Wyvern: A Simple, Typed, and Pure Object-Oriented Language

Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin[1], and Jonathan Aldrich

Carnegie Mellon University

{lnistor, darya, balzers, bwchung, aldrich}@cs.cmu.edu, alex@ecs.vuw.ac.nz[1]

Abstract

The simplest and purest practical object-oriented language designs today are seen in dynamically-typed languages, such as Smalltalk and Self. Static types, however, have potential benefits for productivity, security, and reasoning about programs. In this paper, we describe the design of Wyvern, a statically typed, pure object-oriented language that attempts to retain much of the simplicity and expressiveness of these iconic designs.

Our goals lead us to combine pure object-oriented and functional abstractions in a simple, typed setting. We present a foundational object-based language that we believe to be as close as one can get to simple typed lambda calculus while keeping object-orientation. We show how this foundational language can be translated to the typed lambda calculus via standard encodings. We then define a simple extension to this language that introduces classes and show that classes are no more than sugar for the foundational object-based language. Our future intention is to demonstrate that modules and other object-oriented features can be added to our language as not more than such syntactical extensions while keeping the object-oriented core as pure as possible.

The design of Wyvern closely follows both historical and modern ideas about the essence of object-orientation, suggesting a new way to think about a minimal, practical, typed core language for objects.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages

Keywords Object-oriented, first-class classes, static type checking

1. Introduction

The language designs of Smalltalk [11] and Self [20] are iconic in their elegance, their simplicity, and the success with which they capture the essence of object-oriented programming. Although they are not among the most common languages used today, their design has been an inspiration for many modern languages. Self and Smalltalk are notable for the following characteristics, many of which are shared by more recent object-oriented languages, such as Python, Ruby, Lua, and JavaScript:

- A **high-level, pure object-oriented** model, in which clients can interact with objects only by sending them messages. This model of interaction enforces Meyer's *uniform access principle* [16], a principle stating that an object's services should only be available through a uniform notation (the object's methods in this case).
- An interactive, **value-based** language model, which means that everything significant in the language is a first-class value, either an object or a function. This contributes to a system that is "directly manipulable" in the sense of Smith and Ungar [20]. For example, classes in Smalltalk are not special declarations, but live objects that respond to messages.
- A high degree of **simplicity**, with a few constructs sufficing to provide great expressiveness.
- Good support for **functional programming** via blocks or lambda expressions and closures.

Interestingly, both Smalltalk and Self, as well as many languages that share some of the characteristics above, are dynamically typed. Statically-typed object-oriented languages exist, but many are less pure (e.g., Java classes can expose fields), or less interactive (e.g., classes are typically not first-class), or may be considerably more complex (C++ and Scala, despite their many virtues, come to mind).

Static types are a potentially interesting addition to the set of features above for a number of reasons. In object-oriented languages, types can eliminate message-not-understood errors [12, 24] and be used to optimize programs [10] and to create practical refactoring tools [22]. Preliminary evidence suggests that, in some circumstances, static types can enhance productivity as well [15].

Our interest in this question comes from Wyvern, a language we are designing for secure mobile and web programming. To compare with programming languages in this domain, Wyvern must be a simple, value-based, pure object-oriented language with good support for functional programming—much like the languages mentioned above. However, we also want to provide strong reasoning about security properties. Reasoning is facilitated by a simple, high-level language, but, in addition, we want to use type-based approaches to ensure certain security properties (e.g., using inference rules developed for type checking to also check object ownership and state). Thus, combining static types with a language having the characteristics above is important to meeting this set of goals.

This paper therefore explores the design of an object-oriented language that is pure, value-based, simple, statically type-safe, and supports functional programming. We explore variants with and

MASPEGHI'13, July 01 2013, Montpellier, France
Copyright © 2013 ACM 978-1-4503-2046-7/13/07…$15.00

without classes, our motivation being to show that, while classes are convenient, they are not fundamental to the object-oriented model.

1.1 What Makes an Object-Oriented Model Pure?

Most of the criteria above are fairly clear. The exception involves the purity of the object model. What does it mean, especially in the context of types, for a language to be purely object-oriented?

To answer the question, we consider both history and theory. With respect to history, Alan Kay, a pioneer of the object-oriented programming who developed Smalltalk, describes an "objects as server" metaphor in which each "object would be a server offering services" that clients can use by sending messages to the object [13]. In fact, Kay writes that, in his opinion, "the big idea [of object-oriented programming] is messaging."[1]

With respect to theory, in his essay on the nature of objects, William Cook defines objects as "a value exporting a [*sic*] procedural interface to data or behavior" [9]. The notion of a procedural interface reflects Kay's focus on messaging: the only way to interact with an object is to invoke its methods. This is also consistent with Meyer's uniform access principle.

Cook argues that a key benefit of objects, arising from their procedural interface, is that different implementations of an interface can interoperate. For example, in a pure object-oriented model, we can take the union of a set implemented as a list with a set implemented as a hash table. Moreover, different implementations of an interface can be treated uniformly: for example, we can store instances of two different implementations of a set within the same data structure. These benefits are used in many object-oriented systems [5], and we consider them to be critical goals of our language design.

