Wyvern Formalisation: Objects, Classes, Modules, Type Members

Alex Potanin

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

The Internet [of Things]

- JavaScript
- Ruby on Rails
- Java
- Flash
- PHP
- Python
- Coffee Script
- ...

- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Injection Attacks
- Insecure Direct Object References
- Broken Authentication and Session Management
- ...

(OWASP Top 10 2013)

Wyvern

A web and mobile programming language that is **secure by default**.

http://www.cs.cmu.edu/~aldrich/wyvern/

Our Goal: To simultaneously enhance **security** and **productivity** for **mobile** and **web** applications by co-designing a **language**, its **types**, and its **libraries**.

Wyvern

- Guide: http://www.cs.cmu.edu/~aldrich/wyvern/wyvern-guide.html

```
require stdout

stdout.print("Hello, World!")
```

- Wyvern Language Features
  - Statically type checked
  - Structural types
  - Indentation-based
  - First class classes and modules

Evolution of Wyvern Since its Birth in 2013

What's there in 2014:

- A pure object-oriented model that supports reuse via composition mechanisms (see MASPEGHI 2013)
- Specialization and generalization of types (see Onward! Essay 2013 by Jonathan)
- Support for Type-Specific Languages (see ECOOP 2014)

What's there in 2015:

- High-level abstractions for architecture and data (see IWACO 2014)
- Support for combining structural and nominal typing using tags (see ECOOP 2015)
- A reuse mechanism, such as inheritance or delegation (see FTfJP 2015)
- A first-class, typed module system (*in progress*)
- Support for abstract type members (*in progress*)

What's Pure OO?

- State encapsulation (OO)
- Uniform access principle (Meyer)
- Interoperability and uniform treatment (Cook)

Wyvern Core 0: Extended Lambda Calculus

$$
\begin{array}{rcl}
\tau & ::= & \tau \to \tau \\
& | & \{f_i{:}\tau_i^{i \in 1..n}\} \\
& | & \mathtt{ref}\ \tau \\
& | & t \\
& | & \boldsymbol{\mu} t.\tau
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \boldsymbol{\lambda} x{:}\tau.e \\
& | & \{f_i = v_i^{i \in 1..n}\} \\
& | & l \\
& | & \mathtt{fold}[\tau]\ v
\end{array}
\qquad
\begin{array}{rcl}
e & ::= & x \\
& | & \boldsymbol{\lambda} x{:}\tau.e \\
& | & e(e) \\
& | & \{f_i = e_i^{i \in 1..n}\} \\
& | & e.f \\
& | & \mathtt{fix}\ e \\
& | & \mathtt{alloc}\ e \\
& | & !e \\
& | & e := e \\
& | & \mathtt{fold}[\tau]\ e \\
& | & \mathtt{unfold}[\tau]\ e \\
& | & l
\end{array}
$$

$$
\begin{array}{rcl}
\Gamma & ::= & \{\overline{x}{:}\overline{\tau}\} \\
\Sigma & ::= & \{\bar{l}{:}\overline{\tau}\}
\end{array}
\qquad
\begin{array}{rcl}
S & ::= & \{\bar{l} = \overline{v}\}
\end{array}
$$

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

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

Wyvern Core 1: Adding Objects

$$
\begin{array}{lcl}
e & ::= & x \\
  & | & \boldsymbol{\lambda} x{:}\tau.e \\
  & | & e(e) \\
  & | & \texttt{new}\ \{\overline{d}\} \\
  & | & e.f \\
  & | & e.f = e \\
  & | & e.m
\end{array}
$$

$$
\begin{array}{lcl}
\tau & ::= & t \\
  & | & \tau \to \tau
\end{array}
$$

$$
\begin{array}{lcl}
d & ::= & \texttt{var}\ f : \tau = e \\
  & | & \texttt{def}\ m : \tau = e \\
  & | & \texttt{type}\ t\ = \{\overline{\tau_d}, attributes = e\}
\end{array}
$$

$$
\begin{array}{lcl}
\tau_d & ::= & \texttt{def}\ m : \tau
\end{array}
$$

$$
\begin{array}{lcl}
\sigma & ::= & \tau \\
  & | & \{\overline{\sigma_d}\}
\end{array}
$$

$$
\begin{array}{lcl}
\sigma_d & ::= & \texttt{var}\ f : \tau \\
  & | & \texttt{type}\ t\ = \{\tau\} \\
  & | & \tau_d
\end{array}
$$

Wyvern Core 1: Sample Program

```
type Lot =
   def value : Int

def purchase(q : Int, p : Int) : Lot =
   new
      var quantity : Int = q
      var price : Int = p
      def value : Int = this.quantity * this.price

var aLot  : Lot = purchase(100, 100)
var value : Int = aLot.value
```

Classes are Not Essential

e.g. Self and JavaScript

…but they are convenient.

