapes can only be upper-bounded by purely material refinements on the top type. This ensures subtype derivations always sink towards purely material types, from where no cycles will ever occur and termination of subtyping is guaranteed.

Nominal Wyvern differs from Decidable Wyvern mainly in having a nominal typing and subtyping system. This allows for a slightly simpler set of material-shape separation restrictions, which will be explained in detail in Section 3.4.

```
interface Pair<T> {
  T getLeft()
  T getRight()
}

class Point implements Pair<Int> {
  Int getLeft() { ... }
  Int getRight() { ... }
}
```

```
trait Pair { z =>
  type E ≤ ⊤
  def getLeft(): z.E
  def getRight(): z.E
}
class Point extends Pair { z =>
  type E = Int
  def getLeft(): Int
  def getRight(): Int
}
```

Figure 2.7: Example translation from type parameter to type member

2.3 Nominality

The typing of the DOT calculus is already considered partly nominal as it relies on the names of objects to get path-dependent types. The nominality we present here goes one step further by mandating all structural types be named, and that the subtyping relations between these named structural types be entirely nominal as well. This contributes to a simpler and more usable system. The particular formulation of nominality in Nominal Wyvern also conveniently makes the material-shape separation rules simpler by syntactically preventing forms of mutually recursive structural definitions from appearing (instead, forcing structures to carry their own meaning and relate to others' type members through refinements).

2.3.1 Typing Nominality

In contrast to DOT where a new structural type can be defined anywhere anonymously by simply writing out its members, Nominal Wyvern requires all structural types to be pre-declared and named in the scope. This is due to two reasons:

1. Usability: Explicitly written out structures make the code easier to understand since the names would be representative of what the structure is used for. This is especially important as the object gets larger and contains more kinds of members (i.e. type members, value members, function members). Equally important is that having easily identifiable names would make any information the compiler or other programming assisting tools generate be more readable.

2. Performance: Having named structures allows the typechecker to easily store prechecked subtyping relations so that later subtyping queries (or ones with the same name but with slight refinements) can avoid repeating work. The next subsection also explains how this benefit is augmented with a nominal subtyping system.

2.3.2 Subtyping Nominality

Traditionally, structural subtyping is done by comparing each member between two types to ensure each satisfied the subtyping relation defined on structures (as shown in Figure 2.8). A nominal subtyping system requires, in addition to structural compatibility via

```
{                                    {
  val name : String                    val name : String
  val bestseller : Food          <:
}                                    }
```

(a) Width Subtyping

```
{                                    {
  val name : String                    val name : String
  val bestseller : FastFood      <:    val bestseller : Food
}                                    }
```

(b) Depth Subtyping

```
{                                    {
  def buy(card: CreditCard)            def buy(card: SecuredCard)
        : FastFood               <:          : Food
}                                    }
```

(c) Subtyping with Methods

```
{                                    {
  type exportType <: FastFood          type exportType <: Food
  type importType >: Food        <:    type importType >: FastFood
}                                    }
```

(d) Subtyping with Type Members

Figure 2.8: Structural Subtyping

structural subtyping, that the names associated with structures have been declared to be related. That is, type S subtypes type T only if the programmer writes that they want S to subtype T. In Figure 2.9, if the programmer never declares SchoolCafeteria to be a subtype of Restaurant, the type system cannot force that on them even though they are structurally compatible.

```
SchoolCafeteria {                    Restaurant {
  val name : String                    val name : String
  val bestseller : FastFood     !<:    val bestseller : Food
}                                    }
```

Figure 2.9: Width Subtyping

The provides the two familiar benefits:

1. Usability: Explicitly naming subtyping relations avoids accidental subtyping by making sure that types whose signatures match are not automatically considered

related. Since all structural types are named, types that are structurally compatible should not be related if the meaning associated with the names do not match. This makes the type system better match up with what a programmer reading the code expects, and prevents accidental passing of the wrong argument to a function even if coincidentally the structures match up. The added benefit is a more understandable subtyping relation to the code reader (many subtyping checks are quite involved as the rules will soon show).

2. Performance: Having all subtyping relations defined explicitly means more type-checks can be done on the type signatures (and subtyping assertions) alone. The result of these checks can also be saved and reused throughout the typechecking of the dynamic expression, saving repeated checks that may be long and recursive.

Chapter 3

Nominal Wyvern Design

This chapter discusses how the nominal design of Nominal Wyvern facilitates subtype decidability and usability. Sections 3.1 to 3.2 build up a motivating example; Sections 3.3.1 to 3.5 explain the design while referencing the earlier example.

3.1 A Store of Named Structures

Suppose we want to write a system for keeping track of the stock at a store. Each piece of fruit is labelled with an ID number, and they're weighed when entered into the system. Similar to DOT, this warrants a record (aka "structural") type with the two member values. As evident from the following code, the interface for both apple and orange are the same. If not careful, a completely structural type system would fail miserably as any function that is supposed to operate on `Apples` would "physically" work just as well when given an `Orange`. Having a nominal type system means the type system will prevent the programmer from mixing apples and oranges since structural types are given *names* in addition to their members. Similarly, a nominal subtyping system means that while both McIntosh and Macintosh are structurally compatible with Apple (in this case, a form of width subtyping[1]), only the one that is declared to be a subtype explicitly by the programmer can be used as an Apple. In Nominal Wyvern, this is declared with a special subtype declaration as seen on line 26 of Listing 3.1.

[1]Width subtyping refers to allowing subtypes to have more members (hence 'width') than its supertype. This is often supplemented with depth subtyping, which allows members to have more specific type bounds in the subtype than in the supertype.

```
// fruits for sale
name Apple {
  val id : Int
  val weight : Float
}
name Orange {
  val id : Int
  val weight : Float
}

// a flavor of apple
name McIntosh {
  val id : Int
  val weight : Float
  val price : Int
}
// a product of Apple Inc
name Macintosh {
  val id : Int
  val weight : Float
  val model : String
  val price : BigInt
  ...
}
// explicitly declared subtyping relation
subtype McIntosh <: Apple
```

Listing 3.1: A nominal typing system on structural interfaces (Part 1)

With all the basic setup completed, the following code uses the declared interfaces to keep track of the stock with two simple counters encapsulated in a `StockCounts` interface. `StockTracker` operates on the counter, and the type system makes sure that subtyping is completely nominal by disallowing the mixing of apples and oranges.

```
// simple stock counter
name StockCounts {
  val numApples : Int
  val numOranges : Int
}
name StockTracker {
  def empty () : StockCounts
  def importApple (a : Apple, st : StockCounts) : StockCounts
```

```
35    def importOrange (o : Orange, st : StockCounts) : StockCounts
36  }
37
38  let a1 = new Apple { val id = 0, val weight = 90.0 } in
39  let o1 = new Orange { ... } in
40  let m1 = new McIntosh { ... } in
41  let mac = new Macintosh { ... } in
42  let tracker = new StockTracker {
43    def empty () : StockCounts =
44      new StockCounts { val numApples = 0, val numOranges = 0 }
45    def importApple (a : Apple, st : StockCounts) : StockCounts =
46      new StockCounts {
47        val numApples = st.numApples + 1
48        val numOranges = st.numOranges
49      }
50    def importOrange (o : Orange, st : StockCounts) : StockCounts = ...
51  }
52  let empty_stock = tracker.empty() in
53  stock = tracker.importApple(a1, empty_stock)  // OK
54  stock = tracker.importApple(o1, empty_stock)  // type mismatch
55  stock = tracker.importApple(m1, empty_stock)  // OK
56  stock = tracker.importApple(mac, empty_stock) // type mismatch
```

Listing 3.2: A nominal typing system on structural interfaces (Part 2)

3.2 Adding Generics

Suppose now we want to make our stock a "set" of fruits instead of just a count. This way we can make sure we do not overestimate our stock if we accidentally scan the same piece of fruit twice. We want to make the set generic so it can work for any type of fruit, but in order for the set to properly operate, the element type must have an `equals` method defined on it so we can remove duplicates. In Listing 3.3, we update the code so Apple and Orange both subtype Fruit, which in turn subtypes a new `Equatable` type that has an equals method.

Equatable is defined with F-bounded polymorphism. Similar to the Cloneable example in Figure 2.5, the type member `T`'s purpose is for future subtypes to refine with themselves. In order for `equals` to refer to it, we introduce a self variable to the named interface, just

like with Scala. This self variable (named `z` in the example code) represents the object that the member is being accessed on. For example, for any object that instantiates Fruit, its equals method must take in an object of its own `T` type.

```
1  name Equatable {z =>
2    type T >= BOT
3    def equals (x : z.T) : Bool
4  }
5  name Fruit {z =>
6    type T >= BOT
7    val ID : Int
8    val weight : Float
9    def equals (x : z.T) : Bool
10 }
11 subtype Fruit <: Equatable
12 name Apple {z =>
13   type T >= BOT
14   ...
15 }
16 name Orange {z => ... }
17 subtype Apple {type T >= Apple} <: Fruit
18 subtype Orange {type T >= Orange} <: Fruit
```

Listing 3.3: Adding generics to Nominal Wyvern (Part 1)

The `Set` type consists of an element type `elemT` that must subtype `Equatable`. A refinement serves as a qualification of the `Equatable` type that additionally requires the `T` type of `Equatable` to be lower bounded by `elemT` itself. This is key in enabling any client method to pass objects of `elemT` type to the equals method of another object of `elemT` type.

```
19 name Set {z =>
20   type elemT <= Equatable {type T >= z.elemT}
21   ...
22 }
23 name SetTracker {z =>
24   type S <= Set
25   def empty () : z.S
26   def insert (stock : z.S, item : stock.elemT) : z.S
27   ...
28 }
29
```

```
let apple_tracker = new SetTracker {z =>
  type S = Set { type elemT = Apple }
  def empty () : z.S =
    new Set {s =>
      type elemT = Apple
      ...
    }
  def insert (stock : z.S, item : stock.elemT) : z.S = ...
  ...
} in
let a1 = new Apple {type T = Apple} {z =>
  type T = Apple
  val ID = 0
  val weight = 90.0
  def equals (x : Apple) : Bool = (z.ID == x.ID)
} in
let o1 = new Orange {type T = Orange} {z =>
  type T = Orange
  ...
} in
let apple_stock = apple_tracker.empty() in
apple_stock = apple_tracker.insert(apple_stock, a1) // OK
apple_stock = apple_tracker.insert(apple_stock, o1) // type mismatch
```

Listing 3.4: Adding generics to Nominal Wyvern (Part 2)

3.3 Language Design

Figure 3.1 shows the formal grammar of Nominal Wyvern. As evident from the earlier code example, programs in Nominal Wyvern consist of two parts: A list of top-level declarations ($\overline{D}$), and a main expression (e) to be evaluated during execution.

The top-level declarations is where all structural types and the subtype relations between them are specified. Each named type declaration binds a structure with a name. Similar to DOT, the structure can contain type members, fields, and methods. However, type members in Nominal Wyvern is only bounded on one side (unless it is an exact bound, which bounds on a type member on both sides by the same type). This design decision was made to simplify the syntax since in most practical cases a bound on one side is

sufficient. This is supported by evidence in Chapter 5 that shows the expressiveness of Nominal Wyvern by encoding common patterns from both object-oriented languages and functional languages. Explicit subtype declarations sets up a relation between two named types with an optional refinement allowed on the LHS. Section 3.3.1 details how the top level declarations differ from their DOT counterparts.

A type in Nominal Wyvern is made of a base type and a refinement. Base types define the actual structure of a type, while the refinement allows ad hoc modifications to the base type that override the definitions of some of its members. A base type is either top ($\top$), bottom ($\bot$), a raw named type (n), or a path-dependent type ($p.t$) (A path, as usual, is a variable appended with zero or more successive field accesses)[2]. Refinements in Nominal Wyvern refine members of base types. Types and decidability are the main focus of this thesis, so Nominal Wyvern only considered type member refinements (instead of refinements for all three kinds of members). A more complete refinement formulation may be added to future iterations of this system if deemed necessary.

Nominal Wyvern carries only the basic forms expressions. All expressions in Nominal Wyvern are objects. A path is the smallest unit for object representation. The grammar requires method applications to be on paths instead of arbitrary expressions so that all objects in scope are named, which makes typing (and future evaluation) rules sligtly simpler. It does not hinder expressiveness as Chapter 5 will show since it is trivial to translate a program with arbitrary expressions to this form. "New" expressions are the only way of creating objects. Since all types are named, and since different names can correspond to the same structure, the exact type of an object is specified upon creation. The "let" expression is important for Nominal Wyvern since it is what enables all objects to have a name as required earlier. Finally, an "if" expression is included to allow for simple control flow (mainly useful for showing more complicated examples for the expressiveness argument). The programmer is required to supply the type of the entire expression, which is expected to be a supertype for the types of both branches. This ensures the entire expression is typeable since arbitrary intersection types are removed in favor of naming every structure. The two paths involved in the conditional will be checked dynamically for location equality. This way dynamic conditionals can be achieved without any primitive types like booleans or natural numbers.

[2]A path can also start with a location, which is used to represent heap locations during evaluation. Note that evaluation rules are not included in this work

P	::=		***program:***
	$\overline{D}\ e$		
D	::=		***top-level decls:***
	name $n\ \{x \Rightarrow \overline{\sigma}\}$		*named type decl*
	subtype $n\ r <: n$		*subtype decl*
B	::=		***type bound:***
	$\leq$		*upper bound*
	$\geq$		*lower bound*
	$=$		*exact bound*
σ	::=		***member decl:***
	type $t\ B\ \tau$		*type member decl*
	val $v\ :\ \tau$		*field decl*
	def $f\ :\ \tau\ x \rightarrow \tau$		*method decl*
r	::=		***refinement:***
	$\{\overline{\delta}\}$		
δ	::=		***refinement member decl:***
	type $t\ B\ \tau$		

β	::=		***base type:***
	$\bot$		*bottom type*
	$\top$		*top type*
	n		*named type*
	$p.t$		*path type*
τ	::=		***type:***
	$\beta\ r$		
p	::=		***path:***
	x		*variable*
	l		*store location**
	$p.v$		*val selection*
e	::=		***expression:***
	p		*path*
	$p.f(p)$		*method application*
	new $\tau\{x \Rightarrow \overline{d}\}$		*new object*
	let $x = e$ **in** e		*let expr*
	if$[\tau]\ p = p$ **then** e **else** e		*if expr*
d	::=		***object member defn:***
	type $t\ =\ \tau$		*type member defn*
	val $v\ :\ \tau_v\ =\ e$		*field defn*
	def $f\ :\ \tau\ x \rightarrow \tau = e$		*method defn*

Δ	::= $\overline{n : \{x \Rightarrow \overline{\sigma}\}}$	*Name Definition Context*
Σ	::= $\overline{n\ r <: n}$	*Name Subtype Relation Context*
Γ	::= $\overline{x : \tau}$	*Variable Typing Context*
S	::= $\overline{l : \tau}$	*Location Typing Context*
μ	::= $\overline{l : \{x \Rightarrow \overline{d}\}}$	*Runtime Store*

* Intermediate form only (not user accessible)

Figure 3.1: Nominal Wyvern Grammar

The following subsections dive into certain noteworthy aspects of the Nominal Wyvern design.

3.3.1 A Binary Typing Approach

The main difference of Nominal Wyvern compared to DOT is its heterogenous typing system. In DOT, all nominal types are type members of some other types, whereas in Nominal Wyvern, there are two sorts of types: The concrete structural types (aka "named types") and the abstract member types. At first glance Nominal Wyvern may seem like an extraction of all structural types (along with any width-expanding refinement) to a global object type in DOT, but there is a more fundamental difference between named types and type members that contributes to Nominal Wyvern's theme of usability and decidability.