These ideas can also be considered in a comparison between objects and module systems, particularly, with respect to encapsulating state. Concerning the development of object-oriented programming, Kay writes of his motivations, "The large scale one was to find a better module scheme for complex systems involving hiding of details, and the small scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether" [13]. Hiding assignment behind a method interface—via the uniform access principle—provides a more flexible version of assignment, and one can go further in using that method interface more abstractly to "replace bindings and assignment with goals" [13]. The procedural interface of objects also makes them act like first-class modules.[2]

From these sources, we extract three key requirements that we wish to satisfy in coming up with a typed, pure object-oriented model:

- **Uniform access principle.** Following Meyer, Cook, and Kay, it should be possible to access objects only by invoking their methods.
- **Interoperability and uniform treatment.** Different implementations of the same object-oriented interface should interoperate by default, and it should be easy to treat them uniformly at run time (e.g., by storing different implementations of the same interface within a single run-time data structure).
- **State encapsulation.** All mutable state should be encapsulated within objects.[3]

[1] Alan Kay, email sent October 10, 1998 to `squeak@cs.uiuc.edu`.

[2] A module system for Wyvern is still under development, but a key principle will be that modules can be treated as any other object, following Newspeak [6].

[3] This should not be taken to imply that all objects have mutable state, which is a common misconception. See Cook's essay for details [9].

1.2 Mechanisms for Code Reuse

No discussion of a pure object-oriented programming model would be complete without discussion of code reuse mechanisms, such as inheritance. We have not yet investigated adding inheritance or delegation to Wyvern. However, Wyvern does support specialization and generalization of types, along with a pure object-oriented model that supports reuse via composition mechanisms. Questions of interest to us as we continue the development of Wyvern are:

- How far can a pure object-oriented model without inheritance go in supporting code reuse?
- What reuse mechanisms would fit most naturally into a system with Wyvern's design goals?

We are considering a reuse model for Wyvern based on delegation, possibly, with slightly simpler semantics compared to delegation in Self [20].

1.3 Contributions

This paper makes the following contributions:

- Section 2 presents the design of a simple, typed, value-based, pure object-oriented language. We closely follow the lambda calculus, a simple, foundational model of computation, while enforcing the uniform access principle, supporting object interoperability, and encapsulating state.
- Section 2 also provides the semantics of this language by translation into the typed lambda calculus.
- Section 3 describes a "purely object-oriented" first-class class construct and its definition in terms of pure objects by translation into the language described in the previous section.

The design of Wyvern is ongoing. While we believe the design described here captures our initial goals well, the design is subject to change and needs to be extended in many ways. In the conclusion, we discuss areas for future work, including specific potential extensions of Wyvern.

2. The Object-Oriented Core of Wyvern

In this section, we present the syntax and semantics of the object-oriented core of Wyvern, which we call Featherweight Wyvern (FW). We start with an example.

Figure 1 shows the definition of a `Lot` object in FW. The `Lot` type, which represents the object, is defined as a collection of methods. It's important to note that, in Wyvern, types are structural, and their names serve as abbreviations. Object types in Wyvern are implicitly recursive, allowing the `compare` method to take a `Lot` object as an argument to compare the value of this `Lot` to the value of the passed in `Lot`.

We create lots using the `purchase` method, which acts as a factory. The body of this method is a `new` statement which creates an object. The object defines two mutable variables of type `int`. These variables can be accessed only from methods within the object, enforcing encapsulation of mutable state. Definitions are provided for each method. Method invocations or field access on the receiver object can be expressed using the `this` keyword. In the `sell` method, each line is a different statement, and the value of the expression on the last line of the method is returned from the method. Code below the `purchase` method shows how `Lot` objects can be used.

```
type Lot =
   meth value : int
   meth compare(other : Lot) : int
   meth sell : int

meth purchase(q : int, p : int) : Lot =
   new
      var qty : int = q
      var price : int = p
      meth value : int =
         this.qty * this.price
      meth compare(other : Lot) : int =
         this.value - other.value
      meth sell : int =
         val qtySold = this.qty
         this.qty = 0
         qtySold

val aLot = purchase(100,100)
val d = aLot.compare(purchase(50,90))
val value = aLot.value
val qtySold = aLot.sell
```

Figure 1. A `Lot` Object in the Object-Oriented Core of Wyvern (Featherweight Wyvern)

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \boldsymbol{\lambda} x{:}\tau.e \\
 & | & e(e) \\
 & | & \texttt{new}\ \{\overline{d}\} \\
 & | & e.f \\
 & | & e.f = e \\
 & | & e.m \\
 & & \\
\tau & ::= & t \\
 & | & \tau \rightarrow \tau
\end{array}
\qquad
\begin{array}{rcl}
d & ::= & \texttt{var}\ f : \tau = e \\
 & | & \texttt{meth}\ m : \tau = e \\
 & | & \texttt{type}\ t\ = \{\overline{\tau_d}\} \\
 & & \\
\tau_d & ::= & \texttt{meth}\ m : \tau \\
 & & \\
\sigma & ::= & \tau \\
 & | & \{\overline{\sigma_d}\} \\
 & & \\
\sigma_d & ::= & \texttt{var}\ f : \tau \\
 & | & \texttt{type}\ t\ = \{\tau\} \\
 & | & \tau_d
\end{array}
$$

Figure 2. Featherweight Wyvern Syntax

From the `Lot` type alone we can already see how the design of Wyvern fulfills our criteria for a pure typed object-oriented model. As object types in Wyvern only contain methods, the uniform access principle is enforced. Although only one implementation of `Lot` is shown, the `Lot` type allows multiple implementations that can interoperate and may be treated uniformly. Finally, all state is encapsulated within the `Lot` object.