We believe classes should be syntactic sugar on top of a foundational object-oriented core.

Wyvern Core 2: Adding Classes

$$
\begin{array}{lcl}
e & ::= & x \\
 & | & \boldsymbol{\lambda} x{:}\tau.e \\
 & | & e(e) \\
 & | & \texttt{new}\ \{\overline{d}\} \\
 & | & e.f \\
 & | & e.f = e \\
 & | & e.m
\end{array}
$$

$$
\begin{array}{lcl}
\tau & ::= & t \\
 & | & \tau \rightarrow \tau
\end{array}
$$

$$
\begin{array}{lcl}
\tau_d & ::= & \texttt{def}\ m : \tau
\end{array}
$$

$$
\begin{array}{lcl}
d & ::= & \texttt{var}\ f : \tau = e \\
 & | & \texttt{def}\ m : \tau = e \\
 & | & \texttt{type}\ t\ \{\overline{\tau_d}\} \\
 & | & \texttt{class}\ c\ \{\ \overline{cd};\ \overline{d}\ \}
\end{array}
$$

$$
\begin{array}{lcl}
cd & ::= & \texttt{class var}\ f : \tau = e \\
 & | & \texttt{class def}\ m : \tau = e
\end{array}
$$

$$
\begin{array}{lcl}
\sigma & ::= & \tau \\
 & | & \{\overline{\sigma_{cd}}\}
\end{array}
$$

$$
\begin{array}{lcl}
\sigma_{cd} & ::= & \texttt{class var}\ f : \tau \\
 & | & \texttt{class def}\ m : \tau \\
 & | & \sigma_d
\end{array}
$$

$$
\begin{array}{lcl}
\sigma_d & ::= & \texttt{var}\ f : \tau \\
 & | & \texttt{type}\ t\ \{\overline{\tau_d}\} \\
 & | & \texttt{class}\ c\ \{\ \overline{\sigma_{cd}},\ \overline{\sigma_d}\ \} \\
 & | & \tau_d
\end{array}
$$

Wyvern Core 2: Translating Classes

OO Wyvern with Classes

```
class Option
   var quantity : Int = 0
   var price : Int = 0
   def exercise : Int = ...

   class var totalQuantityIssued : Int = 0
   class def issue(q : Int,
                   p : Int) : Option =
      new
         var quantity : Int = q
         var price : Int = p

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

OO Wyvern Core 1

```
type Option =
   def exercise : Int

type OptionClass =
   def issue : Int -> Int -> Option

var Option : OptionClass =
   new
      var totalQuantityIssued : Int = 0
      def issue(q : Int,
                p : Int) : Option =
         new
            var quantity : Int = q
            var price : Int = p
            def exercise : Int = ...

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

What's Next?

- Today, Work in Progress:
  - Adding Modules
  - Problems with Adding Type Members (*if time*)
- Other Work (*on request*):
  - Adding Nominal and Structural Types using Tags
  - Adding Type-Specific Languages
  - Delegation vs Inheritance
  - Wyvern VM and Implementation Work

Adding Modules by Darya Kurilova @ CMU

Motivation

- Untrusted third-party code is run side by side with trusted system code (e.g. website mashups, web browser extensions, mobile applications, cloud computing platforms)
- Some modules are dangerous (e.g. FFI and IO) or sensitive (e.g. containing user passwords, SSNs, medical records)
- Difficult to control module interaction

Goals

- Simplify specification of module access restrictions
- Automate and provably guarantee enforcement of module access restrictions
- Allow control and audit of module access through a small number of files

