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.

Copyright © 2019 Yu Xiang Zhu
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

Foo Bar
Contents

1 Introduction 1

2 Background and Motivation 5
  2.1 DOT and Path-Dependent Types 5
  2.2 Subtyping and Undecidability 7
    2.2.1 Getting Back Decidability 10
    2.2.2 Material-Shape Separation 12
    2.2.3 Material-Shape Separation for DOT 13
  2.3 Nominality 14
    2.3.1 Typing Nominality 14
    2.3.2 Subtyping Nominality 15

3 Nominal Wyvern Design 17
  3.1 A Store of Named Structures 17
  3.2 Adding Generics 19
  3.3 A Binary Typing Approach 21
    3.3.1 Nominal Subtyping Graph 24
  3.4 Material-Shape Separation 25
  3.5 Term Typing 29

4 Subtyping Decidability 31
  4.1 Subtyping Judgments 31
  4.2 Decidability 33

5 Expressiveness 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Syntax Sugar</td>
<td>35</td>
</tr>
<tr>
<td>5.2 Basic Path-Dependent Types</td>
<td>37</td>
</tr>
<tr>
<td>5.3 F-Bounded Polymorphism</td>
<td>39</td>
</tr>
<tr>
<td>5.3.1 Positive Recursion</td>
<td>39</td>
</tr>
<tr>
<td>5.3.2 Negative Recursion</td>
<td>42</td>
</tr>
<tr>
<td>5.4 Family Polymorphism</td>
<td>42</td>
</tr>
<tr>
<td>5.5 Representing ML Modules</td>
<td>45</td>
</tr>
<tr>
<td>5.6 Object-Oriented Programming</td>
<td>48</td>
</tr>
<tr>
<td>5.6.1 Mixing functional and OOP</td>
<td>50</td>
</tr>
<tr>
<td>6 Prototype Implementation</td>
<td>53</td>
</tr>
<tr>
<td>7 Conclusion and Future Work</td>
<td>55</td>
</tr>
<tr>
<td>Bibliography</td>
<td>57</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Type parameter vs type member ................................................. 8
2.2 Variance of type parameter (in Scala’s type parameter syntax) .......... 8
2.3 F-bounded polymorphism (in Scala) ........................................... 9
2.4 Infinite derivation example (from Greenman et al. [2014]) ............... 10
2.5 Diverging context example ..................................................... 11
2.6 Cloneable list causing expansive inheritance .............................. 12
2.7 Example translation from type parameter to type member ............. 13

3.1 Nominal Wyvern Grammar ..................................................... 22
3.2 Nominal Wyvern Term Typing .................................................. 29

4.1 Nominal Wyvern Subtyping ...................................................... 31
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 (aka. 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 (aka. accepts 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 to have 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 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 intracacies of their language’s type system may end up unable to compile their “well-written” program. The compiler’s failure also meant 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 (following an adaptation of Greenman et al. [2014]’s material-shape separation) to curb subtype undecidability. The goal is to produce a type system that avoids the usability pitfalls mentioned above in 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 gives an overview of a prototype typechecker implementation in Java using the Antlr library. Chapter 7 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., 2012] 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.

```
class Library { l =>
  type Book
  def borrow(title: String) : l.Book
  def renew(item: l.Book) : Unit = {}
}

val hunt : Library = ...
val sorrells : Library = ...
val csapp = sorrells.borrow("CSAPP")
sorrells.renew(csapp) // OK
hunt.renew(csapp) // type mismatch
  // found : sorrells.Book
  // required : hunt.Book
```