2.1 Syntax

Figure 2 shows the syntax for Featherweight Wyvern. The language is expression-based and has the lambda calculus at its core, thus providing functional programming support. In our text examples, we write $\boldsymbol{\lambda} x{:}\tau.e$ as `fn x:`τ `=> e`. To this functional core, we add a `new` construct for creating an object from a sequence of declarations and primitives for reading fields, writing fields, and invoking methods. Types τ may be either the name of an object type t or a function type $\tau \rightarrow \tau$.

The declarations that can appear inside a `new` expression include variable declarations, method declarations, and type abbreviations. Variable declarations are visible only within the object and are assigned a type and an initialization expression, which is executed when the object is created.

$$\frac{\Gamma, \sigma \vdash \overline{d} :: \overline{\sigma_d} \quad \sigma = \{\overline{\sigma_d}\} \quad \overline{\tau_d} \subseteq \overline{\sigma_d}}{\Gamma \vdash \texttt{new}\ \{\overline{d}\} : \{\overline{\tau_d}\}}\ \textit{T-new}$$

$$\frac{\Gamma, \textit{this} : \sigma \vdash e : \tau}{\Gamma, \sigma \vdash \texttt{meth}\ m : \tau = e :: \texttt{meth}\ m : \tau}\ \textit{DT-meth}$$

Figure 3. Featherweight Wyvern Semantic Rules for `new` and `meth`

Method declarations are also typed and have a body expression, but unlike with variables, the body is evaluated each time the method is called, not when the surrouding object is created. The careful reader will note that we do not directly support method arguments. Each method has an implicit argument `this`. If more arguments are needed, the body of the method can be a lambda expression that binds the additional arguments. For example, in our formal core language the declaration of `compare` would be written:

```
meth compare : Lot -> int =
   fn other : Lot =>
      this.value - other.value
```

Type abbreviations are strictly local and cannot be seen from outside the object; we may add type members in a future extension. An object type is defined as a set of method declaration types $\overline{\tau_d}$, which are just like method declarations except that the body is absent. Type abbreviations are recursive, so that t may appear in $\overline{\tau_d}$.

The typing and subtyping rules are mostly standard and are omitted here for space reasons; a companion technical report gives the complete rules [21]. The only exception is the way that `new` and `this` are typed (Figure 3) . In order to type `new`, we type each declaration in it against a *declaration type* σ_d. There is a declaration type form for each kind of declaration; thus, σ_d generalizes τ_d, which is restricted to method declarations only. A sequence of declaration types $\{\overline{\sigma_d}\}$ forms an internal object type σ.

We want to use this σ as the type for `this`, so that we can access not just methods, but also variables on it. Therefore, the σ computed in the `new` rule is passed as an input to the declaration typing judgment, to the left of the turnstile. In the method typing rule, the premise assumes that `this` has this internal object type σ.

Note that this mechanism not only *allows* methods to access fields on the receiver, it is also the mechanism that prevents fields from being accessed elsewhere. σ is not part of the grammar of ordinary types (in fact, it is a superset of types, not a subset), and, thus, a lambda may not have an argument of σ type. Hence, our type system prevents passing a pointer to `this` outside the lexical scope of a function at an internal object type. `this` may still be passed to other functions, but only as an external object type, in which all but the method declarations have been forgotten.

With the addition of integers and a few other small abbreviations, this syntax is expresive enough to write the `Lot` example from Figure 1. Our core language does not have top-level declarations, but we can easily wrap the `Lot` and `purchase` declarations in an object, put the rest of the code in an `eval` method of that object, and invoke `eval` on the top-level object. The example also uses a `val` abbreviation, which is Wyvern's version of `let`, can be encoded with functions, and is used just for visual brevity. For example, `val x =` e_1`;` e_2 can be written as `(fn x :` τ `=>` e_2`)(`e_1`)`, where τ is the type of e_1 in the current scope. Figure 4 shows the code with no syntactic sugar, except for `val`.

```
val globalObject = new
   type Lot =
      meth value : int
      meth compare : Lot -> int
      meth sell : int

   meth purchase : int -> int -> Lot =
      fn q : int => fn p : int =>
      new
         var qty : int = q
         var price : int = p
         meth value : int =
            this.qty * this.price
         meth compare : Lot -> int =
            this.value - other.value
         meth sell : int =
            val qtySold = this.qty
            this.qty = 0
            qtySold

   meth eval : unit =
      val aLot = this.purchase(100,100)
      val d = aLot.compare(this.purchase(50,90))
      val value = aLot.value
      val qtySold = aLot.sell

globalObject.eval
```

Figure 4. The `Lot` Object in Featherweight Wyvern Without Syntactic Sugar Except for `val`

2.2 Discussion and Related Calculi

We believe that the design above fulfills the criteria we set out in the introduction. Featherweight Wyvern is a pure object-oriented language in that it enforces the uniform access principle, interoperability of different implementations of the same object interface, and state encapsulation. The language is statically typed, and all elements of the language, except types, are first-class values. The lambda calculus core provides good support for functional programming.