Wyvern's Approach

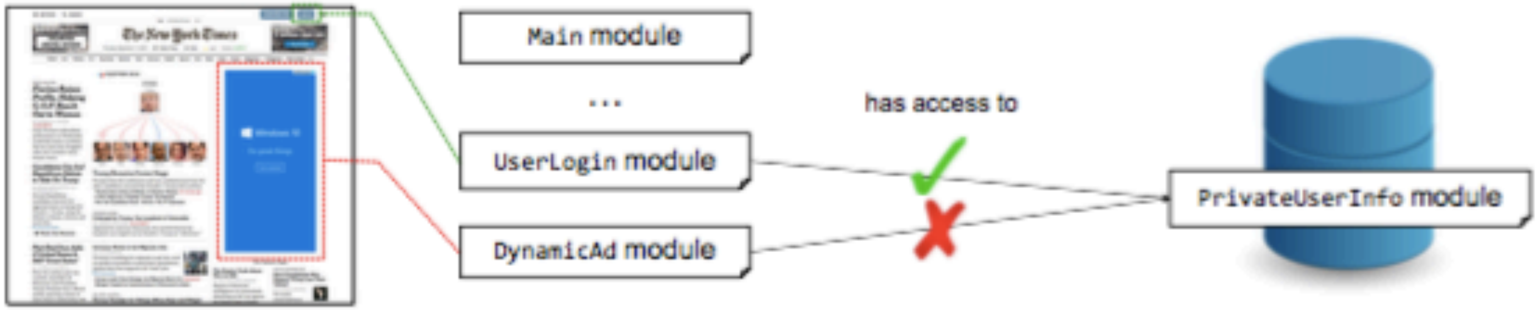
- Wyvern—secure-by-design programming language
- Capability-based approach
- Module access restrictions are enforced by the Wyvern type system
- Module access is controlled in a *single* file

Wyvern with Modules Example 1

```
resource module wyvern/examples/logging

import wyvern/collections/List
require filesystem

resource type Log
      def log(x:String)

def makeLog(path:String):Log
      val logFile = filesystem.openForAppend(path)
      val messageList = List.make()
      new
            def log(x:String)
                  messageList.append(x)
                  logFile.print(x)
```

```
require filesystem

instantiate wyvern/examples/logging(filesystem)
instantiate myapplication(logging)

myapplication.start()
```

Wyvern with Modules Example 2




```
resource module PrivateUserInfo
  var password = ...
  ...

resource module UserLogin
  require PrivateUserInfo
✓ var userPassword = PrivateUserInfo.password
  ...

resource module DynamicAd
  require PrivateUserInfo
✗ var userPassword = PrivateUserInfo.password
  ...

resource module Main
  instantiate PrivateUserInfo() as PUInfo
  instantiate UserLogin(PUInfo)
  instantiate DynamicAd()
```

Compilation error: Required access to `PrivateUserInfo` is not granted

- The **`resource`** keyword indicates that the module is or uses a dangerous or sensitive module
- A **`resource`** module must be **`required`** by modules that want to use it
- A **`required`** module must be **`instantiated`** by the `Main` module
- The `Main` module grants access to use a **`resource`** module (`PrivateUserInfo`) by explicitly passing it into the module that **`required`** it (`UserLogin`)
- Otherwise, the module (`DynamicAd`) is *forbidden* to use the **`resource`** module (`PrivateUserInfo`)
- The `Main` module is the *single* place of security and privacy control and audit

Wyvern Core 3A: Adding Modules

$p ::= e$		*program*
$m ::= h\ \overline{i}\ \overline{d}$		*module*
$h ::= [\texttt{resource}]\ \texttt{module}\ x : URI$		*module header*
$i ::= \texttt{import}\ URI\ [\texttt{as}\ x]$ $\mid\ \texttt{instantiate}\ URI(\overline{x})\ [\texttt{as}\ x]$ $\mid\ \texttt{require}\ URI\ [\texttt{as}\ x]$		
$sm ::= [\texttt{resource}]\ \texttt{signature}\ x = \tau$		*signatures module*
$d ::= ...$		*declarations*
$e ::= ...$		*expressions*

Wyvern Core 3A: Adding Modules

$$
\begin{array}{rcll}
e & ::= & x & \textit{expressions} \\
& | & \mathtt{new}_s(x \Rightarrow d) & \\
& | & e.m(e) & \\
& | & e.f & \\
& | & e.f = e & \\
& | & \mathtt{bind}\ x = e\ \mathtt{in}\ e & \\
& | & l & \textit{(run-time forms)} \\
& | & l.m(l) \rhd e &
\end{array}
$$

$$
s ::= \mathtt{stateful} \mid \mathtt{pure}
$$

$$
\begin{array}{rcll}
d & ::= & \epsilon & \textit{declarations} \\
& | & \mathtt{def}\ m(x : \tau) : \tau = e; d & \\
& | & \mathtt{var}\ f : \tau = x; d & \\
& | & \mathtt{var}\ f : \tau = l; d & \textit{(run-time form)}
\end{array}
$$