Listing 2.1: Path-dependent type
The `Library` class defines an abstract type member `Book`. This `Book` type is then used in the definition of the trait by the `borrow` and `renew` methods (the variable `l` is the self variable with which other members of the trait can be accessed). This means if you have a value, `sorrells`, of type `Library`, calling `borrow` on it would return a value of type `sorrells.Book`. 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 `sorrells.Book` is unknown (abstracted away in `Library`), renewing it from any other `Library` would not typecheck, even if the underlying `Book` type for the two Libraries are the same. The type system thus allows the code to model the real world restrictions of libraries (cannot renew a book borrowed elsewhere) without restricting the number of possible libraries 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). 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.

```plaintext
1 class GenericBook {}
2 class Textbook extends GenericBook {}
3 class Fiction extends GenericBook {}
4
5 class SchoolLibrary extends Library { l =>
6   type Book ≤ Textbook
7   ...
8 }
9
def study(item: Textbook): Unit = ...
10
11 val hunt : Library = ...
12 val sorrells : SchoolLibrary = ...
13 study(sorrells.borrow("CSAPP"))  // OK
14 study(hunt.borrow("Harry Potter"))  // type mismatch
```

Listing 2.2: Bounds on type members

1In this system, a completely opaque type is usually defined with an upper bound of the top type (`⊤`), which is defined as the supertype of all types. On the contrary, a bottom type (`⊥`) typically exists and is defined as the subtype of all types.
In Listing 2.2, `SchoolLibrary` is defined to be the kind of library whose books are all of an abstract type that is a subtype of `Textbook`. This means books borrowed from a `SchoolLibrary` can be passed to `study()`, while a book borrowed from a generic `Library` whose type member `Book` is completely opaque will not typecheck when studied.

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

```python
def borrowAndStudy(lib: Library{type Book ≤ Textbook}): Unit =
    study(lib.borrow("CSAPP"))
```

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\(^2\) 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 subsumption 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.

\(^2\)Or “subclassing”, since types and classes are often conflated in this context.
Subtype checking is thus checking if the two types are correctly related with respect to the 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\(^3\)) 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).

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

\(^3\)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)

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

Subtype checking on these constructed types already involve 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 be able 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 in-
tricacies 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, 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 con-

4Running 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

9
Given types:

\[
\text{Eq}(\langle -T \rangle), \quad \text{List}(\langle +T \rangle) <: \text{Eq}(\text{List}(\text{Eq}(\langle T \rangle))), \quad \text{Tree} <: \text{List}(\text{Tree})
\]

Query: \text{Tree} <: \text{Eq}(\text{Tree})

\[
\text{Tree} <: \text{Eq}(\text{Tree})
\]
\[
\text{List}(\text{Tree}) <: \text{Eq}(\text{Tree})
\]
\[
\text{Eq}(\text{List}(\text{Eq}(\text{Tree}))) <: \text{Eq}(\text{Tree})
\]
\[
\text{Tree} <: \text{List}(\text{Eq}(\text{Tree}))
\]
\[
\text{List}(\text{Tree}) <: \text{List}(\text{Eq}(\text{Tree}))
\]
\[
\text{Tree} <: \text{Eq}(\text{Tree})
\]

\[
\ldots
\]

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

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 \(F_C\) is not a simple case of cycle detection. Figure 2.5 presents a classic example first discovered by Ghelli [1995] encoded in DOT (based on the translation from TODO [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 restriction on “expansive inheritance” by Kennedy and Pierce [2006] as it is used by the Scala compiler. Among their three proposed solutions, the most reasonable restriction was disallowing dependency cycles with “expansive edges”: where one type parameter of type \(S\) may
Given types:

\[ N = \{ n \Rightarrow L \geq \bot \} \]

\[ T = \left\{ z \Rightarrow \begin{array}{c} A \leq \top \\ B \leq N \{ n \Rightarrow L \geq \left\{ z_1 \Rightarrow \begin{array}{c} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\} \} \end{array} \right\} \]

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

Query: \( T_0 <: T \) ?