Furthermore, since in pure object-oriented languages, an object's behavior is characterized by the messages it responds to and a structural type describes this directly, for the simple core of Wyvern, we use structural typing. Empirical studies also have shown structural typing to be useful in practice [14]. We intend to support nominal types as an addition when they are needed, for example, for abstract data types.

We argue that the language is about as simple as it could possibly be while fulfilling these goals: beyond the lambda calculus, which is necessary for functional programming, we have exactly one construct each for object creation, field reads and writes, and method calls. The type language is also extremely simple, supporting recursive object types and function types, as well as the special internal σ type for `this` mentioned above.

We briefly consider whether the language could be simpler while still achieving our goals. One obvious approach to simplifying the language would involve encoding one of the lambda calculus or methods in terms of the other. In fact, in the next subsection, we will give semantics to Featherweight Wyvern by translating objects into the lambda calculus with records. Alas, these encodings are quite awkward for a source-level language, and we wish for Featherweight Wyvern to provide source-level support for both objects and functions.

A second possible simplification would be to unify methods and fields, as done in Abadi and Cardelli's calculi [4]. However, expressing methods using fields at the source level would not enforce

$$
\begin{array}{rcl}
\mathit{trans}(\mathtt{meth}\ m : \tau) & \equiv & m : \mathtt{unit} \to \tau \\
\mathit{trans}(\mathtt{var}\ f : \tau) & \equiv & f : \mathtt{ref}\ \tau \\
\mathit{trans}(\mathtt{meth}\ m : \tau = e; \overline{d}) & \equiv & m : \tau = \boldsymbol{\lambda}\_ : \mathtt{unit}.\mathit{trans}(e); \\
 & & \mathit{trans}(\overline{d}) \\
\mathit{trans}(\mathtt{var}\ f : \tau = e; \overline{d}) & \equiv & f : \tau := \mathtt{alloc}\ \mathit{trans}(e); \\
 & & \mathit{trans}(\overline{d}) \\
\mathit{trans}(e.m) & \equiv & (\mathtt{unfold}\ \mathit{trans}(e)).m() \\
\mathit{trans}(\mathtt{new}\ \{\overline{d}\}) & \equiv & \mathtt{letrec}\ \mathit{this} : \sigma = \\
 & & \quad \{\mathit{trans}(\overline{d})\} \\
 & & \mathtt{in}\ \mathtt{fold}\ \mathit{this} \\
 & & \mathit{where}\ \ \Gamma, \sigma \vdash \{\overline{d}\} : \sigma \\
\mathit{trans}(e.f = e_1) & \equiv & \mathit{trans}(e).f = \mathit{trans}(e_1) \\
\mathit{trans}(e.f) & \equiv & !\mathit{trans}(e).f \\
\mathit{trans}(\mathtt{type}\ t = \{\overline{\tau_d}\}; \overline{d}) & \equiv & \mathit{trans}([\boldsymbol{\mu} t.\mathit{trans}(\overline{\tau_d})/t]\overline{d})
\end{array}
$$

Figure 6. Translation from Featherweight Wyvern to the Extended Lambda Calculus

the uniform access principle (since the fields could be accessed directly) and would leave Wyvern without a purely object-oriented core. Expressing fields as methods raises questions about how to effectively hide state and creates typing complexity, requiring that a distinction is made between updatable methods (i.e., fields) and non-updatable ones [4]. Therefore, this simplification is not appropriate for Featherweight Wyvern.

Featherweight Wyvern does have strong similarities to object calculi proposed in the research literature. Our calculus is very similar in its modeling of objects, functions, and methods to Abadi and Cardelli's $\mathbf{FOb_1}$ calculus. The main difference is that, as discussed above, Abadi and Cardelli use a method update operation instead of modeling state with fields. This keeps their calculus small but also prevents it from being a practical language.

2.3 Semantics

We describe the semantics of FW by translation into the typed lambda calculus with extensions for records, the fix operator, references, iso-recursive types, and subtyping. The syntax of our target language is shown in Figure 5; its semantics are covered in our technical report [21] but are in many cases directly taken from Benjamin Pierce's textbook [18].

The *trans* function in Figure 6 is used to translate expressions and declarations from FW to typed lambda calculus. When translating `meth` $m : \tau$, the keyword `meth` is removed and the method m becomes a field of type τ. A variable `var` $f : \tau$ becomes a reference f. The translation of `meth` $m : \tau = e$ assigns to the field m the translation of expression e, wrapping the expression in a lambda so that it is evaluated when the method is called rather than on object creation. The assignment to a variable `var` $f : \tau = e$ is translated by allocating a cell to hold the result of executing the translated expression e.

The translation of $e.m$ is more interesting since m becomes a field (or a record member) of e and we need to unfold e in order to be able to access m. After unfolding the translated expression, we select the method m and call the resulting function with a unit value to evaluate the method body.

In the translation of `new` $\{\overline{d}\}$, we use the first of the object encodings discussed by Bruce et al. [8]. We create a new object as a record, using the translation of the declarations $\overline{d}$. The created object must be available within its own methods as the `this` variable, and so we recursively bind the record value to `this` using a `letrec` construct. Finally, we fold the result to a recursive type.