The key difference between named types and type members is the way they are specified. Type members are declared with a bound: either with only a lower bound, only an upper bound, or both bounds that are the same in the case of an exact bound. In contrast, named types are defined as a named record for representing entities with the given properties. Semantically, this more closely resembles the usual definition of a "definition": If our store thinks of each Apple as a record with an integer ID and a floating point weight, then that is exactly what the named type "Apple" is. Contrast this with the declaration of a type member "Apple" with a bound on the subtype relation, which more closely resembles a guideline and guarantee on how generic/specific this type may be for future instantiators and users of the parent type (Recall in Listing 3.4, the type member `elemT` of `Set` is defined with an upper bound to make sure all instantiators use an equatable type in sets). This dichotomy separates responsibility by making named types the definer of types, and type members merely users of the pre-defined types. Consequently, all instantiable types (except the native top and bottom types) are defined by some structure in the top-level.

The separation of named types from member types also warrants a separate way of defining the subtype relation between named types. Traditionally in DOT, type bounds perform two roles with different semantic meanings: One can use a type bound to either define a "guarantee" on a particular type member or define a subtype relation between nominal types. Nominal Wyvern separates this mix of semantically distinct roles with two separate constructs. Type bounds still exist but are made more focused: They are used only to specify guarantees on type members. E.g. if we know a particular member type is

always a subtype of `Fruit`, we can use it wherever fruit is accepted. Complementing type members are explicit subtype declarations. Their sole purpose is to define the nominal subtype relation between pre-defined named types. E.g. if we think of McIntosh as a special kind of apple, then we can explicitly declare this subtype relation between the two named types `McIntosh` and `Apple`. We can think of the explicit subtype declarations as defining base cases for the substitution principle. They are only base cases because depth subtyping in the form of type refinement is automatic (e.g. in Listing 3.4, `Apple {type T = Apple} <: Apple`).

In addition to the usability benefits of semantic separation (e.g. clearer code), explicit subtype declarations also allow for more flexibility than traditional bounds in two ways:

1. *Multiple Subtyping*: Type T can be a subtype of multiple types. This is often used when one type wants to have the features of many other types. For example, a resource type (such as Apple) can declare itself a subtype of both Equatable and Hashable to signal it has both an `equals` method and a `hash` method, so types that require either one (e.g. Set, Hashtable) can use it as the key. This mitigates the loss from not having arbitrary intersection types by essentially requiring each intersection type to have a unique name.

2. *Conditional Subtyping*: Type T can be a subtype of another type only if T is under certain refinements. It may be the case that named type T is not inherently a subtype of named typed S. This could be due to either structural incompatibility or semantic incompatibility (i.e. the subtype relation does not match the semantic meanings attached to the types). The syntax allows a refinement on the LHS of the subtype symbol (<:) to make a subtype relation hold conditionally. For example, this was used in Listing 3.3 for the F-bounded polymorphism example.

This difference carries over to the subtyping algorithm as well. While the type bound on a type member S represents the authoritative "next" type to check after S, the subtype declarations with S as the base type on the LHS present us with multiple conditional options for what this type could also be seen as by the substitution principle. Chapter 4 details the binary subtyping algorithm and why it is a decidable problem after material-shape separation is applied.

3.3.2 Top-Level Well-Formedness

The static contexts Δ and Σ are derived solely from the top-level declarations $\overline{D}$, and are used in the typing and subtyping rules detailed in the following sections. Figures 3.2 and 3.3 present the judgment rules for top-level declaration well-formedness. Note that all named types are considered to be declared at the same time (in the same scope and can reference each other). One noteworthy constraint when checking named type well-formedness ($\Delta \vdash n : \{x \Rightarrow \overline{\sigma}\}$ wf) is that the bounds on type members cannot reference any of its sibling fields. This prevents infinite typing derivations that bounce between fields and type members, which will be shown in section 3.5.

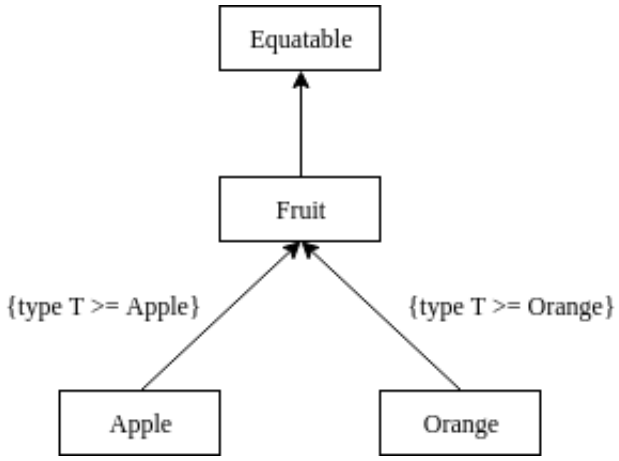
3.3.3 Nominal Subtyping Graph

To aid in nominal subtyping, we define a "nominal subtyping graph" to capture the multiple conditional subtyping relations between named types.

Definition 1 (Nominal subtyping graph)**.** *For a set of top level declarations $\overline{D}$, the nominal subtyping graph is a graph $\langle V, E \rangle$. The vertices, V, consist of all the named types in $\overline{D}$. The edges, E, each represent an explicit subtype declaration in $\overline{D}$, with the refinement labeled on the edge:*

$$\frac{n_1\, r_1 <: n_2}{n_1 \xrightarrow{r_1} n_2}$$

Example 1. *The nominal subtyping graph for the code in Listing 3.4 is:*



$\boxed{\overline{D}\text{ wf}}$

$$\frac{\text{names}(\overline{D}) = \Delta \qquad \forall\, n : \{x \Rightarrow \overline{\sigma}\} \in \Delta.\ \Delta\Sigma \vdash n : \{x \Rightarrow \overline{\sigma}\}\text{ wf} \qquad \text{subs}(\overline{D}) = \Sigma \qquad \forall\, n_1\, r_1 <:\ n_2 \in \Sigma.\ \Delta\Sigma \vdash n_1\, r_1 <:\ n_2}{\overline{D}\text{ wf}}$$

$\boxed{\text{names}(\overline{D}) = \Delta}$

$$\frac{}{\text{names}(\cdot) = \cdot} \qquad \frac{\text{names}(\overline{D}) = \Delta}{\text{names}(\overline{D}, \texttt{name}\ n\ \{x \Rightarrow \overline{\sigma}\}) = \Delta, n : \{x \Rightarrow \overline{\sigma}\}}$$

$\boxed{\text{subs}(\overline{D}) = \Sigma}$

$$\frac{}{\text{subs}(\cdot) = \cdot} \qquad \frac{\text{subs}(\overline{D}) = \Sigma}{\text{subs}(\overline{D}, \texttt{subtype}\ n_1\, r_1 <:\ n_2) = \Sigma, n_1\, r_1 <:\ n_2}$$

$\boxed{\Delta\Sigma \vdash n : \{x \Rightarrow \overline{\sigma}\}\text{ wf}}$

$$\frac{\forall\, \texttt{type}\ t\ B\ \tau_t \in \overline{\sigma}.\ \forall\, \texttt{val}\ v\ :\ \tau_v \in \overline{\sigma}.\ v \notin \tau_t \wedge \Delta\Sigma(x : n) \cdot \vdash \tau_t\text{ wf}}{\Delta\Sigma \vdash n : \{x \Rightarrow \overline{\sigma}\}\text{ wf}}$$

$\boxed{\Delta\Sigma \vdash n_1\, r_1 <:\ n_2}$

$$\frac{\Delta(n_2) = \{x_2 \Rightarrow \overline{\sigma_2}\} \qquad r_1 = \{\overline{\delta_1}\} \qquad \begin{array}{c}\Delta(n_1) = \{x_1 \Rightarrow \overline{\sigma_1}\} \\ \Delta\Sigma(x_1 : n_1\, r_1) \cdot \vdash \overline{\sigma_1} +_\sigma \overline{\delta_1} <:\ [x_1/x_2]\overline{\sigma_2}\end{array}}{\Delta\Sigma \vdash n_1\, r_1 <:\ n_2}$$

where $+_\sigma$ is the binary merge operation on $\overline{\sigma}$. On conflict, RHS is preserved and LHS is discarded.

Figure 3.2: Nominal Wyvern Top-Level Declarations Well-Formedness

$\boxed{\Delta\Sigma\Gamma S \vdash \overline{\sigma} <: \overline{\sigma}}$

$$\frac{}{\Delta\Sigma\Gamma S \vdash \overline{\sigma} <: \cdot} \qquad \frac{\begin{array}{c} r_1 \ni \texttt{type}\ t\ B_1\ \tau_1 \\ \Delta\Sigma\Gamma S \vdash \texttt{type}\ t\ B_1\ \tau_1 <: \texttt{type}\ t\ B_2\ \tau_2 \\ \Delta\Sigma\Gamma S \vdash \overline{\sigma_1} <: \overline{\sigma_2} \end{array}}{\Delta\Sigma\Gamma S \vdash \overline{\sigma_1} <: \overline{\sigma_2}, \texttt{type}\ t\ B_2\ \tau_2}$$

$$\frac{\begin{array}{c} r_1 \ni \texttt{val}\ v\ :\ \tau_1 \\ \Delta\Sigma\Gamma S \vdash \texttt{val}\ v\ :\ \tau_1 <: \texttt{val}\ v\ :\ \tau_2 \\ \Delta\Sigma\Gamma S \vdash \overline{\sigma_1} <: \overline{\sigma_2} \end{array}}{\Delta\Sigma\Gamma S \vdash \overline{\sigma_1} <: \overline{\sigma_2}, \texttt{val}\ v\ :\ \tau_2}$$

$$\frac{\begin{array}{c} r_1 \ni \texttt{def}\ f\ :\ \tau_{a1}\ x_1 \to \tau_{r1} \\ \Delta\Sigma\Gamma S \vdash \texttt{def}\ f\ :\ \tau_{a1}\ x_1 \to \tau_{r1} <: \texttt{def}\ f\ :\ \tau_{a2}\ x_2 \to \tau_{r2} \\ \Delta\Sigma\Gamma S \vdash \overline{\sigma_1} <: \overline{\sigma_2} \end{array}}{\Delta\Sigma\Gamma S \vdash \overline{\sigma_1} <: \overline{\sigma_2}, \texttt{def}\ f\ :\ \tau_{a2}\ x_2 \to \tau_{r2}}$$

where $\Delta\Sigma\Gamma S \vdash \sigma <: \sigma$ is the subtype judgment on σs (defined with the other subtyping rules in Chapter 4).

Figure 3.3: Nominal Wyvern Top-Level Declarations Well-Formedness (continued)

3.4 Material-Shape Separation

The definition of materials and shapes is based on the discovery by Greenman et al. [2014] that shapes should be the only types that enable cycles during subtype derivation. Practically, shapes are used either to define features of the self-type, or to define features of a type family that the self-type is a part of. In the fruit shop example, Equatable is an obvious case of the former use of shape since it prescribes that whatever subtypes it must have an equals method that takes in a value of their own T type.

For Java, Greenman et al. [2014] defines a subtype dependency graph (called "inheritance usage graph" in their paper) that maps each type S to the types mentioned in its inheriting type so that when S becomes the LHS type, we know what types will be visited by the subtype derivation afterwards. By preventing the cycle-inducing shapes from appearing as type arguments, no new usages of shapes are created and eventually all shapes are reduced into materials.

Nominal Wyvern follows a similar adaptation of the subtype dependency graph as

Mackay [2019]'s adaptation in Decidable Wyvern[3]. The difference is due to the binary typing approach that Nominal Wyvern employs. Before defining this graph, we first define the refinement tree for types.

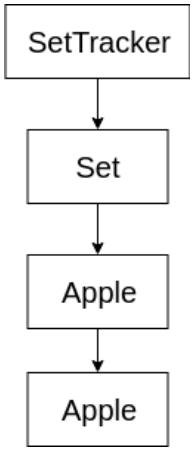
Definition 2 (Refinement tree)**.** *For a type $\tau = \beta\, r$, the refinement tree is a tree $\langle V, v_r, E\rangle$. The vertices, V, consist of all the base types referred to in τ. The root vertex, v_r, is β. The edges, E, each go from a base type, $\beta' \in \tau$ (β' syntactically appears in τ), to the root of the refinement tree generated by the type used as the bound in a refinement to β'. It is formally generated by* `GenTree`*:*
$\texttt{GenTree}(\tau) = \langle V, v_r, E\rangle$*. (Each edge is represented by $v_1 \to v_2$: An edge from node n_1 to n_2).*

$$\frac{r = \texttt{type}\ t_1\ B_1\ \tau_1\ \ldots\ \texttt{type}\ t_n\ B_n\ \tau_n \qquad \texttt{GenTree}(\tau_1) = \langle V_1, r_1, E_1\rangle \qquad \ldots \qquad \texttt{GenTree}(\tau_n) = \langle V_n, r_n, E_n\rangle}{\texttt{GenTree}(\beta\, r) = \langle V_1\ \ldots\ V_n\ \beta,\ \beta,\ E_1 \ldots E_n\ (\beta \to r_1) \ldots (\beta \to r_n)\rangle}$$

Example 2. *The refinement tree for type*

```
SetTracker {
  type S = Set {
    type elemT = Apple {T = Apple}
  }
}
```

is:



[3]Detailed in Section 2.2.3

The subtype dependency graph is defined partially using refinement trees.

Definition 3 (Subtype dependency graph (SDG))**.** *For a set of top level declarations $\overline{D}$, the subtype dependency graph is a graph $\langle V, E\rangle$. The vertices, V, consist of all available base types, which are: $\top$, $\bot$, all declared named types in $\overline{D}$ and their type members. To disambiguate type members with the same name from different named types, each type member t of name n is denoted $n{::}t$ (which will be referred to as a "pseudotype"). The edges, E, are generated as follows:*
For each type member declaration, `type` $t\ B\ \beta_t\, r_t$, in named type n, generate edges:

$$\frac{}{n{::}t \to \beta_t}\ \text{DIRECT} \qquad \frac{\texttt{type}\ t_r\ \_\ \beta_r\, r_r \in^* r_t}{n{::}t \xrightarrow{rta(\beta_t\, r_t, \beta_r)} \beta_r}\ \text{INDIRECT}$$

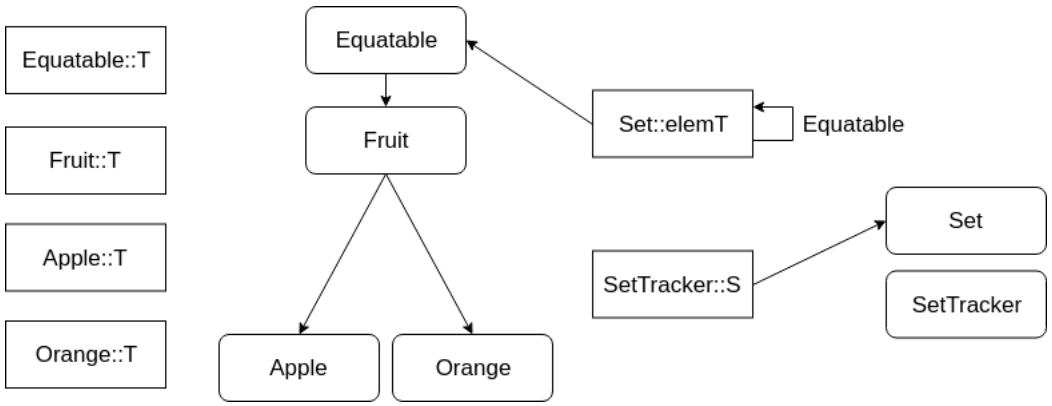
And for each subtype declaration `subtype` $n_1\, r_1 <: n_2$ in $\overline{D}$, generate edges:

$$\frac{}{n_2 \to n_1}\ \text{BACK} \qquad \frac{\texttt{type}\ t_r\ \_\ n_r\, r_r \in^* r_1}{n_2 \to n_r}\ \text{BACK-REF-ROOT}$$

where $\delta \in^* r$ is true if the refinement member δ occurs anywhere syntactically within r, and $\delta \in r$ is true only if δ is immediately within r, and
where $rta(\tau, \beta)$ returns all the ancestors of β in the refinement tree of τ in the order from root of tree to β (not including β itself), and
where $v_1 \to v_2$ represents an unlabeled edge, and $v_1 \xrightarrow{\beta_1\beta_2\ldots} v_2$ represents an edge labeled with the base types $\beta_1, \beta_2, \ldots$ in that order.