\[ \emptyset \vdash T_0 <: T \]

\[ z_0 : T_0 \vdash z_0.A <: N \{ n \Rightarrow L \geq \left\{ z_1 \Rightarrow \begin{array}{c} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\} \} \]

\[ z_0 : T_0 \vdash N \{ n \Rightarrow L \geq T \} <: N \{ n \Rightarrow L \geq \left\{ z_1 \Rightarrow \begin{array}{c} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\} \} \]

\[ z_0 : T_0 \vdash \left\{ z_1 \Rightarrow \begin{array}{c} A \leq z_0.A \\ B \leq z_1.A \end{array} \right\} <: T \]

\[ z_0 : T_0 \vdash z_1.A <: N \{ n \Rightarrow L \geq \left\{ z_2 \Rightarrow \begin{array}{c} A \leq z_1.A \\ B \leq z_2.A \end{array} \right\} \} \]

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 2.5: Diverging context example
appear 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 to 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 2.6, which now includes an expansive edge from $E$ to $E$.

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

Figure 2.6: Cloneable list causing expansive inheritance

### 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 clear and intuitive separation means that instead of arbitrarily restricting what programmers can write (and thus forcing them to adopt an esoteric rule), it can serve as a useful tool in helping them structure their code to be more modular, and, according to the codebase review, in a way that most programmers already do.

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.

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

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

trait Pair { z =>
    type E ≤ T
    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
Nominal Wyvern’s adaptation of material-shape separation is inspired by work by [TODO [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. Chapter 3 details our restrictions, and Chapter 4 proves the termination of the subtyping algorithm under the restrictions.

2.3 Nominality

The typing of the DOT calculus is already considered 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.

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 predefined and named in the scope. This is due to two reasons:

1. Usability: Explicitly written out structures makes 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 hav- 
ing 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 pre- 
checked 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. Since Nominal Wyvern 
disallows anonymous structural types, structural subtyping is no longer required. Follow- 
ing a similar reasoning as typing nominality, Nominal Wyvern requires all subtyping rela- 
tions to be explicitly declared. Type S subtypes type T only if the programmer writes that 
they want S to subtype T. The two benefits are similar:

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 
epects, and prevents accidental passing of the wrong argument to a function even 
if naively the structures match up. The added benefit is a more understandable sub- 
typing 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 N.N to N.N builds up a motivating example; Sections N.N to N.N explains 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 stop the programmer from mixing apples and oranges since structural types are given names in addition to its 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.

\(^1\)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.
as seen on line 26 of Listing 3.1

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 StockCounter 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.
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 2.6, the type member \( T \)'s purpose is for future subtypes to refine with themselves. In order for \texttt{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 \texttt{Fruit}, its \texttt{equals} method must take in an object of its own \( T \) type.

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

\begin{verbatim}
name Equatable {z =>
  type T >= BOT
  fun equals (x : z.T) : Bool
}
name Fruit {z =>
  type T >= BOT
  val ID : Int
  val weight : Float
  fun equals (x : z.T) : Bool
}
subtype Fruit <: Equatable
name Apple {z =>
  type T >= BOT
  ...
}
name Orange {z => ...
}
subtype Apple {type T >= Apple} <: Fruit
subtype Orange {type T >= Orange} <: Fruit
name Set {z =>
  type elemT <= Equatable {type T >= z.elemT}
  ...
}
name SetTracker {z =>
  type S <= Set
  fun empty () : z.S
  fun insert (stock : z.S, item : stock.elemT) : z.S
  ...
}
\end{verbatim}
let apple_tracker = new SetTracker {z =>
  type S = Set
  fun empty () : z.S =
    new Set {s =>
      type elemT = Apple
      ...
    }
  fun 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
  fun 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 3.3: Adding generics to Nominal Wyvern

### 3.3 A Binary Typing Approach

The main difference of Nominal Wyvern compared to DOT is its heterogenous typing system. In DOT, all 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 spec-
Figure 3.1: Nominal Wyvern Grammar
ified. 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 3.3, 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 types are named types behind abstraction boundaries.