Translation of field reads and writes is straightforward; the only interesting issue is that we must add an explicit dereference to field

$$
\begin{array}{lcl}
\tau & ::= & \tau \rightarrow \tau \\
 & | & \{f_i{:}\tau_i^{i\in 1..n}\} \\
 & | & \texttt{ref}\ \tau \\
 & | & t \\
 & | & \boldsymbol{\mu} t.\tau \\
 & & \\
\Gamma & ::= & \{\overline{x{:}\tau}\} \\
\Sigma & ::= & \{\bar{l}{:}\overline{\tau}\}
\end{array}
\qquad
\begin{array}{lcl}
v & ::= & \boldsymbol{\lambda} x{:}\tau.e \\
 & | & \{f_i = v_i^{i\in 1..n}\} \\
 & | & \ell \\
 & | & \texttt{fold}[\tau]\ v \\
 & & \\
S & ::= & \{\bar{l} = \overline{v}\}
\end{array}
\qquad
\begin{array}{lcl}
e & ::= & x \\
 & | & \boldsymbol{\lambda} x{:}\tau.e \\
 & | & e(e) \\
 & | & \{f_i = e_i^{i\in 1..n}\} \\
 & | & e.f \\
 & | & \texttt{fix}\ e \\
 & | & \texttt{alloc}\ e \\
 & | & !e \\
 & | & e := e \\
 & | & \texttt{fold}[\tau]\ e \\
 & | & \texttt{unfold}[\tau]\ e \\
 & | & l
\end{array}
$$

$$\texttt{letrec}\ x{:}\tau_1 = e_1\ \texttt{in}\ e_2 \overset{def}{=} \texttt{let} x{:}\tau_1 = \texttt{fix}(\boldsymbol{\lambda} x{:}\tau_1.e_1)\ \texttt{in}\ e_2$$

$$\texttt{let}\ x{:}\tau_1 = e_1\ \texttt{in}\ e_2 \overset{def}{=} (\boldsymbol{\lambda} x{:}\tau_1.e_2)(e_1)$$

Figure 5. Lambda Calculus with Extensions [18]

```
type Lot = rec t . {
   value : unit -> int
   compare : unit -> t -> int
   sell : unit -> int
}
type LotT = rec t . {
   qty : ref int
   price : ref int
   value : unit -> int
   compare : unit -> Lot -> int
   sell : unit -> int
}
fun purchase(q : int, p : int) : Lot =
   letrec this : LotT = {
      qty : ref int = alloc q
      price : ref int = alloc p
      value : unit -> int = fn _ : unit =>
         !this.qty * !this.price
      compare // not shown
      sell : unit -> int = fn _ : unit =>
         val qtySold = !this.qty
         this.qty = 0
         qtySold
   } in fold this

val aLot = purchase(100, 100)
val d = unfold(aLot).compare()(purchase(50,90)
val value = unfold(aLot).value()
val qtySold = unfold(aLot).sell()
```

Figure 7. The Lot Object Translated to the Lambda Calculus with Extensions

reads because of the way references are handled in our extended lambda calculus. Finally, we translate type bindings by substituting the equivalent recursive type for t in the declarations that follow.

When we apply our translation rules shown in Figure 6, the code from Figure 1 is translated to the extended lambda calculus code shown in Figure 7. In this simpler version of the language, `Lot` is defined as a recursive type. The methods from Figure 1 are translated into corresponding fields `value`, `compare`, and `sell`. We have given this type an abbreviation `Lot` for readability; type abbreviations are technically not in our target language, but they are a standard addition.

We also need to define an internal type `LotT` that captures the implementation used in the `purchase` function; this type is like `Lot` but additionally includes the mutable state. Note that the type of `compare` in `LotT` uses type `Lot` rather than `t` for its argument type. This allows `LotT` to be a subtype of `Lot`, and also ensures that our interoperability design criterion is met: the `LotT` implementation can interoperate with any other implementation of `Lot`.

The remainder of the translation follows the translation rules in a straightforward way.

3. Extending Wyvern with Classes

In this section, we present an extension to Featherweight Wyvern (FW) called Featherweight Wyvern with Classes (FWC) that adds first-class classes. In fact, this language is a subset of the current, working implementation of Wyvern freely available online and briefly described in the next section.

The extended syntax is shown in Figure 8. The basic forms of expressions and types are identical to those from the object-oriented core of Wyvern, presented in the previous section. To the declaration syntax, we add a class declaration. Class declarations can have declarations within them, which reflect the variables and methods of the objects created from that class.

Wyvern also allows classes to declare class variables and class methods, which are members of the class itself. Class methods and variables are inspired by the corresponding constructs in Smalltalk. They are also similar to static methods and static fields in Java but fit more cleanly into a language where classes are first-class objects.

As with the previous core language, we use σ to denote the internal type of a class or object, and we extend the internal declaration types σ_d to add the appropriate class-related declarations.

