The Core Wyvern Language

The Plaid Group

July 11, 2013

1 The Simplest Core Language

This language is a simply typed lambda calculus (chapter 9) with records and recursion (chapter 11), references (chapter 13), subtyping (chapter 15), and iso-recursive types (chapters 20 and 21). This is completely standard, using rules straight from Pierce. Type semantics are iso-recursive, with explicit fold/unfold constructs. In practice we will fold automatically on record creation, and unfold automatically on field access; this will be handled when compiling higher-level languages down to this language. Finally, the soundness proofs and required lemmas need to be written out but should be straightforward. Note that we avoided the need for Unit Type by making the type of assignment match the type of the expression being assigned and the value of the assignment execution to be the value being assigned.

Definition 1 (Store Typing)**.** *A store S is well-typed, written $\Gamma|\Sigma \vdash S$ iff $dom(\Sigma) = dom(S)$ and $\forall l \in dom(S) : \Gamma|\Sigma \vdash S(l) : \Sigma(l)$.*

Theorem 1 (Preservation)**.** *If $\Gamma|\Sigma \vdash e{:}\tau$ and $\Gamma|\Sigma \vdash S$ and $e|S \leadsto e'|S'$, then $\exists\Sigma' \supseteq \Sigma$ such that $\Gamma|\Sigma' \vdash e'{:}\tau$ and $\Gamma|\Sigma' \vdash S'$.*

Proof. **[TODO: Should be straightforward from Pierce. Requires subsumption and other lemmas.]** □

Theorem 2 (Progress)**.** *Suppose e is a closed, well-typed term (that is, $\emptyset|\Sigma \vdash e{:}\tau$ for some τ and Σ). Then either e is a value or else, for any store S such that $\emptyset|\Sigma \vdash S$, there is some e' and S' with $e|S \leadsto e'|S'$.*

Proof. **[TODO: Should be straightforward from Pierce.]** □

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

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

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

Figure 1: Lambda Calculus with Extensions **[?]**

$$\frac{}{M \vdash \tau <: \tau}\ \textit{S-Refl} \quad \frac{M \vdash \tau_1 <: \tau_2 \quad M \vdash \tau_2 <: \tau_3}{M \vdash \tau_1 <: \tau_3}\ \textit{S-Trans}$$

$$\frac{M \vdash \tau_3 <: \tau_1 \quad M \vdash \tau_2 <: \tau_4}{M \vdash \tau_1 \to \tau_2 <: \tau_3 \to \tau_4}\ \textit{S-Arrow}$$

$$\frac{}{M \vdash \{f_i{:}\tau_i^{i\in 1..n+k}\} <: \{f_i{:}\tau_i^{i\in 1..n}\}}\ \textit{S-RcdWidth}$$

$$\frac{\text{for each } i\ M \vdash \tau_i <: {\tau'}_i}{M \vdash \{f_i{:}\tau_i^{i\in 1..n}\} <: \{f_i{:}{\tau'}_i^{i\in 1..n}\}}\ \textit{S-RcdDepth}$$

$$\frac{\{f_j{:}\tau_j^{j\in 1..n}\} \text{ is a permutation of } \{{f'}_i{:}{\tau'}_i^{i\in 1..n}\}}{M \vdash \{f_j{:}\tau_j^{j\in 1..n}\} <: \{{f'}_i{:}{\tau'}_i^{i\in 1..n}\}}\ \textit{S-RcdPerm}$$

$$\frac{M \vdash \tau <: \tau' \quad M \vdash \tau' <: \tau}{M \vdash \mathbf{ref}\ \tau <: \mathbf{ref}\ \tau'}\ \textit{S-Ref}$$

$$\frac{M, t <: t' \vdash \tau <: \tau'}{M \vdash \boldsymbol{\mu} t.\tau <: \boldsymbol{\mu} t'.\tau'}\ \textit{S-Amber} \quad \frac{t <: t' \in M}{M \vdash t <: t'}\ \textit{S-Assumption}$$

$$\frac{\Gamma|\Sigma \vdash e{:}\tau' \quad \emptyset \vdash \tau' <: \tau}{\Gamma|\Sigma \vdash e{:}\tau}\ \textit{T-Sub}$$