The idea is if β has an edge to ("depends on") β', then when β is encountered as a base type during subtype derivation, β' may appear as the base type later in the derivation. The graph is thus a conservative guarantee of what types will not be visited again in later derivations. This serves as the key idea for proving decidability in Chapter 4.

Example 3. *The subtype dependency graph for the declarations in Listing 3.4 is:*



(Rounded rectangles represent named types. Rectangles represent member types. $\top$ *and* $\bot$ *are not included.)*

Shapes are still defined to be the types that enable cycles in the subtype dependency graph:

Definition 4 (Shape type)**.** *Shapes are the set of types such that if all edges labeled with at least one shape are removed from the subtype dependency graph, then the graph is acyclic. All other types are material types.*

Clearly, the solution to which set of types are the shapes in a program is not unique: one can simply label all types as shapes. However, not only does that not follow the semantic meaning of shapes and materials detailed earlier (the two uses of shapes), the material-shape separation requirements will further limit which types are allowed to be shapes. Observe that as long as the separation requirements are fulfilled, the number of possible solutions does not matter for decidability as long as there is one valid set of shapes.

Definition 5 (Material-shape separation)**.** *A program is material-shape separated if there exists a set of shape types such that:*

- *A shape is never used as part of a lower bound syntactically (i.e. after* $\geq$ *or* $=$*).*
- *The upper bound of a shape is always a shape, and named shapes can only subtype named shapes.*
- *Shapes cannot be refined in refinements.*

Example 4. *For the code in Listing 3.4, the only shape type is* `Equatable` *because removing the self loop that is labeled with it will make the subtype dependency graph in Example 3 be acyclic.*

It is a valid use of shape since it follows the restrictions in Definition 5.

The separation rules are partly inherited from earlier works by Greenman et al. [2014] and Mackay [2019], and partly created for Nominal Wyvern.

From a semantic separation standpoint, the intuition behind the rules follow how shapes are supposed to be used in programs. As explained earlier in Section 2.2.2, shapes can be thought of as the types that are used for the sole purpose of F-bounding some other type to guarantee that the other type has certain features (for example, `Equatable` is a shape since it is used as bounds in F-bounded polymorphism, and serves to show that whatever subtypes it has an `equals()` method). As a result, it would, first of all, make sense only to serve as upper bounds of other types. This view of shapes also means that whatever type a shape subtypes must be another shape, otherwise the substitution principle would allow shapes to be used as materials, which does not fall in line with the only role shapes are supposed to perform. Finally, a shape should not be used in a refinement when there are no self-variables in scope. Since shapes are supposed to be used by F-bounded polymorphism, the lack of self-variables means there is no way to use shapes to recursively bound a type. The rules encourage the programmer to declare their F-bounded types as pre-declared structures instead of refining pre-declared structures to utilize F-bounded later.

From a decidability standpoint, the rules allow decidability of subtyping to be achieved (and proved in Chapter 4). However, following the theme of usability, the rules are designed not to be the tightest they can be to make the proof goes through. Instead, they are made to be easier to understand and follow as explained earlier. Nonetheless, the primary goal of the separation rules is still achieving decidability. Therefore, it does not govern other best practices of using shapes vs. materials (for example, one should not be instantiating shapes as objects).

3.4.1 Comparison to Decidable Wyvern

Nominal Wyvern's version of material-shape separation is based on the adaptation of material-shape separation to DOT in Decidable Wyvern by Mackay [2019]. Both Nominal Wyvern and Decidable Wyvern achieve decidable subtyping with material-shape separation, but the details are different due to the added nominality of Nominal Wyvern.

Dependency graph construction is similar for the two systems in that type members

depend on the types in its bound (in fact, this concept finds its roots all the way in Greenman et al. [2014]). The construction process is, however, simpler in Decidable Wyvern because it is based on a more uniform type system (instead of the binary typing approach here). In contrast, Nominal Wyvern's subtype dependency graph is separated into nodes that are named types and nodes that are type members (pseudotypes). By sacrificing uniformity, however, Nominal Wyvern actually allows the entire subtype dependency graph to be easily partitioned into $N + 1$ separate graphs that can be individually checked for cycles (where N is the number of named types in the top-level): For each named type we can identify a sub-graph that only contains the edges between the pseudotypes of this named type (this gives us N sub-graphs), and all the edges between named types generated by the subtype declarations can be put into one other sub-graph. Since a named type node only points to named type nodes, and a pseudotype node $n::t$ only points to either 1) named types, or 2) other pseudotypes in n, there will be no cycles that span multiple sub-graphs. This separation makes graphs more easily checkable for cycles, and also prevents long-spanning dependencies that may be hard to understand.

The way material-shape separation rules are defined also differs. For Nominal Wyvern, the separation rules are treated as an additional requirement that needs to be separately enforced during typechecking. In contrast, Decidable Wyvern incorporates the separation rules directly in its grammar. The benefit of having a syntactical restriction means the separation rules can be easily specified along with its grammar, and can be easily checked by existing parsers. However, this also means the grammar involves many nuanced details that may be hard to follow: Users will need to always be aware of whether they are currently programming a shape or material, and be wary of what syntactical constructs are not allowed. Nominal Wyvern trades-off the easily specifiable separation rules in favor of a simpler grammar with an additional separation check enforced on top.

3.5 Term Typing

All expressions in Nominal Wyvern correspond to objects. All expressions are given names, either assigned with a `let` expression, or as a `val` member in another object. Objects are then used by referring to the path that refers to their names, either directly as the assigned variable in a `let` expression, or by selecting a `val` member from another object. This simplification makes objects slightly easier to work with in the type

system while not hindering expressiveness at all, since to use "anonymous" objects, one only needs to wrap the object in a local `let` expression with a fresh variable name and immediately use it.

The judgments for typing expressions in Nominal Wyvern are shown in Figure 3.4. The two relevant contexts are the variable typing context Γ and the location typing context S. Γ keeps track of the type of each variable and S keeps track of the concrete type of objects for evaluation (S is currently reserved for future evaluation rules). The typing of variables and locations is therefore just looking up the relevant context. The typing of a non-zero-length path $p.v$ involves first typing the immediate sub-path p, expanding its type into the name type it was based on with an additional refinement, and looking up the type of v from there.

Method application typing is done by first checking the argument type is a subtype of the required argument type specified by the type of the method receiver. Note that the argument type of the function comes from the exposed type of the object rather than the actual internal type. The resulting type of the application is the exposed return type with the argument variable and self variable replaced with paths from the current context.

All objects are created with the `new` expression and are given a type explicitly. This is in line with the goals of a nominal type system. Without a name provided by the programmer, a structure can potentially be mapped to many un-related (w.r.t. subtyping) named types. However, the explicit type given to `new` need not be exactly the same type as the following structure. It is only required that the structure is proper structural subtype to the exposed type. This differing view of the same object provides an easy way to abstract the types and other members of the `new`'ed object. The object definition is checked for well-formedness by ensuring the type signature of its members form a structural type that is a subtype of the declared type of the object. Note that even though a self-variable is allowed in the object definition, it is defined to not be in scope for the definition of type members. This ensures there are no looping dependencies among type member definitions. Fields and methods are typechecked with the self-variable given the type of the declared object type plus any additional refinements made implicit by the type member definitions in the actual object definition.

The `let` expressions allow giving arbitrary terms an alias variable. The variable takes on the type of the aliased expression in the prior context without x itself. The rest of the expression is typed with x included.

Conditional `if` expressions simply type into the type provided by the programmer. The type is required to be a supertype of the type of each branch.

3.5.1 Typing Decidability

Term typing ($\Delta\Sigma\Gamma S \vdash e : \tau$) is heavily reliant on path typing, which judges the type that a path represents. In order to find out the type of a path, we need to look at the type of the object whose `val` field is being accessed. To understand the type of this inner object, which is also a path, we need to recursively apply the path typing procedure until we get down to a single variable. However, since the type of a variable can be a path-dependent type, we need to find out the underlying named structure it is based on in order to know the type of its fields. This is reliant on continually following the upper bound of a path-dependent type until a name is reached (This is possible because well-formed programs can only access members of objects whose type is upper bounded, since otherwise we are unable to know if it even has the desired member). The type expansion rules ($\Delta\Sigma\Gamma S \vdash \tau \prec \tau$) capture this process. Therefore, as we back out of the recursive path typing judgments, we repeatedly apply type expansion to find out the type of each successive field access.

To understand why this process always terminates, we first look at how variables depend on each other in the dynamic context.

Theorem 1. *Each record $x : \tau$ in Γ can only mention the variables that were added to Γ before it.*

Proof. There are only two rules that add to Γ during term typing: `T-LET` and `T-NEW`.

- Case `T-LET`: For each "let" expression **`let`** $x = e_x$ **`in`** e, the rule adds to the context $x : \tau_e$ when typing e, where τ_e is the type of the enclosed expression e_x. Since e_x is typed under Γ, its type τ_e can only refer to variables in Γ, which does not yet contain x.

- Case `T-NEW`: For each "new" expression **`new`** $\tau\{x \Rightarrow \overline{d}\}$, for each method declaration **`def`** $f\ :\ \tau_x\ x \rightarrow \tau_r = e$, the rule adds to the context $x : \tau_x$ when typing e. τ_x cannot refer to x because x is not defined to be in scope for τ_x.

$\boxed{\Delta\Sigma\Gamma S \vdash e : \tau}$

$$\frac{\Gamma(x) = \tau}{\Delta\Sigma\Gamma S \vdash x : \tau}\ \text{T-VAR} \qquad \frac{S(l) = \tau}{\Delta\Sigma\Gamma S \vdash l : \tau}\ \text{T-LOC} \qquad \frac{\Delta\Sigma\Gamma S \vdash p : \tau \quad \Delta\Sigma\Gamma S \vdash \tau \prec n\,r \quad \Delta\Sigma\Gamma S \vdash n\,r \ni_x \texttt{val}\ v\ :\ \tau_v}{\Delta\Sigma\Gamma S \vdash p.v : [p/x]\tau_v}\ \text{T-SEL}$$

$$\frac{\begin{array}{c}\Delta\Sigma\Gamma S \vdash p : \tau \qquad \Delta\Sigma\Gamma S \vdash \tau \prec n\,r \\ \Delta\Sigma\Gamma S \vdash n\,r \ni_x \texttt{def}\ f\ :\ \tau_a\ x_a \to \tau_r \\ \Delta\Sigma\Gamma S \vdash p' : \tau' \qquad \Delta\Sigma\Gamma S \vdash \tau' <:\ [p/x]\tau_a\end{array}}{\Delta\Sigma\Gamma S \vdash p.f(p') : [p, p_a/x, x_a]\tau_r}\ \text{T-APP} \qquad \frac{\Delta\Sigma\Gamma S \vdash \tau\ \text{wf} \quad \Delta\Sigma\Gamma S \vdash \tau\{x \Rightarrow \overline{d}\}\ \text{wf}}{\Delta\Sigma\Gamma S \vdash \texttt{new}\ \tau\{x \Rightarrow \overline{d}\} : \tau}\ \text{T-NEW}$$

$$\frac{\begin{array}{c}\Delta\Sigma\Gamma S \vdash e_x : \tau_x \\ \Delta\Sigma\Gamma, x : \tau_x\, S \vdash e : \tau\end{array}}{\Delta\Sigma\Gamma S \vdash \texttt{let}\ x = e_x\ \texttt{in}\ e : \tau}\ \text{T-LET} \qquad \frac{\begin{array}{c}\Delta\Sigma\Gamma S \vdash \tau\ \text{wf} \\ \Delta\Sigma\Gamma S \vdash e_1 : \tau_1 \qquad \Delta\Sigma\Gamma S \vdash \tau_1 <:\ \tau \\ \Delta\Sigma\Gamma S \vdash e_2 : \tau_2 \qquad \Delta\Sigma\Gamma S \vdash \tau_2 <:\ \tau\end{array}}{\Delta\Sigma\Gamma S \vdash \texttt{if}[\tau]\ p_1 = p_2\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \tau}\ \text{T-IF}$$

$\boxed{\Delta\Sigma\Gamma S \vdash \tau\,\{x \Rightarrow \overline{d}\}\ \text{wf}}$

$$\frac{\begin{array}{c}\Delta\Sigma\Gamma S \vdash \tau \prec n\,r \qquad \Delta(n) = \{x_n \Rightarrow \overline{\sigma_n}\} \qquad r = \{\overline{\delta}\} \qquad \overline{\sigma_\tau} = \overline{\sigma_n} +_\sigma \overline{\delta} \\ \tau_x = n\,(r +_r \text{ref}(\text{sig}(\overline{d}))) \qquad \Gamma' = \Gamma, x : \tau_x \qquad \Delta\Sigma\Gamma' S \vdash \text{sig}(\overline{d}) <:\ [x/x_n]\overline{\sigma_\tau} \\ \forall\ \texttt{val}\ v\ :\ \tau_v\ =\ e \in \overline{d}.\ \Delta\Sigma\Gamma'\, S \vdash e : \tau_v \\ \forall\ \texttt{def}\ f\ :\ \tau_a\ x_a \to \tau_r = e \in \overline{d}.\ \Delta\Sigma\Gamma', x_a : \tau_a\, S \vdash e : \tau_r\end{array}}{\Delta\Sigma\Gamma S \vdash \tau\,\{x \Rightarrow \overline{d}\}\ \text{wf}}$$

$\boxed{\Delta\Sigma\Gamma S \vdash \tau\ \text{wf}}$

$$\frac{}{\Delta\Sigma\Gamma S \vdash \top\ \text{wf}} \qquad \frac{}{\Delta\Sigma\Gamma S \vdash \bot\ \text{wf}}$$

$$\frac{\begin{array}{c}\Delta\Sigma\Gamma S \vdash \beta\, r_\beta \prec n\, r_n \qquad \Delta(n) = \{x_n \Rightarrow \overline{\sigma_n}\} \qquad r_\beta = \{\overline{\delta}\} \\ \forall\ \texttt{type}\ t\ B\ \tau \in \delta.\Delta\Sigma\Gamma S \vdash n\, r_n \ni_x \texttt{type}\ t\ B'\ \tau'\ \text{and} \\ \Delta\Sigma\Gamma, x_n : \beta\, r_\beta S \vdash \texttt{type}\ t\ B\ \tau <:\ \texttt{type}\ t\ B'\ \tau'\end{array}}{\Delta\Sigma\Gamma S \vdash \beta\, r_\beta\ \text{wf}}$$

where $+_\sigma$ and $+_r$ are merge operations on $\overline{\sigma}$ and refinements, respectively (On conflict, RHS is preserved and LHS is discarded).
sig : $\overline{d} \to \overline{\sigma}$ transforms object member definitions into member declarations by removing the dynamic expression part of vals and funs.
ref : $\overline{\sigma} \to r$ filters member declarations by preserving only type member declarations.
$\Delta\Sigma\Gamma S \vdash \tau \prec \tau_u$ is true if following the upper bound of τ leads to τ_u, whose upper bound is itself. Judgments not formally defined here are defined with the subtyping rules.