The separation of named types from member types also requires a separate way of defining the subtype relation between named types. Traditionally in DOT, the subtype relation between types (which are all type members) is defined solely by the bounds declared on the type members. Syntactically, this no longer works as named types are separately defined. Semantically, bounds are also no longer suitable since the subtype relation between named types should be a semantic definition instead of a guide/guarantee. In Nominal Wyvern, bounds are complemented by the explicit subtype declaration statements. The purpose of this subtype declaration is for programmers to declare semantically (and the type system to verify structurally) suitable base cases for the subsumption principle: If we think of McIntosh as a special kind of Apple, then a McIntosh should be able to be used anywhere a generic Apple is needed. The declarations are only base cases because depth subtyping in the form of type refinement is automatic (e.g. in Listing 3.3, `Apple {type T = Apple} <: Apple`).

In addition to the benefit of semantic separation, 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. In the absence
of arbitrary intersection types, this circumvents the problem 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 compatibility or semantic compatibility). The syntax allows a refinement on the LHS of the subtype symbol ( <: ) to make a subtype relation hold conditionally. Section P.Q.R presents an example where conditional subtyping is used.

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 subsumption principle. Section N details the binary subtyping algorithm and why it is a decidable problem after material-shape separation is applied.

### 3.3.1 Nominal Subtyping Graph

To aid in nominal subtyping, we define a “nominal subtyping graph” to capture the multiple conditional subtyping relation 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:*

\[
\begin{align*}
n_1 r_1 &\  <: \  n_2 \\
\overline{n}_1 &\  \rightarrow \ n_2
\end{align*}
\]

**Example 1.** *The nominal subtyping graph for the code in Listing 3.3 is:*
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. 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 et al.’s adaptation in Decidable Wyvern. 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 \( \langle V, r, E \rangle \). The vertices, \( V \), consist of all the base types referred to in \( \tau \). The root, \( r \), is \( \beta \). The edges, \( E \), each go from a base type, \( \beta' \), to the root of the refinement tree generated by the type used as the bound in a refinement to \( \beta' \). It is formally generated by \text{GenTree}:
GenTree(τ) = ⟨V, r, E⟩. (Each edge is represented by v₁ → v₂: An edge from node n₁ to n₂).

\[ r = \text{type } t₁ \ B₁ \ τ₁ \ \ldots \ \text{type } tₙ \ Bₙ \ τₙ \]

\[
\text{GenTree}(τ₁) = ⟨V₁, r₁, E₁⟩ \ \ldots \ \text{GenTree}(τₙ) = ⟨Vₙ, rₙ, Eₙ⟩
\]

\[
\text{GenTree}(βr) = ⟨V₁ \ldots Vₙ, β, β, E₁ \ldots Eₙ (β \rightarrow r₁) \ldots (β \rightarrow rₙ)⟩
\]

Example 2. The refinement tree for type

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

is:

The subtype dependency graph is defined partially using refinement trees.

Definition 3 (Subtype dependency graph). For a set of top level declarations \(\overline{D}\), the subtype dependency graph is a graph \(⟨V, E⟩\). The vertices, V, consist of all available base types, which are: \(⊤, \bot\), 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 \(n::t\). The edges, E, are generated as follows:
For each type member declaration, \( \text{type } t \beta r t \) in named type \( n \), generate edges:

\[
\begin{align*}
\text{DIRECT} & : n :: t \rightarrow \beta_t \\
\text{INDIRECT} & : n :: t \overset{\text{rta}(\beta_r t, \beta_r)}{\rightarrow} \beta_r \\
\text{IMPLICIT} & : n :: t \overset{\text{rta}(\beta_r t, n)}{\rightarrow} n_r
\end{align*}
\]

And for each subtype declaration \( \text{subtype } n_1 r_1 <: n_2 \) in \( \overline{D} \), generate edges:

\[
\begin{align*}
\text{BACK} & : n_2 \rightarrow n_1 \\
\text{BACK-REF-ROOT} & : n_2 \rightarrow n_r \\
\text{BACK-REF} & : n_2 \overset{\text{rta}(n, n', r)}{\rightarrow} n'
\end{align*}
\]

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

The idea is if \( \beta \) has an edge to ("depends on") \( \beta' \), then when \( \beta \) is encountered as a base type during subtype derivation, \( \beta' \) 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 3.3 is:
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
- A named shape cannot be refined in subtype declarations

**Example 4.** For the code in Listing 3.3, 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.
## 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.