Figure 9 shows a variation on our earlier example, written in FWC and taking advantage of classes. The `Option` class represents a financial option. We introduce a class field `totalQuantityIssued` to keep track of how many options were issued during the class's lifetime and use a class method `issue` to create new options, i.e., new objects of this class. As with the earlier `Lot` example, there are two object fields, `quantity` and `price`, that store the corresponding information about `Option` instances, and an object method `exercise` that is called upon the end of the lifetime of an object of the `Option` class. Lines 17-18 illustrate how the `issue` and

$$
\begin{array}{lcl}
e & ::= & x \\
  & \mid & \boldsymbol{\lambda} x{:}\tau.e \\
  & \mid & e(e) \\
  & \mid & \texttt{new}\,\{\overline{d}\} \\
  & \mid & e.f \\
  & \mid & e.f = e \\
  & \mid & e.m \\
\\
\tau & ::= & t \\
  & \mid & \tau \to \tau \\
\\
\tau_d & ::= & \texttt{meth}\, m : \tau
\end{array}
\qquad
\begin{array}{lcl}
d & ::= & \texttt{var}\, f : \tau = e \\
  & \mid & \texttt{meth}\, m : \tau = e \\
  & \mid & \texttt{type}\, t\, \{\overline{\tau_d}\} \\
  & \mid & \texttt{class}\, c\, \{\, \overline{cd};\ \overline{d}\, \} \\
\\
cd & ::= & \texttt{class var}\, f : \tau = e \\
  & \mid & \texttt{class meth}\, m : \tau = e
\end{array}
\qquad
\begin{array}{lcl}
\sigma & ::= & \tau \\
  & \mid & \{\overline{\sigma_{cd}}\} \\
\\
\sigma_{cd} & ::= & \texttt{class var}\, f : \tau \\
  & \mid & \texttt{class meth}\, m : \tau \\
  & \mid & \sigma_d \\
\\
\sigma_d & ::= & \texttt{var}\, f : \tau \\
  & \mid & \texttt{type}\, t\, \{\overline{\tau_d}\} \\
  & \mid & \texttt{class}\, c\, \{\, \overline{\sigma_{cd}},\ \overline{\sigma_d}\, \} \\
  & \mid & \tau_d
\end{array}
$$

Figure 8. Syntax of Featherweight Wyvern with Classes

```
class Option
  var quantity : int = 0
  var price : int = 0
  meth exercise : int =
    this.quantity * this.price

  class var totalQuantityIssued : int = 0
  class meth issue : int -> int -> Option =
    fn q : int =>
      fn p : int =>
        totalQuantityIssued =
          totalQuantityIssued + q
        new
          var quantity : int = q
          var price : int = p

var optn : Option = Option.issue(100, 50)
var ret : int = optn.exercise
```

Figure 9. An `Option` Class in Featherweight Wyvern with Classes

$$
\begin{array}{l}
\mathit{trans}(\texttt{class}\ c\ \{\overline{cd}; \overline{d}\}) \equiv \texttt{type}\ c = \tau_i;\ \texttt{var}\ c : \tau_c = e \\
\quad \text{where} \\
\qquad \tau_c = \{\overline{\texttt{meth}\ m : \tau}\} \\
\qquad\quad \text{where } \texttt{meth}\ m : \tau \in \tau_c \\
\qquad\qquad \text{iff } \texttt{class meth}\ m : \tau = e \in \overline{cd} \\
\qquad \tau_i = \{\overline{\texttt{meth}\ m : \tau}\} \\
\qquad\quad \text{where } \texttt{meth}\ m : \tau \in \tau_i \text{ iff } \texttt{meth}\ m : \tau = e \in \overline{d} \\
\qquad \overline{d_{cl}} = \{\overline{\texttt{meth}\ m : \tau = e}\} \cup \{\overline{\texttt{var}\ f : \tau = e}\} \\
\qquad\quad \text{where } \texttt{meth}\ m : \tau = e \in \overline{d_{cl}} \\
\qquad\qquad \text{iff } \texttt{class meth}\ m : \tau = e \in \overline{cd} \\
\qquad\quad \text{and } \texttt{var}\ f : \tau = e \in \overline{d_{cl}} \\
\qquad\qquad \text{iff } \texttt{class var}\ f : \tau = e \in \overline{cd} \\
\qquad \overline{d'_{cl}} = [\texttt{new}\ \{\overline{d} \oplus \overline{d'}\}\ /\ \texttt{new}\ \{\overline{d'}\}]\ \overline{d_{cl}} \\
\qquad \overline{d''_{cl}} = \mathit{trans}(\overline{d'_{cl}}) \\
\qquad e = \texttt{new}\{\overline{d''_{cl}}\}
\end{array}
$$

Figure 10. Translation of a Class from FWC to FW

`exercise` methods can be used. Here, for briefness, we used a shorthand to represent the statement sequence, which can be trivially encoded using a `let` expression. If the code snippet from Figure 9 is wrapped in a `new` statement, it becomes a runnable FWC program.

We argue that we do not need to make classes explicit to have an object-oriented language, and, therefore, Figure 10 presents a translation function. It simplifies our representation by replacing every occurrence of a class declaration with a corresponding type and a variable that can be used to instantiate objects of the class. The translation relies on the fact that typing is structural and that multiple names for the same type do not affect the semantics.

In Featherweight Wyvern, we represent a class as a type abbreviation and a set of methods originally included in the class. We start the translation by collecting all the class methods and converting them into regular methods. The resulting type τ_c is an externally visible type for the class object. Next, we assemble a type that will represent the objects instantiated from the class, τ_i, using the object methods declared in the class.