Figure 3.4: Nominal Wyvern Term Typing

This means every successive member of Γ can only refer to the variables that were added before it. $\square$

To make referring to positions easier, define the position of a variable x in Γ as its rank (The first variable has rank 1). Intuitively, this means the first variable can only refer to static types (names, top, or bottom), and higher ranked variables can refer to path-dependent types whose paths are rooted at the lower ranked variables (the variable that begins a path is denoted as the root of the path, and the rank of a path is defined as the rank of its root). This hints at the possibility that as path typing progresses, its range of reference decreases, and derivation eventually stops. For the following theorems, we define the length of a path p (written $|p|$) as the number of field accesses it has. A single variable has length 0, and $x.v_1.v_2.\cdots.v_m$ has length m.

Theorem 2. *$\Delta\Sigma\Gamma S \vdash p : \tau_p$ is decidable for all path p.*

Theorem 3. *$\Delta\Sigma\Gamma S \vdash p.t\,r \prec \tau$ is decidable for all path-dependent type $p.t\,r$.*

Since the path-typing and type expansion are mutually recursive, the two theorems are proved at the same time with a nested induction proof.

Proof. The two theorems are combined and strengthened to get the following statement.
$P_{\Delta\Sigma\Gamma S}(p.t\,r)$: let x_p be the root of path p. $\Delta\Sigma\Gamma S \vdash p.t\,r \prec \tau$ and $\Delta\Sigma\Gamma S \vdash p : \tau_p$, and τ_p can only contain either static types, path-dependent types whose path has a lower rank than p, or path-dependent types rooted at x_p whose length is less than $|p|$.
We wish to show: P(p) holds for all p.
Prove by induction on the rank of p.
Base Case: The rank of p is 1.

Prove by induction on the length of the longest x_p-rooted path in $p.t\,r$.
Base Case: The length of the longest x_p-rooted path is 0.

According to rule `T-VAR`, $\tau_p = \Gamma(x_p)$, which is easily decidable by looking up Γ.
Theorem 1 indicates τ_p cannot contain any path-dependent types. Therefore, it does not contain any x_p-rooted path at all.
To expand a $p.t\,r$, expansion rules first expand τ_p. Since τ_p only contains static types, its expansion is τ_p itself (rule `TE-NAME`). When accessing type member t from τ_p, if the bound is a lower bound, the judgment ends, so we only consider when t is

not just lower bounded. There are two cases (let x denote the self-variable from the membership judgment ($\Delta\Sigma\Gamma S \vdash \tau_p \ni_x \texttt{type}\ t \mathrel{\overset{=}{\in}} \beta_t\, r_t$)):

If t is in the refinement of τ_p: Then $\beta_t\, r_t$ can only contain static types or 0-length x_p-rooted paths. Therefore, the replacement $[p/x]$ has no effect. Combining two refinements does not create any new types, so the resulting type $\beta_t\, r_t +_r r_1$ has the same property (can only contain static types or 0-length x_p-rooted paths). If β_t is a name, the second expansion also terminates. If β_t is $x_p.t'$ (for some t'), then expansion recurses on this new path-dependent type. However, material-shape separation rules require there be no cyclic dependencies between type members unless it goes through a refinement (i.e. a labeled edge). Therefore, this limited recursion either ends when it accesses a type member that was refined, or when it naturally ends before going through all type members of p.

If t is not in the refinement (i.e. the bound of t comes from the name type definitions): The bound of t can only contain static types or 0-length x-rooted paths (Due to restriction on length of paths for type member bounds). The replacement $[p/x]$ replaces x with x_p. This means we get to the exact same situation as in the previous case when t is in the refinement.

Inductive Case: The length of the longest x_p-rooted path in $p.t\, r$ is l ($l > 1$). Let length of p be m ($m \leq l$). Denote p as $x.v_1.\cdots.v_m$.

Inductive Hypothesis: The statement $P_{\Delta\Sigma\Gamma S}()$ holds for all $p.t\, r$ whose longest x_p-rooted path (where x_p is root of p) is less than $|p|$, and p has rank 1 in Γ.

Let p' = $x.v_1.\cdots.v_{m-1}$.

Rule `T-SEL` depends on the typing of p'. The IH determines that $\Delta\Sigma\Gamma S \vdash p' : \tau_{p-1}$. Since τ_{p-1} does not contain x_p-rooted paths with length $\geq |p| - 1$, type expansion is decidable for τ_{p-1} according to IH.

Now consider when the field declaration of v_m is accessed in τ_{p-1}. Again, let x denote the self-variable. If the type of v_m in τ_{p-1} (denote as τ_v) does not refer to x, it will be static only. If it does refer to x, then any x-rooted paths will have 0 length. After performing $[p'/x]$, the longest x_p-rooted path in τ_v is still no longer than $m - 1$.

Next consider the expansion of $p.t\, r$. Expansion rules first expand the type of p, which was just showed to not contain any x_p-rooted paths longer than $m - 1$. According to IH, this expansion is decidable. The proof for accessing the type member

t is very similar to the proof in the base case: The bound of t (denote as $\beta_t\, r_t$) will either have a name type as base, in which case the second expansion terminates, or it is a x_p-rooted path with length equal to m, and r can only contain static types or x_p-rooted paths with length no greater than l. In the latter case, the recursion is with another type member of p, but this cannot happen infinitely due to material-shape separation.

Inductive Case: The rank of p is r ($r > 1$).

Inductive Hypothesis: The statement $P_{\Delta\Sigma\Gamma S}()$ holds for all $p.t\, r$ where p has rank less than r in Γ.

Prove by induction on the length of the longest x_p-rooted path in $p.t\, r$.

This part is omitted for brevity: Similar to the base case on rank, when the longest length is 0, the type of p can only refer to lower-ranked paths. By IH, they are all type-able and expand-able. When the longest length is l, it can refer to both lower-ranked and shorter paths. The key is always that the material-shape separation rules and the restriction on path-length of type member bounds prevent infinite cyclic expansion for any one particular path.

□

One additional lemma we are able to gain from this strengthened proof is that typing a path will either decrease the rank of the path or the length of the path. We can thus define a measure on paths.

Definition 6 (rank-length (RL))**.** *The rank-length measure of a path, p, given context Γ (written $RL_\Gamma(p)$, or just $RL(p)$) is the pair of natural numbers:*

$$(\mathit{rank}(p), \mathit{length}(p))$$

, where rank(p) and length(p) are the rank and length of p, respectively.
Ordering of RL follows dictionary ordering. Two RL measures $(i_r, i_l) < (i'_r, i'_l)$ iff $i_r < i'_r$, or $i_r = i'_r \wedge i_l < i'_l$. They are equal if both components are equal.

Lemma 1. *If $\Delta\Sigma\Gamma S \vdash p : \tau_p$, then for any path-dependent type $p'.t\, r$ in τ_p, $RL_\Gamma(p') < RL_\Gamma(p)$.*

In addition, as can be seen from the proof, given any type $p.t\, r$, if $\Delta\Sigma\Gamma S \vdash p.t\, r \prec \tau$, then the largest RL of any path in τ cannot be larger than the largest RL of any path in

$p.t\,r$. This is because at no point during expansion can a path type get longer. The only time a new path type is created is when the type bound came from a name type definition, in which case any self variable x (length 0) is replaced with p.

Lemma 2. *If $\Delta\Sigma\Gamma S \vdash p.t\,r \prec \tau$, then the largest RL of any path in τ cannot be larger than the largest RL of any path in $p.t\,r$.*

Chapter 4

Subtyping Decidability

4.1 Subtyping Judgments

Figures 4.1 and 4.2 present the subtyping judgments for Nominal Wyvern. They consist of three main parts:

1. *Nominal type subtyping*: subtyping between two named types.
2. *Member type subtyping*: subtyping between two type members.
3. *Structural subtyping*: subtyping between two structural types.

Nominal type subtyping follows the nominal subtyping graph (Definition 1). To check $n_1\, r_1 <: n_2\, r_2$, we first check if the two are related by the nominal subtype relation by finding a path from n_1 to n_2 in the nominal subtyping graph. If there is no such path, we can immediately conclude 'false'. Due to conditional subtyping, even if there are paths from n_1 to n_2, we must check if there is a path such that the refinement labeled on every edge of the path (the "conditions") are each satisfied by r_1. Finally, if we have a conditional path, we still have to check if the refinements in r_2 are still supertypes of the corresponding types in $n_1\, r_1$. It is possible that r_2 makes some type member of n_2 too specific for even $n_1\, r_1$. Listing 4.1 contains one example for each of these three cases.

$\boxed{\Delta\Sigma\Gamma S \vdash \tau <: \tau}$

$$\frac{}{\Delta\Sigma\Gamma S \vdash \tau <: \top}\ \text{S-TOP} \qquad \frac{}{\Delta\Sigma\Gamma S \vdash \bot <: \tau}\ \text{S-BOT}$$

$$\frac{\Delta\Sigma\Gamma S \vdash p : \tau_p \qquad \Delta\Sigma\Gamma S \vdash \tau_p \prec n\,r \qquad \Delta\Sigma\Gamma S \vdash n\,r \ni_x \mathbf{type}\ t \mathrel{\overset{=}{\supseteq}} \beta_t\,r_t \qquad \Delta\Sigma\Gamma S \vdash [p/x](\beta_t\,r_t +_r r_1) <: \tau_2}{\Delta\Sigma\Gamma S \vdash p.t\,r_1 <: \tau_2}\ \text{S-UPPER}$$

$$\frac{\Delta\Sigma\Gamma S \vdash p : \tau_p \qquad \Delta\Sigma\Gamma S \vdash \tau_p \prec n\,r \qquad \Delta\Sigma\Gamma S \vdash n\,r \ni_x \mathbf{type}\ t \mathrel{\overset{=}{\subseteq}} \beta_t\,r_t \qquad \Delta\Sigma\Gamma S \vdash \tau_1 <: [p/x](\beta_t\,r_t +_r r_2)}{\Delta\Sigma\Gamma S \vdash \tau_1 <: p.t\,r_2}\ \text{S-LOWER}$$

$$\frac{\Delta\Sigma\Gamma S \vdash r_1 <: r_2}{\Delta\Sigma\Gamma S \vdash p.t\,r_1 <: p.t\,r_2}\ \text{S-STRUCT}$$

$$\frac{\Delta\Sigma\Gamma S \vdash n_1 \xrightarrow{r_1} n_2 \qquad \Delta\Sigma\Gamma S \vdash r_1 <: r_2}{\Delta\Sigma\Gamma S \vdash n_1\,r_1 <: n_2\,r_2}\ \text{S-NAME}$$

$\boxed{\Delta\Sigma\Gamma S \vdash \tau \prec \tau}$

$$\frac{}{\Delta\Sigma\Gamma S \vdash \top \prec \top}\ \text{TE-TOP} \qquad \frac{}{\Delta\Sigma\Gamma S \vdash \bot \prec \bot}\ \text{TE-BOT}$$

$$\frac{}{\Delta\Sigma\Gamma S \vdash n\,r \prec n\,r}\ \text{TE-NAME}$$

$$\frac{\Delta\Sigma\Gamma S \vdash p : \tau_p \qquad \Delta\Sigma\Gamma S \vdash \tau_p \prec n\,r \qquad \Delta\Sigma\Gamma S \vdash n\,r \ni_x \mathbf{type}\ t \mathrel{\overset{=}{\supseteq}} \beta_t\,r_t \qquad \Delta\Sigma\Gamma S \vdash [p/x](\beta_t\,r_t +_r r_1) \prec \tau_2}{\Delta\Sigma\Gamma S \vdash p.t\,r_1 \prec \tau_2}\ \text{TE-UPPER}$$

$$\frac{\Delta\Sigma\Gamma S \vdash p : \tau_p \qquad \Delta\Sigma\Gamma S \vdash \tau_p \prec n\,r \qquad \Delta\Sigma\Gamma S \vdash n\,r \ni_x \mathbf{type}\ t \ \geq\ \tau_t}{\Delta\Sigma\Gamma S \vdash p.t\,r_1 \prec p.t\,r_1}\ \text{TE-LOWER}$$

where $\mathbf{type}\ t \overset{B_1}{B_2}\ \tau$ *matches a type member declaration with either bounds* B_1 *or* B_2*.*

Figure 4.1: Nominal Wyvern Subtyping

$\boxed{\Delta\Sigma\Gamma S \vdash r <: r}$

$$\frac{}{\Delta\Sigma\Gamma S \vdash r <: \{\}}\ \text{SR-EMPTY} \qquad \frac{\begin{array}{c} r_1 \ni \mathbf{type}\ t\ B_1\ \tau_1 \\ \Delta\Sigma\Gamma S \vdash \mathbf{type}\ t\ B_1\ \tau_1 <: \mathbf{type}\ t\ B_2\ \tau_2 \\ \Delta\Sigma\Gamma S \vdash r_1 <: \{\overline{\delta_2}\} \end{array}}{\Delta\Sigma\Gamma S \vdash r_1 <: \{\overline{\delta_2}, \mathbf{type}\ t\ B_2\ \tau_2\}}\ \text{SR-CONS}$$

$\boxed{\Delta\Sigma\Gamma S \vdash \sigma <: \sigma}$

$$\frac{\Delta\Sigma\Gamma S \vdash \tau_1 <: \tau_2}{\Delta\Sigma\Gamma S \vdash \mathbf{type}\ t \leqq \tau_1 <: \mathbf{type}\ t\ \leq\ \tau_2}\ \text{SS-UPPER}$$

$$\frac{\Delta\Sigma\Gamma S \vdash \tau_2 <: \tau_1}{\Delta\Sigma\Gamma S \vdash \mathbf{type}\ t \geqq \tau_1 <: \mathbf{type}\ t\ \geq\ \tau_2}\ \text{SS-LOWER}$$

$$\frac{\Delta\Sigma\Gamma S \vdash \tau_1 <: \tau_2 \qquad \Delta\Sigma\Gamma S \vdash \tau_2 <: \tau_1}{\Delta\Sigma\Gamma S \vdash \mathbf{type}\ t\ =\ \tau_1 <: \mathbf{type}\ t\ =\ \tau_2}\ \text{SS-EXACT}$$

$$\frac{\Delta\Sigma\Gamma S \vdash \tau_1 <: \tau_2}{\Delta\Sigma\Gamma S \vdash \mathbf{val}\ v\ :\ \tau_1 <: \mathbf{val}\ v\ :\ \tau_2}\ \text{SS-VAL}$$

$$\frac{\Delta\Sigma\Gamma S \vdash \tau_{a2} <: \tau_{a1} \qquad \Delta\Sigma\Gamma, x_1 : \tau_{a2}\, S \vdash \tau_{r1} <: [x_1/x_2]\tau_{r2}}{\Delta\Sigma\Gamma S \vdash \mathbf{def}\ f\ :\ \tau_{a1}\ x_1 \to \tau_{r1} <: \mathbf{def}\ f\ :\ \tau_{a2}\ x_2 \to \tau_{r2}}\ \text{SS-DEF}$$

$\boxed{\Delta\Sigma\Gamma S \vdash n \xrightarrow{r} n}$

$$\frac{}{\Delta\Sigma\Gamma S \vdash n \xrightarrow{r} n}\ \text{SN-REFL}$$

$$\frac{\Sigma \ni n_1\, r_1 <: n_2 \qquad \Delta\Sigma\Gamma S \vdash r_1' <: r_1 \qquad \Delta\Sigma\Gamma S \vdash n_2 \xrightarrow{r_1'} n_3}{\Delta\Sigma\Gamma S \vdash n_1 \xrightarrow{r_1'} n_3}\ \text{SN-TRANS}$$

$\boxed{\Delta\Sigma\Gamma S \vdash n\, r \ni_x \sigma}$

$$\frac{\sigma \in \overline{\delta} \qquad x \notin \Gamma}{\Delta\Sigma\Gamma S \vdash n\,\{\overline{\delta}\} \ni_x \sigma}\ \text{M-REF} \qquad \frac{\sigma \notin \overline{\delta} \qquad \Delta(n) = \{x \Rightarrow \overline{\sigma_n}\} \qquad \sigma \in \overline{\sigma_n}}{\Delta\Sigma\Gamma S \vdash n\,\{\overline{\delta}\} \ni_x \sigma}\ \text{M-NAME}$$

$\sigma \in \overline{\delta}$ is true if σ is a type member declaration (i.e. δ), and is part of $\overline{\delta}$.
$x \notin \Gamma$ is true when x is a fresh variable under the current variable typing context.