$$
\tau ::= \{\sigma\}_s \qquad \textit{types}
$$

$$
\begin{array}{rcll}
\sigma & ::= & \epsilon & \textit{decl. types} \\
& | & \mathtt{def}\ m : \tau \to \tau; \sigma & \\
& | & \mathtt{var}\ f : \tau; \sigma &
\end{array}
$$

$$
\begin{array}{rcll}
\Gamma & ::= & \varnothing & \textit{contexts} \\
& | & \Gamma,\ x : \tau &
\end{array}
$$

$$
\begin{array}{rcll}
\mu & ::= & \varnothing & \textit{store} \\
& | & \mu,\ l \mapsto \{x \Rightarrow d\}_s &
\end{array}
$$

$$
\begin{array}{rcll}
\Sigma & ::= & \varnothing & \textit{store type} \\
& | & \Sigma,\ l : \tau &
\end{array}
$$

$$
\begin{array}{rcll}
E & ::= & [\,] & \textit{evaluation contexts} \\
& | & E.m(e) & \\
& | & l.m(E) & \\
& | & E.f & \\
& | & E.f = e & \\
& | & \mathtt{bind}\ x = E\ \mathtt{in}\ e & \\
& | & l.f = E & \\
& | & l.m(l) \rhd E &
\end{array}
$$

Wyvern Modules Summary

- We prove an "authority safety theorem" that guarantees using our type system whether a module is stateful or pure based on a points-to relation.
- We provide a translation from the more abstract grammar to the base grammar very similar to Wyvern Cores and prove the latter sound.
- We are developing a threat/attacker model to be able to demonstrate our module access guarantees by utilising the capabilities.
- <u>Type members</u> are part of the module's signatures *(next step)*

Why Add Type Members to Wyvern?

- Much discussion of type members since Beta and gBeta and later Scala adopting them
- Type members can encode generics but are more expressive and require less annotations, e.g.

```
def copyCell(c:Cell):Cell
     new Cell
          type t = c.t
          val data : t = c.data
```

versus

```
def copyCell<T>(c:Cell<T>):Cell<T> ...
```

Why Add Type Members to Wyvern?

```
datatype DiverseTree
    case type Leaf
        type T
        val v:T
    case type Branch
        val t1:DiverseTree
        val t2:DiverseTree
```

Why Add Type Members to Wyvern?

```
type Table
     type Key
     type Value
     def get(k:Key):Value
     def add(v:Value):Key

// the Key type of the returned table is abstract
def newTable<ValueType>()
     :Table<Value=ValueType>
```

Wyvern Core 3B: Adding Type Members

$$
\begin{array}{llll}
e & ::= & x & \textit{expression} \\
& | & \mathtt{new}\ \{z \Rightarrow \overline{d}\} & \\
& | & e.m_T(e) & \\
& | & e.f & \\
& | & e \trianglelefteq T & \\
& | & l & \\
p & ::= & x & \textit{paths} \\
& | & l & \\
& | & p.f & \\
& | & p \trianglelefteq T & \\
v & ::= & l & \textit{value} \\
& | & v.f & \\
& | & v \trianglelefteq T & \\
d & ::= & \mathtt{val}\ f : T = p & \textit{declaration} \\
& | & \mathtt{def}\ m(x:T) = e : T & \\
& | & \mathtt{type}\ L : T..T & \\
\Gamma & ::= & \varnothing \mid \Gamma,\ x : T & \textit{Environment}
\end{array}
$$

$$
\begin{array}{llll}
T & ::= & \{z \Rightarrow \overline{\sigma}\} & \textit{type} \\
& | & p.L & \\
& | & \top & \\
& | & \bot & \\
\sigma & ::= & \mathtt{val}\ f : T & \textit{decl type} \\
& | & \mathtt{def}\ m : T \to T & \\
& | & \mathtt{type}\ L : T..T & \\
E & ::= & \bigcirc & \textit{eval context} \\
& | & E.m(e) & \\
& | & p.m(E) & \\
& | & E.f & \\
& | & E \trianglelefteq T & \\
d_v & ::= & \mathtt{val}\ f : T = v & \textit{declaration value} \\
& | & \mathtt{def}\ m(x:T) = e : T & \\
& | & \mathtt{type}\ L : T..T = T & \\
\mu & ::= & \varnothing \mid \mu,\ l \mapsto \{z \Rightarrow \overline{d}\} & \textit{store} \\
\Sigma & ::= & \varnothing \mid \Sigma,\ l : \{\mathbf{z} \Rightarrow \overline{\sigma}\} & \textit{store type}
\end{array}
$$