After establishing the types we need for our translation, we proceed by translating the class's declaration statements. First, we collect all the class methods and class fields and rewrite them without the keyword `class` ($\overline{d_{cl}}$). Then, each `new` statement inside the collection $\overline{d_{cl}}$ is substituted with a `new` statement that carries a context and includes an overriding union of declarations $\{\overline{d} \oplus \overline{d}'\}$, where $\overline{d}'$ are the declaration that were in the `new` statement previously and $\overline{d}$ are the declarations defined in the class. Finally, we recursively translate the resulting set of declarations $\overline{d'_{cl}}$ and wrap it with a `new` statement. Hence, the class is translated to be a type abbreviation and a variable containing declarations from the class.

Our translation thus reveals that a class is treated both as a type and as a value in Wyvern. This is natural given that classes are first-class in Wyvern and reflects existing usage of classes in prior languages: for example, classes are types in Java and most other typed object-oriented languages, and classes are values in Smalltalk. This duality does not cause semantic problems because values and types are in different namespaces in Wyvern, and the context always makes it clear whether a value or a type is required.

Figure 11 shows the result of applying the translation function to the `Option` class we saw earlier (Figure 9). The resulting code starts with the definitions of type abbreviations that directly correspond to the types created by the translation function. Specifically, `type Option` represents the type τ_i for objects of the class, and `type OptionC` represents the type τ_c for the class itself. Then comes the translation of the class itself, which, in FW, is a variable definition. Its type is the externally visible class type `OptionC`, and it is defined to be a `new` statement containing the class declarations from the original class. In particular, it contains a variable `totalQuantityIssued`, which is a class field in FWC, and a method `issue`, which is a class method in FWC and whose return type is `Option`, i.e., the type representing objects of the class. Inside the `issue` method we find the regular declarations that pertain

```
type Option =
  meth exercise : int

type OptionC =
  meth issue : int -> int -> Option

var Option : OptionC =
  new
    var totalQuantityIssued : int = 0
    meth issue : int -> int -> Option =
      fn q : int =>
        fn p : int =>
          totalQuantityIssued =
            totalQuantityIssued + q
          new
            var quantity : int = q
            var price : int = p
            meth exercise : int =
              this.quantity * this.price

var optn : Option = Option.issue(100, 50)
var ret : int = optn.exercise
```

Figure 11. `Option` Class Translated to Featherweight Wyvern

to the FWC class: variables `quantity` and `price` and the `exercise` method.

Our technical report [21] accompanying this paper provides the subtyping rules, both static and dynamic semantics, states the soundness theorems, and allows the reader to see how the resulting language can be proved safe by the appropriate translation to the simple typed lambda calculus with extensions. The next section briefly surveys our prototype implementation.

4. Implementation

Wyvern[4] is implemented in Java, with the eventual goal of self-hosting, i.e., being written in Wyvern itself, and supports a superset of features presented here. A selection of tests with simple Wyvern programs is available as part of the source code.

Wyvern uses a fixed whitespace-indented lexing approach similar to that used by languages, like Python. The current indentation convention is based on the number of whitespace characters.

Wyvern was developed to support extensible parsing interface. This means that, as such, there are no keywords defined in the language: instead each keyword is mapped to a corresponding parser that can parse the relevant block of code. For example, core implementation includes a "class parser" that supports class declarations that may include other declarations, such as "methods" and "variables", which are, in turn, parsed by the associated parsers. As a result, Wyvern front-end effectively combines the parsing and type checking stages that are usually separated in more traditional compilers, such as `javac` or `gcc`.

Currently, Wyvern supports three targets: the Java interpreter and compilation to either JavaScript or Java, all of which enable us to test and run Wyvern programs. We are working on introducing full interoperability with both JavaScript and Java to enable simple Wyvern programs to join a web-oriented workflow and, hence, allow us to perform usability and security-oriented studies of our language.

[4] `https://github.com/wyvernlang/wyvern`

5. Related Work

A central inspiration in the design of Wyvern and its core language is Smalltalk [11]. Smalltalk was seminal in the development of object-oriented languages and is still in use today, e.g., in its Squeak [3] and GemStone/S [2] incarnations. While simple and elegant, it is dynamically typed, as opposed to Wyvern, which is statically typed. In both Smalltalk and FWC, classes are first-class objects. Smalltalk is more strongly class-based however, in that every object is an instance of a class; this is not true of FWC, and especially not of FW.

Strongtalk [7] is a Smalltalk environment with optional static typing support. The type system is incremental and operates independently of the compiler. Wyvern's type system has some similarities to Strongtalk's, e.g., both type systems are structural. Strongtalk's type system is more complex and powerful than Wyvern's, including features, such as parametric polymorphism and brands. We may add some of these features to Wyvern, but unlike Strongtalk, we deliberately do not brand class types to make them unique. This means that, when two classes implement the same interface, the types they define are identical, and, therefore, the classes can interoperate. We disagree with the choice made by Strongtalk but, also, by more recent languages, such as Java, to make branding the default for classes because this compromises the interoperability provided by the object-oriented paradigm. If we add brands to Wyvern, programmers will have to request their use explicitly.

Gradualtalk [1] is another Smalltalk dialect offering gradual typing and is fully compatible with Smalltalk code. Although more recent and more expressive than Strongtalk, its relationship to Wyvern is similar.

Self [20] is a prototype-based object-oriented language, providing an object model that is even simpler than Smalltalk's, although the surface syntax and environments of the two languages are similar. Wyvern shares Self's philosophy that classes are not required but provides them as syntactic sugar for convenience. Indeed, experience with Self suggests that many programmers create libraries of prototype objects that act similarly to classes.