Figure 4.2: Nominal Wyvern Subtyping (continued)

```
name A {z => ...}
name B {z => ...}
name C {z => ...}
subtype C <: B
subtype B <: A

name N1 {z => type t <= ⊤}
name N2 {z => type t <= A}
name N3 {z => type t <= A}
subtype N1 {type t <= B} <: N2

/* Query 1: N1 {type t <= B} <: N3
 *                                    --> false, no path in graph
 * Query 2: N1 {type t <= A} <: N2
 *                                    --> false, condition not met
 * Query 3: N1 {type t <= B} <: N2 {type t <= C}
 *                                    --> false, r_2 too specific
 */
```

Listing 4.1: 3 ways nominal subtyping can fail

Member type subtyping is similar to DOT. Follow the upper bound for LHS base types, and follow the lower bound for RHS base types. Existing refinements are merged with new refinements by discarding the new ones on conflict. Reflexivity applies for when base types on both sides are exactly the same path, in which case structural subtyping applies to the refinements. If any one side's base type becomes a name type, it waits for the other side to also reduce into a name type, at which point nominal type subtyping applies. If both sides gets stuck without reflexivity or nominal subtyping applying, the query concludes to 'false'.

Structural subtyping follows standard subtyping on record types. Width subtyping allows LHS to contain more members than the RHS. Depth subtyping allows the bounds on the LHS to be more specific than the corresponding bounds on the RHS.

- For *type members*, the LHS type bound must be in the same direction (or be an exact bound) as the RHS, and be no less specific than the RHS
- For *field members*, the LHS type of the val must be a subtype of the RHS type
- For *method members*, first replace the argument variable so that both sides use the

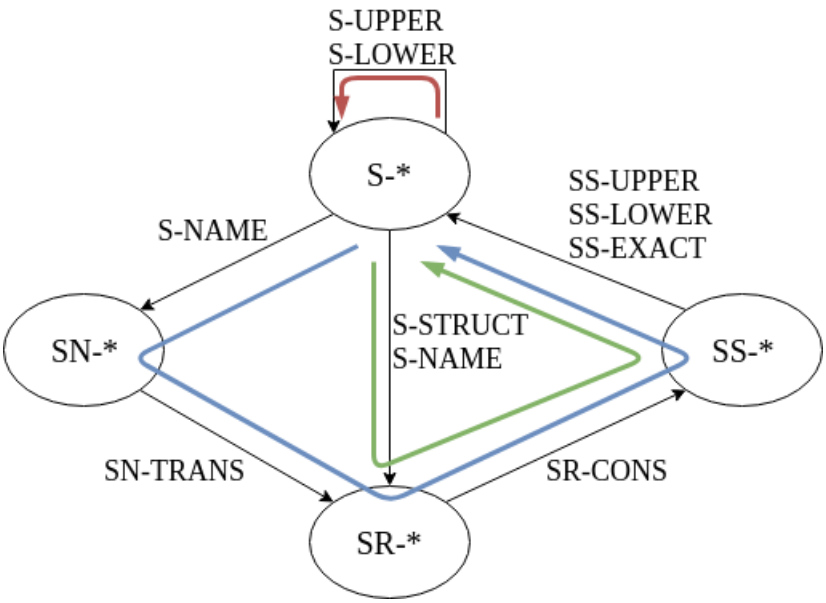


Figure 4.3: Ways to get recursive subtyping judgments

same name. The argument type of LHS must be a supertype of the RHS, and the result type must be a subtype of RHS.

For all subsequent subtype queries, the self variable (and the argument variable in the case of method) are added to the single context with the LHS parent type.

In all three parts, whenever a bottom type appears on the LHS, or a top type appears on the RHS, derivation immediately returns 'true'.

4.2 Decidability

Observe from the subtyping rules the only ways nested subtype judgments can occur in a derivation tree. Figure 4.3 shows the ways one subtype judgment (any `S-*` rule) can call back to a subtype judgment. Each node represents one family of subtype judgment rules (based on the prefix of the rule names in Figure 4.1): `SN-*` rules judge subtype relations between named types; `SR-*` rules judge structural subtyping relations between refinements; `SS-*` rules judge subtyping relations between individual refinement member definitions. Each edge (black arrow) represents a possibility of a rule from one rule family calling into another family, labeled with the inducing rule names. Note that there are two

outgoing edges from S−⋆ labeled with S−NAME. This is because the S−NAME rule both directly calls into SR−⋆ and calls SR−⋆ as well inside SN−TRANS. The three colored paths cover the three general ways recursion can occur:

- [RED] S−UPPER or S−LOWER: called directly.
- [GREEN] S−STRUCT or S−NAME directly: called directly via structural subtyping rules, specifically SS−LOWER or SS−UPPER.
- [BLUE] S−NAME indirectly: called while checking conditional subtyping between names in the nominal subtyping graph.

All the potential ways of getting nested subtype judgments can be partitioned into the following cases (the name of each case is given in parentheses):

1. [RED] via S−UPPER (RSU): The new LHS is the upper bound of the old LHS type. RHS stays the same.
2. [RED] via S−LOWER (RSL): The new RHS is the lower bound of the old RHS type. LHS stays the same.
3. [GREEN] via S−STRUCT (GSS): The new type on each side comes from inside the refinement of each side. (Left and right may be swapped if path also went through SS−LOWER or SS−EXACT).
4. [GREEN] via S−NAME (GSN): The new RHS comes from inside the old RHS refinement. The new LHS comes from either the old LHS refinement or a type bound in the definition of the LHS named type. (Left and right may be swapped if path also went through SS−LOWER or SS−EXACT).
5. [BLUE] (BSN): The new LHS comes from inside the old LHS refinement. The new RHS comes from the refinement labeled on an edge between the old LHS named type and old RHS named type in the nominal subtyping graph. (Left and right may be swapped if path also went through SS−LOWER or SS−EXACT).

To prove decidability, we define the notion of a "lineage" in the context of a subtype derivation. The idea is that given a subtype query ($\Delta\Sigma\Gamma S \vdash \tau <: \tau$), a lineage $\mathcal{L}$ captures the trace a type goes through during the subtype derivation that starts with that

initial query. Concretely, a lineage is a tree with types as nodes. The shape of the tree corresponds exactly to the derivation tree of the initial subtype query. Each subtyping derivation creates two lineages: an initial left-lineage rooted at the initial LHS type, and an initial right-lineage rooted at the initial RHS type. For each nested subtyping judgment in the derivation tree, the inner judgment's (deeper in tree) LHS and RHS types are linked to the outer judgment's types depending on which rules were used between the inner and outer subtype judgments. In most cases, the inner type is added as a child of the outer type of the same side. However, if the judgments involve a `SS-LOWER` or `SS-EXACT` (i.e. the bound on the type in the structure involved a lower bound), the inner LHS type is added as a child of the outer RHS type ("the left lineage swings to the right"), and the inner RHS links to the outer LHS. This is the only case when the two lineages swap sides.

Definition 7 (Lineage)**.** *Given a subtype derivation tree rooted at $\Delta\Sigma\Gamma S \vdash \tau_{initl} <: \tau_{initr}$, the two lineages of the derivation tree are each a tree. Each subtype judgment in the derivation tree is given a label, and for each subtype judgment J: $\Delta\Sigma\Gamma S \vdash \tau_l <: \tau_r$, two vertices are created: $J\#\tau_l$ and $J\#\tau_r$. For each pair of judgments J_1: $\Delta\Sigma\Gamma S \vdash \tau_{l1} <: \tau_{r1}$ and J_2: $\Delta\Sigma\Gamma S \vdash \tau_{l2} <: \tau_{r2}$ such that J_1 is the closest ancestor of J_2 that is a subtype judgment `S-*`, denote new sets of edges:*

- *Covariant edges: $J_1\#\tau_{l1} \to J_2\#\tau_{l2}, J_1\#\tau_{r1} \to J_2\#\tau_{r2}$*
- *Contravariant edges: $J_1\#\tau_{l1} \to J_2\#\tau_{r2}, J_1\#\tau_{r1} \to J_2\#\tau_{l2}$*

, with each edge labeled with how the recursive call was made (one of RSU, RSL, GSS, GSN, BSN). Then generate edges depending on the path from J_1 to J_2:

- *If J_1 calls to J_2 via a `SS-EXACT`, add both contravariant and covariant edges*
- *If J_1 calls to J_2 via a `SS-LOWER`, add contravariant edges*
- *Otherwise, add covariant edges*

The initial left-lineage and right-lineage are the trees rooted at τ_{initl} and τ_{initr}, respectively.

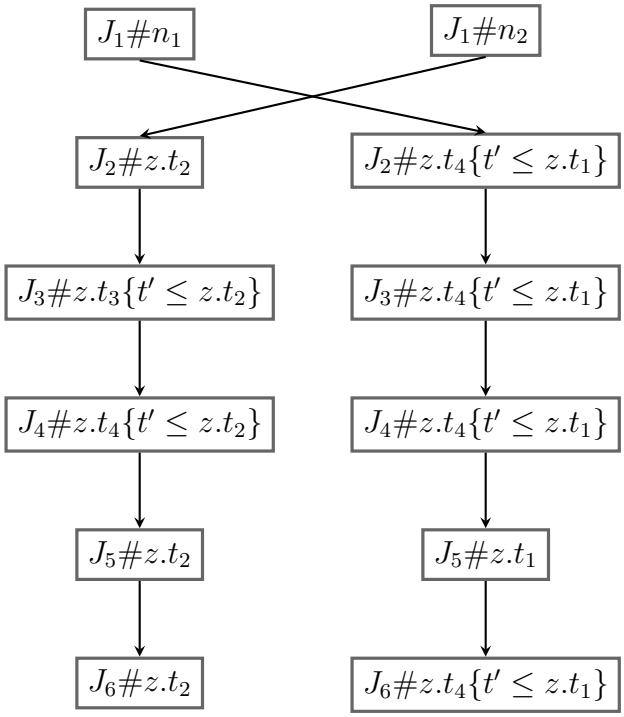
Example 5. *Consider the following partial derivation (judgments are labeled on the right):*

$$
\begin{array}{lcrcll}
\emptyset & \vdash & & n_1 & <: \; n_2 & [J_1] \\
z : n_1 & \vdash & & z.t_2 & <: \; z.t_4\{t' \leq z.t_1\} & [J_2] \\
z : n_1 & \vdash & z.t_3\{t' \leq z.t_2\} & <: & z.t_4\{t' \leq z.t_1\} & [J_3] \\
z : n_1 & \vdash & z.t_4\{t' \leq z.t_2\} & <: & z.t_4\{t' \leq z.t_1\} & [J_4] \\
z : n_1 & \vdash & & z.t_2 & <: \; z.t_1 & [J_5] \\
z : n_1 & \vdash & & z.t_2 & <: \; z.t_4\{t' \leq z.t_1\} & [J_6] \\
 & & & & \ldots &
\end{array}
$$

The setup for the derivation is shown below on the left. The two lineages are the two trees (each with only one branch) on the right.

$$
n_1 = \left\{ z \Rightarrow \begin{array}{l} t_1 \geq z.t_4\{t' \leq z.t_1\} \\ t_2 \leq z.t_3\{t' \leq z.t_2\} \\ t_3 \leq z.t_4 \\ t_4 \leq n_2 \\ t' \leq \top \end{array} \right\}
$$

$$
n_2 = \left\{ z \Rightarrow \begin{array}{l} t_1 \geq z.t_2 \\ t_2 \leq \top \\ t' \leq \top \end{array} \right\}
$$



A lineage captures the relation between types in recursively dependent subtyping judgments. As long as all paths in a lineage (from the root downwards) are finite, the entire corresponding subtype derivation is finite. We can consider any path starting from the root of a lineage as made up of many segments, divided by edges labeled with `S-NAME` recursions (i.e. GSN or BSN). Below, we first study the behavior within a segment, and then extend to across segments.

To prove decidability, we define a measure $\mathcal{E}$ on types that will decrease during deriva-

tion. To define $\mathcal{E}$, we first define two measures, $\mathcal{M}$ and $\mathcal{A}$, on type members of a given name type n. $\mathcal{M}(n::t)$ captures the number of other type members of n reachable from t. $\mathcal{A}(n::t)$ captures the other dependencies of t.

Definition 8 ($\mathcal{M}$ and $\mathcal{A}$ measures of pseudotypes)**.** *Given a name type definition $n : \{x \Rightarrow \overline{\sigma}\} \in \Delta$ and the subtype dependency graph derived from $\Delta\Sigma$, the $\mathcal{M}_{\Delta\Sigma}$ and $\mathcal{A}_{\Delta\Sigma}$ measures of a pseudotype $n::t$ is defined as:*

$$\frac{\texttt{type}\ t\ B\ \tau \in \overline{\sigma} \qquad T = \{n::t' | x.t' \in \tau \wedge n::t \rightarrow_B n::t'\}}{\mathcal{M}_{\Delta\Sigma}(n::t) = 1 + \sum_{x.t' \in T} \mathcal{M}_{\Delta\Sigma}(n::t')}$$

$$\frac{\texttt{type}\ t\ B\ \tau \in \overline{\sigma} \qquad T = \{n::t' | x.t' \in \tau \wedge n::t \rightarrow_B n::t'\}}{\mathcal{A}_{\Delta\Sigma}(n::t) = 1 + \sum_{x.t' \in T} \mathcal{A}_{\Delta\Sigma}(n::t') + \sum_{n' \in \tau} \mathcal{E}_{\Delta\Sigma\Gamma S}(n')}$$

where $n::t \rightarrow_B n::t'$ is true if there is a path in the subtype dependency graph from $n::t$ to $n::t'$ that only consists of edges whose variance is B and are not labeled with shapes, and the path does not consist of nodes that are not pseudotypes of n.
$\mathcal{E}_{\Delta\Sigma\Gamma S}()$ is the measure on types to be defined below.

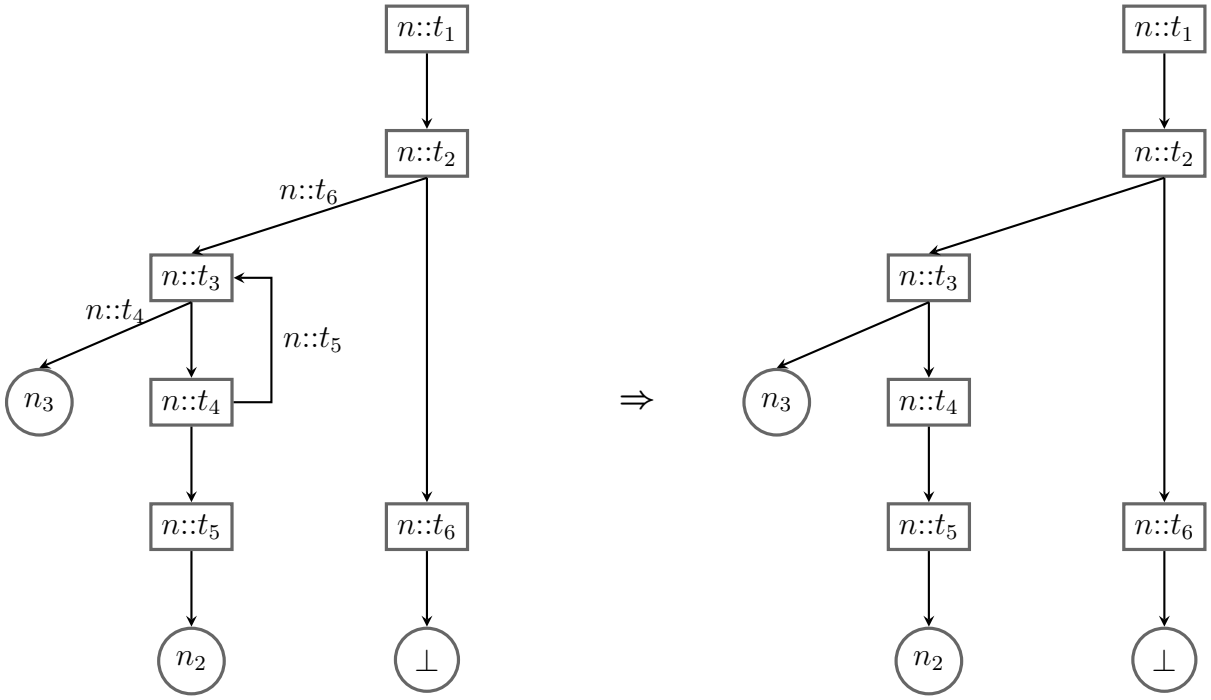
The $\mathcal{M}$ and $\mathcal{A}$ measures are computable thanks to the shape-material separation rules. Since there are no cycles without going through labeled edges, a topological ordering based on dependency exists on all type members of a given name type. Both measurements can be computed "bottom-up" from this ordering.