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

\[
\Delta \Sigma \Gamma S \vdash \ e : \tau
\]

\[
\Delta \Sigma \Gamma S \vdash x : \tau \quad \Gamma(x) = \tau \quad \text{T-VAR}
\]

\[
\Delta \Sigma \Gamma S \vdash l : \tau \quad S(l) = \tau \quad \text{T-LOC}
\]

\[
\Delta \Sigma \Gamma S \vdash p : \tau \quad \Delta \Sigma \Gamma \vdash \exists_x \text{val } v : \tau_v \quad \text{T-SEL}
\]

\[
\Delta \Sigma \Gamma S \vdash p : \tau \quad \Delta \Sigma \Gamma \vdash \tau \forall_x \text{fun } f : \tau_a x_a \rightarrow \tau_r \quad \text{T-APP}
\]

\[
\Delta \Sigma \Gamma S \vdash p \, . \, f(p') : [p, p_a/x_a] \tau_r \quad \Delta \Sigma \Gamma S \vdash \tau \{ x \rightarrow \bar{d} \} \text{ wf} \quad \text{T-NEW}
\]

\[
\Delta \Sigma \Gamma S \vdash p : \tau \quad \Delta \Sigma \Gamma S \vdash \tau \forall_x \text{let } x = e_x \text{ in } : \tau \quad \text{T-LET}
\]

\[
\Delta \Sigma \Gamma S \vdash \text{if}[\tau] p_1 = p_2 \text{ then } e_1 \text{ else } e_2 : \tau \quad \text{T-IF}
\]

Figure 3.2: Nominal Wyvern Term Typing

*It is a valid use of shape since it follows the restrictions in Definition 5.*
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. Section N will show why this separate view of the same type is type safe.

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 parent object of the method. 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 type of the return type with the argument variable and self variable replaced with paths from the current context.
Chapter 4

Subtyping Decidability

4.1 Subtyping Judgments

TODO

Figure 4.1: Nominal Wyvern Subtyping

Figure 4.1 presents 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 type.

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 are 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.
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’. Figure N illustrates an example type member subtype derivation sequence.

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
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. Figure N presents an example of structural subtyping.

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
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 syntax 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 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 ⇒ σ}`). 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 type members of the parent type are swapped for either `⊤` or `⊥` when moved outside. The use site subsequently refines the fresh name type with the referenced 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 `<method_ID>_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 ⇒ 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 Library
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.

```plaintext
// assume a pre-defined String type. String objects are created
// directly with string literals.
name Unit {}

name GenericBook {}
name Textbook {}
subtype Textbook <: GenericBook
name Fiction {}
subtype Fiction <: GenericBook

name Library {l =>
type Book
  fun borrow : String title -> l.Book
  fun renew : l.Book item -> Unit
}

name SchoolLibrary {l =>
type Book <= Textbook
  fun borrow : String title -> l.Book
  fun renew : l.Book item -> Unit
}
subtype SchoolLibrary <: Library

name Utils {
def study : Textbook item -> Unit
}

let hunt = new Library {l =>
```

37
The types of books are empty structures whose names give them meaning. Two types of libraries are defined. Note that the `type` 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`.