Figure 2: Subtyping Rules

$$\frac{x{:}\tau \in \Gamma}{\Gamma|\Sigma \vdash x{:}\tau}\ \textit{T-Var}$$

$$\frac{\Gamma, x{:}\tau_1|\Sigma \vdash e_2{:}\tau_2}{\Gamma|\Sigma \vdash \boldsymbol{\lambda} x{:}\tau_1.e_2{:}\tau_1 \rightarrow \tau_2}\ \textit{T-Abs} \quad \frac{\Gamma|\Sigma \vdash e_1{:}\tau_{11} \rightarrow \tau_{12} \quad \Gamma|\Sigma \vdash e_2{:}\tau_{11}}{\Gamma|\Sigma \vdash e_1(e_2){:}\tau_{12}}\ \textit{T-App}$$

$$\frac{\text{for each } i \quad \Gamma|\Sigma \vdash e_i{:}\tau_i}{\Gamma|\Sigma \vdash \{f_i = e_i\ ^{i\in 1..n}\}{:}\{f_i{:}\tau_i\ ^{i\in 1..n}\}}\ \textit{T-Rcd} \quad \frac{\Gamma|\Sigma \vdash e_1{:}\{f_i{:}\tau_i\ ^{i\in 1..n}\}}{\Gamma|\Sigma \vdash e_1.f_j{:}\tau_j}\ \textit{T-Proj}$$

$$\frac{\Gamma|\Sigma \vdash e_1{:}\tau_1 \quad \Gamma, x{:}\tau_1|\Sigma \vdash e_2{:}\tau_2}{\Gamma|\Sigma \vdash \textbf{let } x = e_1 \textbf{ in } e_2{:}\tau_2}\ \textit{T-Let} \quad \frac{\Gamma|\Sigma \vdash e_1{:}\tau_1 \rightarrow \tau_1}{\Gamma|\Sigma \vdash \textbf{fix } e_1{:}\tau_1}\ \textit{T-Fix}$$

$$\frac{\Sigma(l) = \tau}{\Gamma|\Sigma \vdash l{:}\textbf{ref } \tau}\ \textit{T-Loc} \quad \frac{\Gamma|\Sigma \vdash e{:}\tau}{\Gamma|\Sigma \vdash \textbf{alloc } e{:}\textbf{ref } \tau}\ \textit{T-Alloc} \quad \frac{\Gamma|\Sigma \vdash e{:}\textbf{ref } \tau}{\Gamma|\Sigma \vdash !e{:}\tau}\ \textit{T-Deref}$$

$$\frac{\Gamma|\Sigma \vdash e_1{:}\textbf{ref } \tau \quad \Gamma|\Sigma \vdash e_2{:}\tau}{\Gamma|\Sigma \vdash e_1 := e_2{:}\tau}\ \textit{T-Assign}$$

$$\frac{\tau = \boldsymbol{\mu} t.\tau_1 \quad \Gamma|\Sigma \vdash e : [t \mapsto \tau]\tau_1}{\Gamma|\Sigma \vdash \textbf{fold}[\tau]\ e : \tau}\ \textit{T-Fold} \quad \frac{\tau = \boldsymbol{\mu} t.\tau_1 \quad \Gamma|\Sigma \vdash e : \tau}{\Gamma|\Sigma \vdash \textbf{unfold}[\tau]\ e : [t \mapsto \tau]\tau_1}\ \textit{T-Unfold}$$

Figure 3: Typing Rules (Static Semantics)

$$\frac{e_1|S \rightsquigarrow e_1'|S'}{e_1(e_2)|S \rightsquigarrow e_1'(e_2)|S'}\ \textit{E-App1} \qquad \frac{e_2|S \rightsquigarrow e_2'|S'}{v_1(e_2)|S \rightsquigarrow v_1(e_2')|S'}\ \textit{E-App2}$$

$$\frac{}{(\boldsymbol{\lambda} x{:}\tau.e)v|S \rightsquigarrow [x \mapsto v]e|S}\ \textit{E-AppAbs}$$

$$\frac{e_j|S \rightsquigarrow e_j'|S'}{\begin{array}{c}\{f_i = v_i\ {}^{i\in 1..j-1}, f_j = e_j, f_k = e_k\ {}^{k\in j+1..n}\}|S \\ \rightsquigarrow \{f_i = v_i\ {}^{i\in 1..j-1}, f_j = e_j', f_k = e_k\ {}^{k\in j+1..n}\}|S'\end{array}}\ \textit{E-Rcd}$$

$$\frac{e|S \rightsquigarrow e'|S'}{e.f|S \rightsquigarrow e'.f|S'}\ \textit{E-Proj} \qquad \frac{}{\{f_i = v_i\ {}^{i\in 1..n}\}.f_j|S \rightsquigarrow v_j|S}\ \textit{E-ProjRcd}$$

$$\frac{}{\textbf{let}\ x = v\ \textbf{in}\ e|S \rightsquigarrow [x \mapsto v]e|S}\ \textit{E-LetV} \qquad \frac{e_1|S \rightsquigarrow e_1'|S'}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2|S \rightsquigarrow \textbf{let}\ x = e_1'\ \textbf{in}\ e_2|S}\ \textit{E-Let}$$

$$\frac{}{\textbf{fix}(\boldsymbol{\lambda}{:}\tau_1.e_2)|S \rightsquigarrow [x \mapsto (\textbf{fix}(\boldsymbol{\lambda}{:}\tau_1.e_2)]e_2|S}\ \textit{E-FixBeta} \qquad \frac{e|S \rightsquigarrow e'|S'}{\textbf{fix}\ e|S \rightsquigarrow \textbf{fix}\ e'|S'}\ \textit{E-Fix}$$