Example 6. *Consider the following setup:*

$$n = \left\{ z \Rightarrow \begin{array}{l} t_1 \geq z.t_2 \\ t_2 \geq z.t_6\{t' \leq z.t_3\} \\ t_3 \leq z.t_4\{t'' \leq n_3\} \\ t_4 \leq z.t_5\{t' \leq z.t_3\} \\ t_5 \leq n_2 \\ t_6 \geq \bot \end{array} \right\}$$

The partial subtype dependency graph generated from the type members of named type n is the graph below on the left. It shows that there is only one shape pseudotype, $n::t_5$. Since the definition of n follows material-shape separation rules, if we remove all dependencies

that go through shapes, we get the graph on the right, which is guaranteed to be acyclic (the edge labels in the right hand side graph are omitted since they are not needed for the $\mathcal{M}$ and $\mathcal{A}$ measures).



From the right hand side graph, we can calculate the $\mathcal{M}$ and $\mathcal{A}$ measures for each pseudotype of n "bottom-up":

	$\mathcal{M}$	$\mathcal{A}$
$n::t_1$	$\mathcal{M}(n::t_2)+1$ $=6$	$\mathcal{A}(n::t_2)+1$ $=4+\mathcal{E}(n_2)+\mathcal{E}(n_3)+\mathcal{E}(\perp)$
$n::t_2$	$\mathcal{M}(n::t_3)+\mathcal{M}(n::t_6)+1$ $=5$	$\mathcal{A}(n::t_3)+\mathcal{A}(n::t_6)+1$ $=3+\mathcal{E}(n_2)+\mathcal{E}(n_3)+\mathcal{E}(\perp)$
$n::t_3$	$\mathcal{M}(n::t_4)+1$ $=3$	$\mathcal{A}(n::t_4)+\mathcal{E}(n_3)+1$ $=2+\mathcal{E}(n_2)+\mathcal{E}(n_3)$
$n::t_4$	$\mathcal{M}(n::t_5)+1$ $=2$	$\mathcal{A}(n::t_5)+1$ $=1+\mathcal{E}(n_2)$
$n::t_5$	1	$\mathcal{E}(n_2)$
$n::t_6$	1	$\mathcal{E}(\perp)$

As evident from the example, for any non-shape dependency (i.e. any edge in the right

hand side dependency graph above), both $\mathcal{M}$ and $\mathcal{A}$ are strictly larger for the source node than for the destination node. Moreover, if the source node has multiple non-shape dependencies, the $\mathcal{M}$ value of the source node will be strictly larger than the sum of the $\mathcal{M}$ values of all its out-neighbors (and the same holds for $\mathcal{A}$). This property of $\mathcal{M}$ and $\mathcal{A}$ will prove to be useful when used by the $\mathcal{E}$ measurement on types defined below.

We now wish to define the measure $\mathcal{E}$ on types. It can be thought of as the "potential energy" of a type, and that continued subtype derivation requires spending energy.

Definition 9 (Energy measure of a type)**.** *Given the contexts $\Delta\Sigma\Gamma S$, first define the energy $\mathcal{E}_{\Delta\Sigma\Gamma S}$ of a **base type** as:*

$$\frac{}{\mathcal{E}_{\Delta\Sigma\Gamma S}(\top) = 0} \qquad \frac{}{\mathcal{E}_{\Delta\Sigma\Gamma S}(\bot) = 0}$$

$$\frac{N = \{n_1 | n_1 r_1 <: \ n \in \Sigma\} \cup \{n' | n_1 r_1 <: \ n \in \Sigma, n' \in r_1\}}{\mathcal{E}_{\Delta\Sigma\Gamma S}(n) = \sum_{n' \in N} (\mathcal{E}_{\Delta\Sigma\Gamma S}(n')) + 1}$$

$$\frac{\Delta\Sigma\Gamma S \vdash p : \tau_p \qquad \Delta\Sigma\Gamma S \vdash \tau_p \prec n\, r}{\mathcal{E}_{\Delta\Sigma\Gamma S}(p.t) = \mathcal{E}_{\Delta\Sigma\Gamma S}(n\, r) \times \mathcal{M}_{\Delta\Sigma}(n{::}t) + \mathcal{A}_{\Delta\Sigma}(n{::}t)}$$

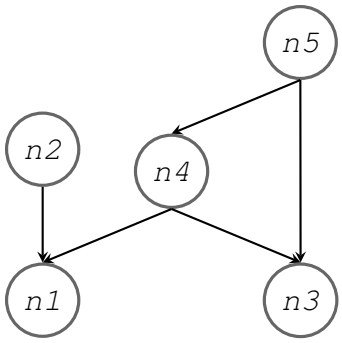
The energy $\mathcal{E}_{\Delta\Sigma\Gamma S}$ of a type τ is the sum of the energies of the base types that occur in τ (i.e. the nodes of the refinement tree of τ).

$$\mathcal{E}_{\Delta\Sigma\Gamma S}(\tau) = \sum_{\beta \in \tau} \mathcal{E}_{\Delta\Sigma\Gamma S}(\beta)$$

Example 7. *Consider the following top-level subtype declarations:*

```
subtype n1 <: n2
subtype n1 { type t' <= n3 } <: n4
subtype n3 { type t'' <= n4 } <: n5
```

The partial subtype dependency graph generated from the above declarations is shown below on the left (omitting edge labels). The energy measure of each name can be calculated from the bottom up (shown in the table on the right below).



	$\mathcal{E}$
$n1$	1
$n2$	$\mathcal{E}(n1)+1$ $=2$
$n3$	1
$n4$	$\mathcal{E}(n1)+\mathcal{E}(n3)+1$ $=3$
$n4$	$\mathcal{E}(n3)+\mathcal{E}(n4)+1$ $=5$

The material-shape separation requirements ensure that the energy measurement is defined on name types: The rules dictate a topological order over all named types, which make sure that it is always possible for the recursive energy calculation to terminate (as shown in Example 7). For the path-dependent types, lemmas 1 and 2 make sure that every recursive $\mathcal{E}$ call can only contain paths strictly smaller in terms of RL. This means recursion eventually all stops on static types.

The intuition behind the energy equation of path-dependent types is to make sure that even if one type member t depends on many other type members, the energy of $p.t$ for any p should be greater than the combined energy of all that t depends on. Due to the existence of cyclic dependencies through shapes, this is not always possible to compute. However, the definition of $\mathcal{M}$ and $\mathcal{A}$ takes this into account by ignoring any dependencies that are inside the refinement of a shape. This means during subtype derivation, when following the type bound of t, as long as the bound does not include any shapes, the energy before will be bigger than after. If there is a shape, we'll show that it is still fine.

First we wish to show that, within any segment of a lineage, the energy of nodes decrease when the lineage is on the right-hand side.

Theorem 4. *Given a subtype derivation under the contexts $\Delta\Sigma\Gamma S$ and one of its lineages $\mathcal{L}$, if $J_1\#\tau_1$ is the parent of $J_2\#\tau_2$ in the same segment in $\mathcal{L}$ and τ_2 is on the RHS of J_2, then $\mathcal{E}_{\Delta\Sigma\Gamma S}(\tau_2) < \mathcal{E}_{\Delta\Sigma\Gamma S}(\tau_1)$ (unless the edge between $J_1\#\tau_1$ and $J_2\#\tau_2$ is RSU, in which case the energy stays the same).*

Proof. Within a segment we only need to case on the following three recursion pathways:

- *RSU*: $\tau_2 = \tau_1$, and the energy stays the same.

- *RSL*: Let $\tau_1 = p.t\, r_1$. S-LOWER shows that $\Delta\Sigma\Gamma S \vdash p : \tau_p$, $\Delta\Sigma\Gamma S \vdash \tau_p \prec n\, r$, and $\Delta\Sigma\Gamma S \vdash n\, r \ni_x \texttt{type}\ t \geqq \beta_t\, r_t$. There are two places the bound $\beta_t\, r_t$ could come from:

  - Case - t is from r: The energy of $p.t$ already includes the energy from $n\, r$, which includes the energy of $\beta_t\, r_t$.
  - Case - t is from n: $\beta_t\, r_t$ came from Δ. The energy measure $\mathcal{E}_{\Delta\Sigma\Gamma S}([p/x]\beta_t\, r_t)$ is the sum of the energies of all types in $[p/x]\beta_t\, r_t$. Observe that since $n{::}t$ is lower bounded, any pseudotype that bound it are reachable from $n{::}t$ without going through shapes. Therefore, $\mathcal{M}_{\Delta\Sigma}(n{::}t) = \sum_{x.t' \in \beta_t\, r_t} \mathcal{M}_{\Delta\Sigma}(n{::}t')$, and $\mathcal{A}_{\Delta\Sigma}(n{::}t) = 1 + \sum_{x.t' \in \beta_t\, r_t} \mathcal{A}_{\Delta\Sigma}(n{::}t') + \sum_{n' \in \tau} \mathcal{E}_{\Delta\Sigma\Gamma S}(n')$. This means the energy-calculating equation of $p.t$ already subsumed all the energy from its bound.

  In all cases, $\mathcal{E}_{\Delta\Sigma\Gamma S}([p/x]\beta_t\, r_t)$ is no greater than the energy contributed by one part of $p.t\, r_1$. Therefore, total energy decreases for the RHS after RSL.

- *GSS*: τ_2 is a bound in the refinement part of tau_1, which means $\mathcal{E}_{\Delta\Sigma\Gamma S}(\tau_2)$ is strictly smaller since it contains strictly less types.

□

The same is true for the left-hand side, but since we allow certain cyclic dependencies to exist through shapes, the LHS energy can increase when introducing a shape.

Definition 10. *A path-dependent base type $p.t$ is considered a shape if $\Delta\Sigma\Gamma S \vdash p : \tau_p$, $\Delta\Sigma\Gamma S \vdash \tau_p \prec n\, r$, and $n{::}t$ is marked as a shape in the subtype dependency graph.*

Theorem 5. *Given a subtype derivation under the contexts $\Delta\Sigma\Gamma S$ and one of its lineages $\mathcal{L}$, if $J_1 \# \tau_1$ is the parent of $J_2 \# \tau_2$ in the same segment in $\mathcal{L}$ and τ_2 is on the LHS of J_2, then $\mathcal{E}_{\Delta\Sigma\Gamma S}(\tau_2) < \mathcal{E}_{\Delta\Sigma\Gamma S}(\tau_1)$ unless 1) τ_2 contains a shape, or 2) the edge from $J_1 \# \tau_1$ to $J_2 \# \tau_2$ is RSL, in which case the energy stays the same.*

Proof. Within a segment we only need to case on the following three recursion pathways:

- *RSL*: $\tau_2 = \tau_1$, and the energy stays the same.

- *RSU*: Let $\tau_1 = p.t\, r_1$. S−UPPER shows that $\Delta\Sigma\Gamma S \vdash p : \tau_p$, $\Delta\Sigma\Gamma S \vdash \tau_p \prec n\, r$, and $\Delta\Sigma\Gamma S \vdash n\, r \ni_x \mathbf{type}\ t \leqq \beta_t\, r_t$. There are two places the bound $\beta_t\, r_t$ could come from:

  - Case - t is from r: The energy of $p.t$ already includes the energy from $n\, r$, which includes the energy of $\beta_t\, r_t$.
  - Case - t is from n: $\beta_t\, r_t$ came from Δ. The energy measure $\mathcal{E}_{\Delta\Sigma\Gamma S}([p/x]\beta_t\, r_t)$ is the sum of the energies of all types in $[p/x]\beta_t\, r_t$.
    - If $[p/x]\beta_t\, r_t$ does not contain a shape: Similar to the RHS proof, if the bound does not contain any shapes, then all type members and names that are in this bound are already included in the energy calculation of $p.t$.
    - If $[p/x]\beta_t\, r_t$ contains a shape: Then the energy calculation of $p.t$ only included the energies from the types that are not refining the shape. If the shape does not refine anything, the energy of the entire $[p/x]\beta_t\, r_t$ is still covered by $p.t$, but if the shape is refined, the LHS may incur an increase in energy.

  In all cases other than when $[p/x]\beta_t\, r_t$ contains a refined shape, $\mathcal{E}_{\Delta\Sigma\Gamma S}([p/x]\beta_t\, r_t)$ is no greater than the energy contributed by one part of $p.t\, r_1$.

- *GSS*: τ_2 is a bound in the refinement part of tau_1, which means $\mathcal{E}_{\Delta\Sigma\Gamma S}(\tau_2)$ is strictly smaller since it contains strictly less types.

□

An energy increase on the LHS is fine because the proof shows that to do so, a path-dependent shape was encountered on the LHS, and it requires an S−STRUCT for both lineages in order to progress. However, this means the RHS base type was also a shape. Note that shapes rarely appear as base type on the RHS within a segment:

Theorem 6. *Given a subtype derivation under the contexts $\Delta\Sigma\Gamma S$ and one of its lineages $\mathcal{L}$, if $J_1\#\tau_1$ is the parent of $J_2\#\tau_2$ within a segment in $\mathcal{L}$ (and $\tau_1 \neq \tau_2$), and τ_2 is on the RHS of J_2, and τ_2 is a shape, then the edge from $J_1\#\tau_1$ to $J_2\#\tau_2$ must be GSS and τ_1 must also be on the RHS of J_1.*

Proof. Within a segment we only need to case on the following three recursion pathways:

- *RSU*: $\tau_2 = \tau_1$, the RHS type does not change.
- *RSL*: τ_2 is the base type of the lower bound of τ_1. However, since shapes can never be used after a lower bound, τ_2 is not a shape.
- *GSS*:
  - If recursion is via `SS-LOWER`: τ_2 is the base type of a lower bound in the refinement of τ_1. Therefore, τ_2 cannot be a shape.
  - If recursion is via `SS-UPPER`: τ_2 is the base type of an upper bound in the refinement of τ_1. It is possible that it is a shape.
  - If recursion is via `SS-EXACT`: both of the above will happen, though in two separate branches of the lineage (i.e. reduces to the above two cases).

□

As a result, the only shapes that can appear on the RHS as base types are the ones that were already in the refinement on the RHS. We know every time the LHS increases its energy by going through a shape, the RHS loses a shape. However, even though the LHS can replenish energy, the RHS cannot replenish its number of shapes. When the RHS runs out of shapes, it will no longer be able to aid in the energy increase of the LHS by performing `S-STRUCT`. Therefore, any infinite derivation cannot be entirely within one segment. Both lineages must eventually go through a `S-NAME`.

`S-NAME` can trigger two kinds of recursions: one through the green path (GSN), one through the blue path (BSN). Denote the types on the LHS and RHS of the judgment immediately before `S-NAME` as $n_1\, r_1$ and $n_2\, r_2$, respectively.