Adding Type Members by Julian Mackay @ VUW

- A lot of work in the 90's (including Atsushi).
- Wyvern Type Members are based on those in Scala.
- Recent work by Nada Amin, Tiark Rompf et al. on trying to prove a type system with full type members support sound (FOOL 2012, OOPSLA 2014, ongoing…)
- Issues with just proving preservation include:
  - Path equality problem (we *do not evaluate paths till required*)
  - Inability to resolve some type members during type checking due to environment narrowing (we *keep track of the declared type*)
  - Nonsensical expansions of declarations and loss of well formedness when combining environment narrowing and intersection types (we try to *avoid environment narrowing at all costs*)
  - Subtype transitivity problem (complex *mutual induction* in proofs)

Issue 1: Path Equality Problem

```
val b = new {z ⇒
              type L : ⊤ .. ⊤
              val l : z.L = b};
val a = new {z ⇒
              val i : {z ⇒
                        type L : ⊥ .. ⊤
                        val l : z.L} = b
              def meth : ⊤ (x : z.i.L){x}};
a.meth(a.i.l)
```

a.i.l reduces to b.l of type b.L but we can’t ensure that b.L <: a.i.L

Issue 2: Term Membership Restriction

```
Y = {z ⇒
      val l : ⊤
      def m : Y(y:Y){
        val a = new {};
        y
      }
    }

X = {z ⇒
      type L : ⊤ .. ⊤
      val l : z.L
      def m : Y(y:Y){
        y
      }
    }
```

the following expression is well typed.

```
val x = new X(l = z);
val y = new Y(l = (val z = new {}; z));
y.m(x).l
```

Unfortunately, small step reduction requires the following next expression to be well typed, which is not as we have nothing to substitute for z.l:

```
(val a = new {}; x).l
```

Therefore, our Method Reduction Rule is:

$$\frac{\mu \vdash \underline{v_1 \leadsto l} \qquad \mu(l) = \{\mathbf{z} \Rightarrow ..., m : T(x : S) = e, ...\}}{\mu \mid v_1.m_U(v_2) \ \rightarrow \mu \mid [l/\mathbf{z}, v_2 \underline{\trianglelefteq S}/x]e \underline{\trianglelefteq U}} \quad (\text{R-METH})$$

Issue 3: Expansion Lost

```
X = {z ⇒
      type A : ⊥ .. z.B
      type B : ⊥ .. ⊤
    }
Y = {z ⇒
      type A : ⊥ .. ⊤
      type B : ⊥ .. z.A
    }
```

While both these types are expandable, their intersection is not.

```
X ∧ Y = {z ⇒
          type A : ⊥ .. z.B
          type B : ⊥ .. z.A
        }
```

We can construct a less obvious expression that effectively results in the same contradictory intersection type…

Issue 4: Loss of Well-Formedness

```
S = {z ⇒
      type A : ⊥ .. List
    }
T = {z ⇒
      type A : Integer .. ⊤
    }
```

If we try and use the intersection of S and T we get.

```
S ∧ T = {z ⇒
          type A : Integer .. List
        }
```

Again, we can construct a less obvious expression that effectively results in the same contradictory intersection type implying two unrelated types subtype each other!

Issue 5: Subtype Transitivity Problem

- Subtype Transitivity is mutually dependent on Environment Narrowing:

$$\frac{\Gamma, (x : U); \Sigma \vdash T <: T' \qquad \Gamma; \Sigma \vdash S <: U}{\Gamma, (x : S); \Sigma \vdash T <: T'}$$

- Thus, we weaken environment narrowing proof by admitting subtype transitivity (same tactic as Amin et al.).
- Then we prove the relaxed subtype narrowing and subtype transitivity and show that the admitted subtyping judgement is equivalent to the original.
- Finally, thanks to Julian Mackay, our proofs are done both on paper and in COQ.

Wyvern Type Members Summary

- We have a preservation (and thus soundness as progress is easy) proof for a restricted language defined using small step semantics.
- We are working on other issues and the kinds of assumptions we can use in "OO world" that might not be acceptable in more "pure world" that would allow us to have a sound type system.
- We plan to extend our work by utilising the formal benefit of small step to capability reasoning (when merging our type members work with our modules work) and explore the implications for type refinement and graduate types in Wyvern.

Suggestions?