The guarantees provided by DOT are preserved in Nominal Wyvern:

- A book can only be renewed from the library it was borrowed from: Each library object \(x\) carries with it a type \(x\.Book\) that is hidden from the outside. In this case, the object \(csapp\) has type \(sorrells\.Book\) (derived directly from the signature of the `borrow()` method of `SchoolLibrary`, the type of `sorrells`). Therefore, it is incompatible with the input type of `hunt\.renew()`, which requires the argument to be of the entirely opaque type `hunt\.Book`.

\[\text{Listing 5.1: Basic path-dependent types in Nominal Wyvern}\]

```plaintext
let sorrells = new SchoolLibrary {l =>
  type Book = Textbook
  fun borrow : String title -> l.Book = new Textbook {}
  fun renew : l.Book item -> Unit = new Unit {}
} in
// attempt to renew
let csapp = sorrells.borrow("CSAPP") in
let legal = sorrells.renew(csapp) in // OK
let illegal = hunt.renew(csapp) in // type mismatch

let utils = new Utils {
  def study : Textbook item -> Unit = new Unit {}
} in
// attempt to study
let pass = utils.study(sorrells.borrow("CSAPP")) in // OK
let fail = utils.study(hunt.borrow("Harry Potter")) in
  // type mismatch
...```

\[1\] This difference is discussed more in depth in section 5.6
• Only a textbook can be “studied”: Even though the exact type of `sorrells.Book` is unknown, `SchoolLibrary` provides the hint that its book type is a subtype of `TextBook`. As a result, it fits the requirement of `util.study()`.
In contrast, since `hunt.Book` is completely abstract, it is not safe to pass into `util.study()`.

Since Nominal Wyvern supports refinements, one can avoid the need to declare a specific library 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 `sorrells` library can be written without a `SchoolLibrary`:

```plaintext
... let sorrells = new Library {type Book <= Textbook} {l =>
  type Book = Textbook
  fun borrow : String title -> l.Book = new Textbook {}
  fun renew : l.Book item -> Unit = new Unit {}
} in
...
```

Listing 5.2: Expressing `SchoolLibrary` with a refinement

The underlined type declared in the `new` expression denotes the type that this new object has. The `Book` type is refined so that books borrowed here can be studied. 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 parameterized 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 \( F\text{-}{\text{Movable}}[t] \) that represents a type with a \texttt{move()} method. This can be expressed in Nominal Wyvern with subtyping.

```plaintext
// assume pre-defined Real type with "+" operator
name RealPair {
  val l : Real
  val r : Real
}
name Movable {m =>
  type t <= T
  fun move : RealPair amount -> m.t
}
name Point {p =>
  type t <= Point
  val x : Real
  val y : Real
  fun move : RealPair amount -> p.t
}
subtype Point <: Movable
// constructor for points
name PointCons {pc =>
  fun create : RealPair pos -> Point
}

name Utils {
  // arbitrarily translate any movable object
  fun translate : {x =>
    type t <= Movable {type t <= x.t}
    val obj : x.t
  } arg -> arg.t
}

let utils = new Utils {
  // translate any movable object by 1.0 (:Real) in both directions
  fun translate : {x =>
```
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.

```ocaml
... name Vector2D {v =>
  type t <= Vector2D
  val x : Real
  val y : Real
  fun move : RealPair amount -> v.t
}

subtype Vector2D <: Movable
...

// zero : Vector2D

utils.translate({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 overridden 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.

```plaintext
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   fun touches : n.e edge -> Bool
8 }
9
10 name Edge {e =>
11   type n <= Node
12   // the two endpoints of this edge
13   val l : e.n
14   val r : e.n
15 }
```
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).

Listing 5.5: Family polymorphism: general nodes and edges
// constructors for both types

name Graph {...}

name OnOffNode {n =>
    type e <= OnOffEdge
    fun 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 =>
    fun 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 a{type n <= OnOffNode, type e <= OnOffEdge} {c =>
    type n = OnOffNode
    type e = OnOffEdge
    fun createNode : Unit x -> OnOffNode =
        new Node {n =>
            type e = OnOffEdge
            fun touches : OnOffEdge edge -> Bool =
                if[Bool] edge.l = n then edge.enabled
                else if[Bool] edge.r = n then edge.enabled
                else false
        }
    fun createEdge : {val a : OnOffNode, val b : OnOffNode} arg
        -> OnOffEdge =
        new Edge {e =>
            type n = OnOffNode
            val enabled = true // default to enabled

```plaintext
val l = arg.a
val r = arg.b
}

let utils = new Utils {u =>
  fun 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 $\tau$ is typically a product 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

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

```ml
// assume pre-defined Nat type.