First, consider BSN recursions. BSN recursion is when `S-NAME` calls into `SN-TRANS`, which in turn recurses by replacing the RHS with types defined on the edges of the nominal subtyping graph. After BSN, the old left-lineage's energy decreases since we recursed on its refinement members (i.e. strictly smaller refinement tree). The old right-lineage turns into a completely new type, but it contains only static types. In fact, the energy of the old right-lineage also decreased due to the way energy is defined on named types. The old name type n_2 has a greater energy than the entire new RHS type.

BSN has a great impact on the right-lineage. It contains only static types, and the only shapes it contain are not refined (due to material-shape restrictions). This means the only

recursions it and its paired lineage can perform are `S-NAME` and `S-UPPER`. The lineage with the all-static types can never use `S-UPPER`, so it can never increase in energy at all within any lineage segment. As we will soon see, going through GSN decreases the energy for both lineages. We know that this is true for BSN as well. The result is the lineage with the all-static types will monotonically decrease its energy until its type becomes either top, bottom, a path-dependent type that is not lower-bounded, or a named type that is not declared to be the supertype of any other name type, all of which ends the derivation.

Now consider GSN recursions. GSN recursion is when `S-NAME` recursively checks subtyping between the commonly refined members of both sides. It behaves exactly like an `S-STRUCT`: There are no replenishing of energies or shapes on either side. Instead, the energy of both sides decrease because the recursion is on a proper subtree of the original type's refinement tree. This means the depth of the refinement trees of types on both sides strictly decrease every time recursion passes through GSN. Recursion continues as before, which means any infinite derivation will eventually visit `S-NAME` again. As shown earlier, the branch that goes through BSN is determined to end. The branch that goes through GSN again will decrease in depth. Since no available recursion paths (other than BSN) can increase the depth of any type, recursion eventually stops when there are no refinements on the RHS type and GSN no longer occurs.

Due to the combined result of energy decreasing, number of RHS shapes decreasing, and depth decreasing, no lineage can have infinite length. All subtype derivations must eventually terminate.

Chapter 5

Expressiveness

This chapter discusses the expressiveness of Nominal Wyvern by showing the encoding of some of the important features of both object-oriented languages and functional languages that we identified. Section 5.1 introduces the syntax sugar that will be used as shorthand in this section. Sections 5.2 through 5.4 shows how some of the common patterns expressible in DOT are encoded in Nominal Wyvern. Sections 5.5 and 5.6 show how Nominal Wyvern can be compatible with common patterns in both functional and object-oriented programming languages.

5.1 Syntax Sugar

For brevity, this chapter makes use of the following syntactic sugar for expressing common patterns.

- *Omitting self-variables*
  The self-variable that immediately follows the open curly brace of name declarations and new object definitions may be omitted if they are never referred to in the structure. Desugaring would just be adding a fresh identifier as the self-variable wherever it was omitted.

- *Omitting val types during object creation*
  In `new` expressions, the type annotation on `val` can be omitted if they are identical to the type of the `val` defined by the type (or its upper bound) that is being created.

This is always known during `new` because of the need to check well-formedness. Desugaring would just be adding back the required type of the `val`.

- *Inline member declarations*
  Members of named type definitions (σ), refinements (δ), and new object definitions (d) are allowed to be separated with commas to make it clearer to the reader when placed on a single line. This can be easily desugared by removing the commas.

- *Inline expressions*
  The abstract syntax requires the argument to methods to be pre-bound to a variable. This is relaxed to allow using any arbitrary expression directly as the argument to a method. The desugaring would be moving the expression into a `let` expression (with a fresh variable) that wraps the method call, and instead invoking the method with the fresh variable.

- *Multiple method arguments*
  The abstract syntax only allows one argument in method declarations. This is obviously not practical, but it is also not as limiting as it appears to be. Since all expressions are objects, multiple arguments are simply considered a new object that encapsulates them. For the examples shown in this chapter, the type of a function argument is allowed to be a structural type with a self-variable ($\{x \Rightarrow \overline{\sigma}\}$). Desugaring would be declaring the structure as a fresh name in the top level and then using that name in the method signature. Any references to the method's sibling type members are swapped for a fresh type member (bounded with $\top$ or $\bot$ as appropriate) in the argument type when moved outside. The use site subsequently refines the fresh name type with the referenced sibling type members.
  This serves as a useful feature for a concrete syntax since it does not break the theme of semantic separation of Nominal Wyvern: The arguments to a function should be considered to be defined by the function itself. It would therefore be perfectly fine if each method also declared a named type $\langle$`method_ID`$\rangle$`_arg`. Therefore, to spare the need to separately declare each such type, they are declared together with the method signatures.

- *Passing new objects to methods*
  Since the argument to a method can now be of an anonymous named type, it is impossible to `new` such an object. As a result, we introduce a syntax sugar that

allows the creation of a new object by simply defining a structure ($\{x \Rightarrow \overline{d}\}$) to be passed to any method whose argument type is an anonymous name. The desugaring for this is paired with the desugaring of the method signature: the anonymous named type is declared with a name, and the object is re-defined with a `new` expression using this name.

5.2 Basic Path-Dependent Types

Since Nominal Wyvern is based on DOT, the first example re-implements the Bank example of Listing 2.1 and Listing 2.2 in Nominal Wyvern to show that it preserves the abstraction boundaries and provides the same guarantees.

```
1  // assume a pre-defined String type. String objects are created
2  // directly with string literals.
3  name Unit {}
4
5  name CreditCard {}
6  name SecuredCard {}
7  subtype SecuredCard <: CreditCard
8  name AuthorizedUserCard {}
9  subtype AuthorizedUserCard <: CreditCard
10
11 name Bank {b =>
12   type Card
13   def applyForCard : String name -> b.Card
14   def payOff : b.Card c -> Unit
15 }
16
17 name RegionalBank {b =>
18   type Card <= SecuredCard
19   def applyForCard : String name -> b.Card
20   def payOff : b.Card c -> Unit
21 }
22 subtype RegionalBank <: Bank
23
24 name Utils {
25   def giveChildren : SecuredCard card -> Unit
26 }
```

```
27
28 let pnc = new Bank {b =>
29   type Card = CreditCard
30   def applyForCard : String name -> b.Card = new CreditCard {}
31   def payOff : b.Card c -> Unit = new Unit {}
32 } in
33 let veryCautiousBank = new RegionalBank {b =>
34   type Card = SecuredCard
35   def applyForCard : String name -> b.Card = new SecuredCard {}
36   def payOff : b.Card c -> Unit = new Unit {}
37 } in
38 // attempt to pay off
39 let myCard = veryCautiousBank.applyForCard("...") in
40 let legal = veryCautiousBank.payOff(myCard) in   // OK
41 let illegal = pnc.payOff(myCard) in              // type mismatch
42
43 let utils = new Utils {
44   def giveChildren : SecuredCard card -> Unit = new Unit {}
45 } in
46 // attempt to give children
47 let pass = utils.giveChildren(
48   veryCautiousBank.applyForCard("...")) in        // OK
49 let fail = utils.study(pnc.applyForCard("")) in   // type mismatch
50 ...
```

Listing 5.1: Basic path-dependent types in Nominal Wyvern

The types of credit cards are empty structures whose names give them meaning. Two types of banks are defined. Note that the `name` declarations are equivalent to interface declarations in other languages, therefore they do not contain implementation. Instead, the implementation is given to the particular object at creation time with `new`. [1]

The guarantees provided by DOT are preserved in Nominal Wyvern:

- A card can only be paid off at the bank that issued it: Each bank object `x` carries with it a type `x.Card` that is hidden from the outside. In this case, the object `myCard` has type `veryCautiousBank.Card` (derived directly from the signature of the `applyForCard()` method of `RegionalBank`, the type of `veryCautiousBank`). Therefore, it is incompatible with the input type of `pnc.payOff()`, which re-

[1] This difference is discussed more in depth in section 5.6.

quires the argument to be of the entirely opaque type `pnc.Card`.

- Only a secured card can be given to children: Even though the exact type of `veryCautiousBank.Card` is unknown, `RegionalBank` provides the hint that its card type is a subtype of `SecuredCard`. As a result, it fits the requirement of `util.giveChildren()`. In contrast, since `pnc.Card` is completely abstract, it is not safe to pass into `util.giveChildren()`.

Since Nominal Wyvern supports refinements, one can avoid the need to declare a specific bank type if the difference can be succinctly represented by a refinement (and preferably only when the new type is not used as a common concept in the code). For example, the veryCautiousBank can be written without a `RegionalBank`:

```
...
let veryCautiousBank = new Bank {type Card <= SecuredCard} {b =>
  type Card = SecuredCard
  def applyForCard : String name -> b.Card = new SecuredCard {}
  def payOff : b.Card c -> Unit = new Unit {}
} in
...
```

Listing 5.2: Expressing `RegionalBank` with a refinement

The underlined type declared in the `new` expression denotes the type that this new object has. The `Card` type is refined so that cards issued here can be given to children. Note that it does not need to match the exact type declared inside the definition. This allows a more fine-tuned approach to expressing the degree of abstraction desired.

5.3 F-Bounded Polymorphism

Recall from section 2.2 that one of the benefits of combining subtype polymorphism with parametric polymorphism is the ability to express F-bounded polymorphism.

5.3.1 Positive Recursion

Positive recursion refers to when the recursive type variable is at an "output" position, or covariant. For example, since the `clone()` method in Figure 2.3 returns the param-

eterized type, the `Cloneable` class is considered a positive recursion. When positive recursive usages are encoded in mainstream object-oriented languages that do not support F-bounded polymorphism, the output type is usually the most general type, and a dynamic cast is performed to get back the original type. With F-bounded polymorphism, the type of the output can be guaranteed statically.

The example in the original paper (Canning et al. [1989]) was able to express a type t as "movable" by bounding it with a special constructor `F-Movable[t]` that represents a type with a `move()` method. This can be expressed in Nominal Wyvern with subtyping.

```
// assume pre-defined Real type with "+" operator
name RealPair {
  val l : Real
  val r : Real
}
name Movable {m =>
  type t <= ⊤
  def move : RealPair amount -> m.t
}

name Point {p =>
  type t <= Point
  val x : Real
  val y : Real
  def move : RealPair amount -> p.t
}
subtype Point <: Movable
// constructor for points
name PointCons {pc =>
  def create : RealPair pos -> Point
}

// container for F-bounded Movable objects
name F-Movable {x =>
  type t <= Movable {type t <= x.t}
  val obj : x.t
}
name Utils {
  // arbitrarily translate any movable object
  def translate : F-Movable arg -> arg.t
}
```

```
32
33 let utils = new Utils {
34   // translate any movable object by 1.0 (:Real) in both directions
35   def translate : F-Movable arg -> arg.t =
36     arg.move(new Pair {val l = 1.0, val r = 1.0})
37 } in
38 let pointCons = new PointCons {pc =>
39   def create : RealPair pos -> Point =
40     new Point {p =>
41       type t = Point
42       val x = pos.l
43       val y = pos.r
44       def move : RealPair amount -> Point =
45         pc.create(new RealPair {
46           val l = p.x + amount.l
47           val r = p.y + amount.r
48         })
49     }
50 } in
51 let origin = pointCons.create(
52   new RealPair {val l = 0.0, val r = 0.0}) in
53 utils.translate(
54   new F-Movable {x => type t = Point, val obj = origin})
```

Listing 5.3: F-Movable example in Nominal Wyvern

Any structure that structurally satisfies the `Movable` interface and semantically supports such a `move` operation may subtype `Movable`. As a result, it will be allowed to be passed to the `utils.translate` method to get a translated version of itself.

```
1  ...
2  name Vector2D {v =>
3    type t <= Vector2D
4    val x : Real
5    val y : Real
6    def move : RealPair amount -> v.t
7  }
8  subtype Vector2D <: Movable
9  ...
10 // zero : Vector2D
11 utils.translate(
```

```
12    new F-Movable {x => type t = Vector2D, val obj = zero})
```

Listing 5.4: More movable types

5.3.2 Negative Recursion

In contrast to positive recursion, negative recursion is when the parameterized type is an input to a method, or contravariant. One popular use of this is the built-in `equals()` methods of `Object` in Java. Traditionally in Java, any object that wants to override the equals method needs to put a boilerplate at the beginning to make sure the object that is passed in is indeed of the same type as the parent type. This is because any overriding methods have to preserve the original signature. Thus, all `equals()` methods takes in a generic `Object`. With F-bounded polymorphism, this boilerplate can be checked by the type system. For Nominal Wyvern, this is illustrated as the motivating example of chapter 3 in section 3.2. The property of having an `equals()` method is expressed with the named type `Equatable`.

5.4 Family Polymorphism

Family polymorphism [Ernst, 2001] is useful when subtyping a set of types that are mutually dependent. A classic example is a `node` type and an `edge` type. The types are inter-dependent because a node references its incident edges, and an edge references its two endpoint nodes. To create a specific type of graph, one may wish to subtype both `node` and `edge` types. Without variances, the subtypes still refer to the general `node` and `edge` types, and dynamic checks have to be performed. This can be made statically safe with path-dependent types.

```
1  // assume pre-defined type Bool with constructors "true" and "false"
2  name Unit {}
3
4  name Node {n =>
5    type e <= Edge
6    // checks if edge is incident on this node
7    def touches : n.e edge -> Bool
8  }
9  name Edge {e =>
```

```
  type n <= Node
  // the two endpoints of this edge
  val l : e.n
  val r : e.n
}
// constructors for both types
name Graph {c =>
  type n <= Node
  type e <= Edge
  def createNode : Unit x -> c.n
  def createEdge : {val a : c.n, val b : c.n} arg -> c.e
}

let g = new Graph {c =>
  type n = Node
  type e = Edge
  def createNode : Unit x -> Node =
    new Node {n =>
      type e = Edge
      def touches : Edge edge -> Bool =
        if[Bool] edge.l = n then true
        else if[Bool] edge.r = n then true
             else false
    }
  def createEdge : {val a : Node, val b : Node} arg -> Edge =
    new Edge {e =>
      type n = Node
      val l = arg.a
      val r = arg.b
    }
} in
let node1 = g.createNode(new Unit {}) in
let node2 = g.createNode(new Unit {}) in
let edge12 = g.createEdge({val a = node1, val b = node2}) in
...
```

Listing 5.5: Family polymorphism: general nodes and edges

The example above sets up the general `Node` and `Edge` types for creating a general graph. Now we can subtype the two inter-dependent types to create the OnOffGraph from Ernst [2001]. (Identical structures are omitted below for conciseness).

```
// assume pre-defined type Bool with constructors "true" and "false"
name Unit {}

name Node {...}
name Edge {...}
// constructors for both types
name Graph {...}

name OnOffNode {n =>
  type e <= OnOffEdge
  def touches : n.e edge -> Bool
}
name OnOffEdge {e =>
  type n <= OnOffNode
  val enabled : Bool      // each edge can be on or off
  val l : e.n
  val r : e.n
}
subtype OnOffNode <: Node
subtype OnOffEdge <: Edge

name Utils {u =>
  def build : {arg =>
      val g : Graph
      val a : arg.g.n
      val b : arg.g.n
    } arg -> arg.g.e
}

let g = new Graph {...} in
let oog = new Graph {type n <= OnOffNode, type e <= OnOffEdge} {c =>
  type n = OnOffNode
  type e = OnOffEdge
  def createNode : Unit x -> OnOffNode =
    new Node {n =>
      type e = OnOffEdge
      def touches : OnOffEdge edge -> Bool =
        if[Bool] edge.l = n then edge.enabled
        else if[Bool] edge.r = n then edge.enabled
            else false
    }
```

```
  def createEdge : {val a : OnOffNode, val b : OnOffNode} arg
    -> OnOffEdge =
      new Edge {e =>
        type n = OnOffNode
        val enabled = true  // default to enabled
        val l = arg.a
        val r = arg.b
      }
} in
let utils = new Utils {u =>
  def build : {...} arg -> arg.g.e =
    arg.g.createEdge({val a = arg.a, val b = arg.b})
} in
let n1 = g.createNode(new Unit {}) in
let n2 = g.createNode(new Unit {}) in
let oon1 = oog.createNode(new Unit {}) in
let oon2 = oog.createNode(new Unit {}) in
// OK
let e12 = utils.build({val g = g, val a = n1, val b = n2}) in
let ooe12 = utils.build({val g = oog, val a = oon1, val b = oon2}) in
// type mismatch
let fail1 = utils.build({val g = oog, val a = n1, val b = n2}) in
let fail2 = utils.build({val g = oog, val a = oon1, val b = n2}) in
...
```

Listing 5.6: Family polymorphism with OnOffGraph

OnOffEdges can be turned on or off, so they are a special type of edge. Family polymorphism guarantees that `utils.build` will only work if the two types are of the same graph family.