Scala [17] is an object-functional programming and scripting language for general software applications and is statically typed. Scala includes full support for functional and object-oriented programming. Many of its features are akin to those of Wyvern. For example, it is pure in the sense that every value is an object; a function or method $foo()$ can also be called as just foo, thus enforcing the uniform access principle. While Scala is quite successful, it has been criticized for its complexity. Wyvern tries to provide a practical language design with some of the same goals as Scala but with a strong emphasis on simplicity at the expense of adding features.

Emerald [19] is an "object-based" programming language designed for building distributed applications. Like Wyvern, it is purely object-oriented, supports the uniform access principle, and does not rely on the concept of a class. However, there are significant differences. The key difference is the ultimate goals of the languages: Emerald targets distributed application overall whereas Wyvern focuses specifically on the web and mobile applications and aims at ensuring applications' security properties. In addition, Emerald is a mature language and, thus, is more feature-rich while Wyvern is still at an early stage of development and, therefore, might seem as a subset of Emerald. Nevertheless, considering the difference in the goals, as Wyvern design progresses, the divergences of the features is expected to be more pronounced.

JavaScript is a scripting programming language with great opportunities of code re-use via mixins, but it is dynamically typed and lacks direct support for classes and inheritance. However, its counterpart TypeScript [16], which represents a superset of JavaScript, is enhanced with static typing and class-based object-oriented paradigms. Classes in TypeScript are implemented in accordance with the upcoming ECMAScript 6 [23] standard. Neither JavaScript nor TypeScript supports the uniform access principle. Instead, objects are modeled as records, and fields can be accessed directly from outside an object without indirection via methods. Although TypeScript supports pure object-oriented interfaces, types generated from classes are branded, and, when these types are used, they interfere with interoperability between different implementations of the same interface.

6. Conclusion

In this paper, we presented Wyvern, a simple, pure, value-based, statically typed object-oriented language that supports functional programming and is designed for building secure web and mobile applications. We show how its foundational object-oriented core allows the introduction of more complex program structures, such as classes, and reuse mechanisms in a clean and organized manner. There are a number of potential extensions of Wyvern that remain open questions, including the following:

- A reuse mechanism, such as inheritance or delegation,
- A first-class, typed module system,
- Support for tags, `instanceof`, and pattern matching, and
- Support for abstract type members.

7. Acknowledgements

We are grateful to Du Li, Cyrus Omar, and the workshop reviewers for their valuable feedback on this paper. This research was supported in part by the Swiss National Science Foundation through grant PBEZP2_140051, by the U.S. National Science Foundation under grant #CCF-1116907, "Foundations of Permission-Based Object-Oriented Languages", and by the U.S. Department of Defense.

References

[1] A Practical Gradual Type System For Smalltalk. `http://pleiad.cl/research/software/gradualtalk`.

[2] GemStone/S. `http://gemtalksystems.com`.

[3] Squeak Smalltalk. `http://www.squeak.org`.

[4] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[5] Jonathan Aldrich. The Power of Interoperability: Why Objects Are Inevitable, 2013. Submitted for publication. Available at `http://www.cs.cmu.edu/~aldrich/papers/objects-essay.pdf`.

[6] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as objects in newspeak. In *ECOOP*, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Gilad Bracha. The strongtalk type system for smalltalk. In *OOPSLA Workshop on Extending the Smalltalk Language*, 1996.

[8] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing Object Encodings. *Information and Computation*, 155(1–2):108–133, 1999.

[9] William R. Cook. On understanding data abstraction, revisited. *SIGPLAN Not.*, 44(10):557–572, October 2009.

[10] A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 23(1):30–72, January 2001.

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[12] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

[13] Alan C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, March 1993.

[14] Donna Malayeri and Jonathan Aldrich. Is Structural Subtyping Useful? An Empirical Study. In *ESOP*, pages 95–111, Berlin, Heidelberg, 2009. Springer-Verlag.

[15] C. Mayer, S. Hanenberg, R. Robbes, É. Tanter, and A. Stefik. An empirical study of the influence of static type systems on the usability of undocumented software. In *OOPSLA*, pages 683–702, New York, NY, USA, 2012. ACM.

[16] Microsoft. TypeScript. `http://www.typescriptlang.org`.

[17] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2010.

[18] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[19] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: a general-purpose programming language. *Softw. Pract. Exper.*, 21(1):91–118, December 1990.

[20] Randall B. Smith and David Ungar. Programming as an experience: the inspiration for Self. In *ECOOP*, volume 952, pages 303–330. Springer-Verlag, 1995.

[21] The Plaid Group. The Core Wyvern Language. Technical report, Carnegie Mellon University, April 2013. Available at `http://www.cs.cmu.edu/~aldrich/papers/temp/core-language.pdf`.

[22] F. Tip, R. M. Fuhrer, A. Kieżun, M. D. Ernst, I. Balaban, and B. D. Sutter. Refactoring using type constraints. *ACM Trans. Program. Lang. Syst.*, 33(3):9:1–9:47, May 2011.

[23] Allen Wirfs-Brock. ES6 Max-min class semantics. Mozilla, Presented at TC-39 meeting, San Francisco, CA, July 2012.

[24] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.