Wyvern Formalisation: Objects, Classes, Modules, Type Members

A/Prof Alex Potanin

VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA
NEW ZEALAND

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)

Wyvern

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

http://wyvernlang.github.io/

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


DYNAMIC TYPING
STATIC TYPING
DONE!

What’s Pure OO?

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

Wyvern Core 0: Extended Lambda

$$
\begin{array}{rcl}
\tau & ::= & \tau \rightarrow \tau \\
 & | & \{f_i{:}\tau_i^{i\in 1..n}\} \\
 & | & \texttt{ref}\ \tau \\
 & | & t \\
 & | & \boldsymbol{\mu} t.\tau
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \boldsymbol{\lambda} x{:}\tau.e \\
 & | & \{f_i = v_i^{i\in 1..n}\} \\
 & | & l \\
 & | & \texttt{fold}[\tau]\ v
\end{array}
$$

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

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \boldsymbol{\lambda} x{:}\tau.e \\
 & | & e(e) \\
 & | & \{f_i = e_i^{i\in 1..n}\} \\
 & | & e.f \\
 & | & \texttt{fix}\ e \\
 & | & \texttt{alloc}\ e \\
 & | & !e \\
 & | & e := e \\
 & | & \texttt{fold}[\tau]\ e \\
 & | & \texttt{unfold}[\tau]\ e \\
 & | & l
\end{array}
$$

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

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

Wyvern Core 1: Adding Objects

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

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

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

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

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

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

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

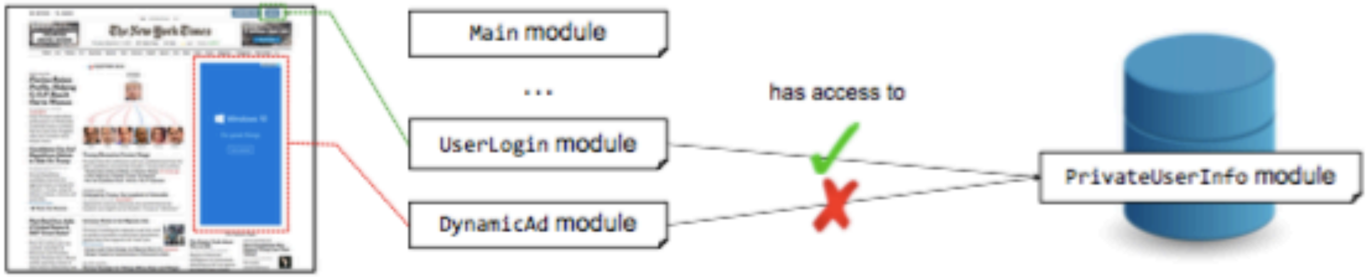
```
type Option =
   def exercise : Int

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
```

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

Carnegie Mellon University CyLab

Wyvern Core 3A: Adding Modules

$$
\begin{array}{llr}
p & ::= e & \textit{program} \\[1em]
m & ::= h\ \overline{i}\ \overline{d} & \textit{module} \\[1em]
h & ::= [\texttt{resource}]\ \texttt{module}\ x : URI & \textit{module header} \\[1em]
i & ::= \texttt{import}\ URI\ [\texttt{as}\ x] & \\
 & \mid\ \ \texttt{instantiate}\ URI(\overline{x})\ [\texttt{as}\ x] & \\
 & \mid\ \ \texttt{require}\ URI\ [\texttt{as}\ x] & \\[1em]
sm & ::= [\texttt{resource}]\ \texttt{signature}\ x = \tau & \textit{signatures module} \\[1em]
d & ::= \ldots & \textit{declarations} \\[1em]
e & ::= \ldots & \textit{expressions}
\end{array}
$$

Wyvern Core 3A: Adding Modules

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

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

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

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

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

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

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

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

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

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 depeloped 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

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

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

Adding Type Members by Julian Mackay @ VUW

- A lot of work in the 90's (including Atsushi Igarashi).
- 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 2016)
  - https://lampwww.epfl.ch/~amin/cv/
- Recent work also by Ondřej Lhoták:
  - https://plg.uwaterloo.ca/~olhotak/Publications.html
- 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)
  - And much more, so see my "Decidable Subtyping for Path Dependent Types" talk ☺