5.5 Representing ML Modules

Data abstraction in ML is based on abstract data types (ADT). An ADT encapsulates an abstract type along with operations on the type. This serves as an interface that clients of the ADT can use without depending on (or even having any knowledge of) the implementation details, including what the abstract type actually represents.

Formally, ADTs are modeled with existential types: $\exists t.\tau$, where τ is typically a prod-

uct of functions that operate on the abstract type t. This can simply be represented in DOT-based systems by an object with a type member. For example, a `natlist` type in System FE (modified from Harper [2016]) is

$$\exists(t.\langle \text{emp} \hookrightarrow t, \text{ins} \hookrightarrow nat \times t \to t, \text{rem} \hookrightarrow t \to (nat \times t) + void\rangle$$

, where `emp`, `ins`, `rem` are the empty (i.e. create new), insert, and remove operations on the abstract list type.

Note that the implementation of the functions are coupled with the type. Any two expressions both of type `natlist` will use the same implementation (hidden, but fixed nonetheless). In DOT based systems, an interface to a type does not define its implementation. On this front, objects in DOT, and by extension Nominal Wyvern, are more similar to objects than ADTs. ADTs can still be represented, though, with a pre-defined interface and a "standard" implementation.

```
// assume pre-defined Nat type.
// assume the Option type has the following signature
name Option {o =>
  type elem <= ⊤         // type of the enclosed element
  def isSome : Unit x -> Bool
  def get : Unit x -> o.elem
}
name Product {p =>
  type Ta <= ⊤
  type Tb <= ⊤
  val a : Ta
  val b : Tb
}

name NatListInterface {nl =>
  type t <= ⊤
  val emp : nl.t
  def ins : {val elem : Nat, val list : nl.t} arg -> nl.t
  def rem : nl.t list ->
    Option {type elem = Product {type Ta = Nat, type Tb = nl.t}}
}
let natlist = new NatListInterface {nl => ...} in ...
```

Listing 5.7: Existential types in Nominal Wyvern

ML modules solve the single implementation problem of ADTs by wrapping them in named structures. Signatures define interfaces, and structures ascribe to signatures and define their own implementation. This is very closely modeled by Nominal Wyvern. Below is a classic `NatSet` example translated into Nominal Wyvern.

```
// assume pre-defined Bool type with constructors "true" and "false"
// NAT_SET interface
name NAT_SET {s =>
  type set
  val emptyset : s.set
  def insert : {val x : Nat, val S : s.set} arg -> s.set
  def member : {val x : Nat, val S : s.set} arg -> Bool
}

...
// assume a pre-defined 'natlist' object as defined earlier
let NatSet = new NAT_SET {s =>
  type set = natlist.t
  val emptyset = natlist.emp
  def insert : {val x : Nat, val S : s.set} arg -> s.set =
    natlist.ins({val elem = x, val list = S})
  def member : {val x : Nat, val S : s.set} arg -> Bool =
    let elem = natlist.rem(S) in
    if[Bool] elem.isSome(new Unit {}) = false then false
    else
      let data = elem.get(new Unit {}) in
      if data.a.equals(x) = true then true
      else s.member({val x = x, val S = data.b})
} in ...
```

Listing 5.8: `NatSet` in Nominal Wyvern

The example follows the SML naming convention. Signatures are named in ALL_CAPS and structures are named in CamelCase. Module `NatSet` ascribing to signature `NAT_SET` in SML is translated into object `NatSet` exhibiting type `NAT_SET`. The benefit of having objects represent modules is the ability to have first-class modules. For the previous example, a function could take in a generic list module (represented as an object with type List), and use it to produce a nat_set module (an object with type NAT_SET).

```
// a generic List interface
name LIST {l =>
```

```
  type elem <= ⊤  // element type
  type t <= ⊤
  val emp : l.t
  def ins : {val x : l.elem, val L : l.t} arg -> l.t
  def rem : l.t list ->
    Option {type elem = Product {type Ta = l.elem, type Tb = l.t}}
}
name Utils {
  def createNatSet : LIST {type elem <= Nat} l -> NAT_SET
}

let utils = new Utils {
  def createNatSet : LIST {type elem <= Nat} l -> NAT_SET =
    new NAT_SET {s =>
      type set = l.t
      val emptyset = l.emp
      def insert : {val x : Nat, val S : s.set} arg -> s.set =
        l.ins({val elem = x, val list = S})
      ...
    }
} in ...
```

Listing 5.9: Representing functors as functions

Note that signature modifications with `where` can be somewhat modeled with type refinements. Type refinements are more expressive in that it can specify bounds on type members of a module, but `where` is more flexible in that it can be used to directly relate members of two modules.

5.6 Object-Oriented Programming

One of the main differences between pure objects in object-oriented programming (OOP) languages and ADTs is how each paradigm relates interfaces to implementations. The interface of an object type is defined separately from its implementation, whereas the implementation of the functions in an ADT is part of its type. While modules allowed the separation of the interface and implementation via signatures and structures, it is not able to overcome the problem of having the implementation tied to the type it provides. Even if two modules both ascribe to the `List` signature, they cannot operate on each other's list

type. This is due to the internal need to unpack abstract types when operating on them, which only the type-providing module can do. Objects, however, do not provide any types. They instead provide implementations for a common type with a common interface. In fact, multiple objects of the same type can have wildly different implementations. Yet they can still interact with each other with no regard to the internal differences since, instead of unpacking the implementation type, they only rely on dynamically dispatched method calls over the common interface. This added interoperability contributes to the success of OOP languages [Aldrich, 2013].

The real world's version of OOP languages paints a different picture than just described. In the aforementioned "pure" OOP system, interoperability is enabled by autognosis [Cook, 2009], or not caring about the implementation of other objects: An object can only be interacted with over its public interface, which is considered its type. In contrast, popular OOP languages such as Java and C++ are heavily based on classes. Instead of knowing only about an object's type/interface, we can also know about its class, which may reveal information about the object's implementation if the class is concrete. This additional information breaches autognosis, making different parts of a system more interdependent than in pure OOP languages.

Nominal Wyvern's semantic separation naturally supports a pure OOP approach: Named types serve as interface definitions, and objects created from named types serve as constructors, or "classes". This way, the syntax guarantees interfaces are not tied to any implementation, and classes are syntactically different constructs than types. Classes are thus able to serve as pure organizers of implementations. The following listing translates the sets example from Cook [2009] into Nominal Wyvern. In Cook's paper, `ISet` defines the interface for sets, while the classes are simply constructor functions. Once created, an object is no longer associated with its constructing class, and can be freely used with objects created from other classes.

```
1 // assume pre-defined Int and Bool types.
2 // Int type has builtin constructors from literals, and an equals()
3 // method. Bool type has builtin constructors "true" and "false",
4 // and binary operator "||" for logical or.
5
6 // interface for sets
7 name ISet {s =>
8   def isEmpty() : Bool
9   def contains(i: Int) : Bool
```

```
  def insert(i: Int) : ISet
  def union(s: ISet) : ISet
}
// define classes/constructors
name SET_CONS {c =>
  def Empty() : ISet
  def Insert(s: ISet, n: ISet) : ISet
  def Union(s1: ISet, s2: ISet) : ISet
}

let Set = new SET_CONS {c =>
  def Empty() =
    new ISet {z =>
      def isEmpty() = true
      def contains(i: Int) = false
      def insert(i: Int) = c.Insert(z, i)
      def union(s: ISet) = s
    }
  def Insert(s: ISet, n: Int) =
    if[Bool] s.contains(n) = true then s else
    new ISet {z =>
      def isEmpty() = false
      def contains(i: Int) = (i.equals(n)) || (s.contains(i))
      def insert(i: Int) = c.Insert(z, i)
      def union(s: ISet) = c.Union(z, s)
    }
  def Union(s1: ISet, s2: ISet) =
    new ISet {z =>
      def isEmpty() = s1.isEmpty() || s2.isEmpty()
      def contains(i: Int) = (s1.contains(i)) || (s2.contains(i))
      def insert(i: Int) = c.Insert(z, i)
      def union(s: ISet) = c.Union(z, s)
    }
} in

let s1 = Set.Empty() in              // {}
let s2 = Set.Insert(s1, 1) in        // {1}
let s3 = s1.insert(2) in             // {2}
let s4 = Set.union(s2,s3) in         // {1,2}
...
```

Listing 5.10: Pure OOP in Nominal Wyvern

5.6.1 Mixing functional and OOP

The following listing shows an interesting combined usage of objects and functional modules. The pair type is used to store two objects of the same generic type. Like modules, the `PAIR_MOD` interface provides a `p` type that only the providing module can open. However, the provided type can act like an object in that it is self-contained and thus interoperable. It can be used like any `PAIR` without regard to who constructed it (although its constructor can be easily accessed via its `class` member.)

```
// interface for pairs
name PAIR {p =>
  type t <= ⊤              // type of the elements
  val class : PAIR_MOD      // reference to its class
  def l : Unit x -> arg.t
  def r : Unit x -> arg.t
}
// module providing pair types
name PAIR_MOD {c =>
  type p <= PAIR
  def create :
    {arg => type t
            val left : arg.t
            val right : arg.t} arg
      -> c.p {type t = arg.t}
  def l : c.p arg -> arg.t
  def r : c.p arg -> arg.t
}

// define a custom implementation
// ValPair uses two vals to store its info
name ValPair {p =>
  type t
  val class : PAIR_MOD
  def l : Unit x -> arg.t
  def r : Unit x -> arg.t
  val a : p.t
  val b : p.t
}
subtype ValPair <: PAIR

let ValPairMod = new PAIR_MOD {c =>
```

```
  type p = ValPair
  def create {arg => type t
                     val left : arg.t
                     val right : arg.t} arg
      -> c.p {type t = arg.t}
    new ValPair {type t = arg.t} {p =>
      type t = arg.t
      val class = c
      val a = arg.left
      val b = arg.right
      def l : Unit x -> arg.t = c.l(p)
      def r : Unit x -> arg.t = c.r(p)
    }
  def l : c.p arg -> arg.t = arg.a
  def r : c.p arg -> arg.t = arg.b
} in
let origin = ValPairMod.create(
  {type t = Int, val left = 0, val right = 0}) in
let zero1 = origin.l() in               // OO
let zero2 = ValPairMod.l(origin)        // functional
```

Listing 5.11: Mixing OOP and FP in Nominal Wyvern

The examples in this chapter serve to show that, in addition to the added code clarity brought by nominality, the restrictions made by Nominal Wyvern are not significant enough to impact its ability to express practical common patterns.

Chapter 6

Conclusion and Future Work

This thesis presents Nominal Wyvern, a new core type system for Wyvern based on the DOT calculus. Nominal Wyvern achieves a higher degree of nominality in a DOT-based system by semantically separating the definition of structures and their subtype relations from arbitrary type refinements and the declaration of type bounds. This contributes to a system with more explicit meanings and relations, useful for both human readers to reason about and programming tools to refer to. Nominality also helps with achieving subtype decidability. In line with the theme of semantic separation, Nominal Wyvern adapts material-shape separation so that decidability results from an intuitive separation of types with different roles. This contributes to a restriction that is more easily understandable and articulable. The resulting system preserves the ability to express common patterns expressible with DOT, at the same time allowing for patterns that will be familiar to programmers already used to traditional functional or object-oriented programming languages.

Some further areas of study are discussed below.

- *Type Safety*. This thesis focused on the nominality and decidability aspects without giving a soundness argument for the type system. Such a property is very important for a practical programming language. Therefore, the logical next step is to show that the system is sound with a type safety proof. One of the main difference from DOT that may make soundness not follow directly from DOT is the explicit subtype declarations that allow for multiple and conditional subtyping. Such a concept does not exist, and will need to be proven or otherwise tweaked to make sound.

- *More flexible width subtyping*. Nominal Wyvern separated out depth subtyping from width subtyping, making the formal automatic and the latter disallowed (have to define new names). One may argue that width subtyping has more to do with what the structure contains, which is more integral to the semantic meaning of the object than the absolute types of each member. If a member needs to be a more specific type, there is likely no need for an altogether semantically different structural name. In fact, the type system additionally constraints how one can refine a type by disallowing additional self-references in refinements (since that would likely change the semantics of the structure). However, there are likely exceptions to both: there may be cases where a width-refinement does not add much to the structure semantically, and there may be cases where a depth-refinement significantly changes the meaning of a structure. There are two potential solutions worth looking into:

  - Allow certain kinds of structural width subtyping. The downside is too much freedom risks taking away the benefits of the nominal system.
  - Encourage programmers to write a new structure when a certain depth refinement significantly alters the meaning.

  Both are subjective and warrant further study.

- *Bringing back uniformity*. One of the benefits of the DOT system is that the entire program is a first class value. In contrast, Nominal Wyvern breaks this uniformity by introducing a set of second-class declarations at the top level. It would be nice to bring back the uniformity of DOT while maintaining the benefits of nominality presented in this thesis. One potential solution is integrating the named type definitions and explicit subtype declarations as members of object. A separate notation would need to be created to refer to the named types of objects, as the binary typing approach (section 3.3.1) makes it necessary to keep type members and named types separate.

Bibliography

Jonathan Aldrich. The power of interoperability: Why objects are inevitable. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 101–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2514738. URL http://doi.acm.org/10.1145/2509578.2514738. 5.6

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. 1, 2.1, 2.2

Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 273–280, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: 10.1145/99370.99392. URL http://doi.acm.org/10.1145/99370.99392. 2.2, 5.3.1

Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. ISSN 0360-0300. doi: 10.1145/6041.6042. URL http://doi.acm.org/10.1145/6041.6042. 2.2

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. 1, 5.6

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. 5.4, 5.4

Giorgio Ghelli. Divergence of f≤ type checking. *Theor. Comput. Sci.*, 139(1-2):131–162, March 1995. ISSN 0304-3975. doi: 10.1016/0304-3975(94)00037-J. URL http://dx.doi.org/10.1016/0304-3975(94)00037-J. 2.2.1

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. (document), 1, 2.4, 2.2.1, 2.2.2, 3.4, 3.4, 3.4.1

Radu Grigore. Java generics are turing complete. *SIGPLAN Not.*, 52(1):73–85, January 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009871. URL http://doi.acm.org/10.1145/3093333.3009871. 1, 2.2

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, New York, NY, USA, 2nd edition, 2016. ISBN 1107150302, 9781107150300. 5.5

Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. 2006. 2.2.1

Julian Mackay. *Decidable Subtyping for Path Dependent Types*. PhD thesis, Victoria University of Wellington, 2019. In Submission. 1, 2.2.1, 2.2.3, 3.4, 3.4, 3.4.1

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. 1

Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP*, 2003. 1

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. 1, 2.2

Tom Van Vleck. Barbara liskov - a.m. turing award laureate, 2008. URL https://amturing.acm.org/award_winners/liskov_1108679.cfm. Accessed April 15, 2019. 1