// assume the Option type has the following signature
name Option {o =>
    type elem <= T  // type of the enclosed element
    fun isSome : Unit x -> Bool
    fun get : Unit x -> o.elem
}

name Product {p =>
    type Ta <= T
    type Tb <= T
    val a : Ta
    val b : Tb
}

name NatListInterface {nl =>
    type t <= T
    val emp : nl.t
    fun ins : {val elem : Nat, val list : nl.t} arg -> nl.t
    fun 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.

```ml
// assume pre-defined Bool type with constructors "true" and "false"

// NAT_SET interface
```
name NAT_SET {s =>
    type set
    val emptyset : s.set
    fun insert : {val x : Nat, val S : s.set} arg -> s.set
    fun 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
    fun insert : {val x : Nat, val S : s.set} arg -> s.set =
        natlist.ins({val elem = x, val list = S})
    fun 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. A direct result is we get generative functors for free. 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 <= T // element type
    type t <= T
    val emp : l.t
    fun ins : {val x : l.elem, val L : l.t} arg -> l.t
    fun rem : l.t list ->
        Option {type elem = Product {type Ta = l.elem, type Tb = l.t}}
Listing 5.9: Representing functors as functions

Note that signature modifications with where are modeled directly with type refinements.

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 reveals information about its implementation. This additional information breaches autognosis, making different parts of a system more inter-dependent 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.

```plaintext
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 // interface for sets
6 name ISet {s =>
7   fun isEmpty() : Bool
8   fun contains(i: Int) : Bool
9   fun insert(i: Int) : ISet
10  fun union(s: ISet) : ISet
11 }
12
13 // define classes/constructors
14 name SET_CONS {c =>
15   fun Empty() : ISet
16   fun Insert(s: ISet, n: ISet) : ISet
17   fun Union(s1: ISet, s2: ISet) : ISet
18 }
```
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)
    }
  }
}

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 \texttt{PAIR} without regard to who constructed it (although its constructor can be easily accessed via its \texttt{class} member.)

```plaintext
// interface for pairs
name PAIR {p =>
  type t <= T  // type of the elements
  val class : PAIR_MOD  // reference to its class
  fun l : Unit x -> arg.t
  fun r : Unit x -> arg.t
}

// module providing pair types
name PAIR_MOD {c =>
  type p <= PAIR
  fun create :
    {arg => type t
      val left : arg.t
      val right : arg.t} arg
    -> c.impl {type t = arg.t}
    fun l : c.impl arg -> arg.t
    fun r : c.impl arg -> arg.t
}

// define a custom implementation
// ValPair uses two vals to store its info
name ValPair {p =>
  type t
  val class : PAIR_MOD
  fun l : Unit x -> arg.t
  fun 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}
```
```ocaml
new ValPair {type t = arg.t} {p =>
  type t = arg.t
  val class = c
  val a = arg.left
  val b = arg.right
  fun l : Unit x -> arg.t = c.l(p)
  fun 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
}

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
Chapter 6

Prototype Implementation

This chapter provides an overview to a prototype implementation of the Nominal Wyvern typechecker, and discusses some of the optimizations we are able to achieve in practice.
Chapter 7

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.

- 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 disal-
allowing 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) makes it necessary to keep type members and named types separate.
Bibliography


Nada Amin, Adriaan Moors, and Martin Odersky. Dependent object types - towards a foundation for scala’s type system. In *19th International Workshop on Foundations of Object-Oriented Languages*, 2012. 2.1


William R. Cook. On understanding data abstraction, revisited. In *Proceedings of