$$\frac{l \notin dom(S)}{\textbf{alloc}\ v|S \rightsquigarrow l|(S, l \mapsto v)}\ \textit{E-AllocV} \qquad \frac{e|S \rightsquigarrow e'|S'}{\textbf{alloc}\ e|S \rightsquigarrow \textbf{alloc}\ e'|S'}\ \textit{E-Alloc}$$

$$\frac{S(l) = v}{!l|S \rightsquigarrow v|S}\ \textit{E-DerefLoc} \qquad \frac{e|S \rightsquigarrow e'|S'}{!e|S \rightsquigarrow !e'|S'}\ \textit{E-Deref} \qquad \frac{}{l := v|S \rightsquigarrow v|[l \mapsto v]S}\ \textit{E-Assign}$$

$$\frac{e_1|S \rightsquigarrow e_1'|S'}{e_1 := e_2|S \rightsquigarrow e_1' := e_2|S'}\ \textit{E-Assign1} \qquad \frac{e_2|S \rightsquigarrow e_2'|S'}{e_1 := e_2|S \rightsquigarrow e_1 := e_2'|S'}\ \textit{E-Assign2}$$

$$\frac{e|S \rightsquigarrow e'|S'}{\textbf{fold}[\tau]\ e|S \rightsquigarrow \textbf{fold}[\tau]\ e'|S'}\ \textit{E-Fold} \qquad \frac{e|S \rightsquigarrow e'|S'}{\textbf{unfold}[\tau]\ e|S \rightsquigarrow \textbf{unfold}[\tau]\ e'|S'}\ \textit{E-Unfold}$$

$$\frac{}{\textbf{unfold}[\tau](\textbf{fold}[\tau_1]\ v)|S \rightsquigarrow v|S}\ \textit{E-UnfoldFold}$$

Figure 4: Evaluation Rules (Dynamic Semantics)

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

Figure 5: Featherweight Wyvern Syntax

2 A Method-Based Language

This language is the lambda calculus with iso-recursive types and mutable objects. It enforces the uniform access principle. It also encapsulates state within objects. The differences vs. the core language are as follows:

Contribution: OO language that is closest to lambda calculus, while enforcing the uniform access principle and encapsulating state.

- Distinguishes the "internal" type of a receiver that is known within a method, which contains both methods and mutable fields, from types that can be written in the external language, which only contain methods.
- Encodes references as mutable fields.
- Uses different operations for field dereference vs. method call

justify lazy eval: otherwise m acts as a field, we want computation and the uniform access principle

T-new probably needs to change to fold the result type.

Note: recursively bound occurrences in an internal type (σ) are of an external type. That is, the language does not support a "thistype".

$$\frac{\Gamma \vdash e : \tau}{\Gamma, \sigma \vdash \textbf{var}\ f : \tau = e :: \textbf{var}\ f : \tau}\ \textit{DT-var}$$

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

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

$$\frac{\Gamma \vdash [\tau/t]d}{\Gamma \vdash \textit{type}\ t\ \{\tau\}; d}\ \textit{T-typeab}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau_1}{\Gamma \vdash \boldsymbol{\lambda} x{:}\tau.e :: \tau \rightarrow \tau_1}\ \textit{T-abs}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textit{T-varx}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma = \{\textbf{var}\ f : \tau_1, ...\}}{\Gamma \vdash e.f : \tau_1}\ \textit{T-field}$$

$$\frac{\Gamma \vdash e : \tau \quad \tau = \{...\textbf{meth}\ m : \tau_1 = e_1, ...\}}{\Gamma \vdash e.m : \tau_1}\ \textit{T-meth2}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma = \{\textbf{var}\ f : \tau_1 = e_1, ...\} \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e.f = e_2 : \tau_1}\ \textit{T-assign}$$

$$\frac{\Gamma \vdash e : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e(e_1) : \tau_2}\ \textit{T-appl}$$

Figure 6: Static Semantics Rules Core 2

$$\frac{}{\tau <: \tau} \; \textit{T-refl}$$

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \; \textit{T-trans}$$

$$\frac{}{\{\textbf{meth } m_i{:}\tau_i^{i\in 1..n+k}\} <: \{\textbf{meth } m_i{:}\tau_i^{i\in 1..n}\}} \; \textit{T-RcdWidth}$$

$$\frac{\text{for each } i \; \tau_i <: {\tau'}_i}{\{\textbf{meth } m_i{:}\tau_i^{i\in 1..n}\} <: \{\textbf{meth } m_i{:}{\tau'}_i^{i\in 1..n}\}} \; \textit{T-RcdDepth}$$

$$\frac{\{\textbf{meth } m_j{:}\tau_j^{j\in 1..n}\} \text{ is a permutation of } \{\textbf{meth } {m'}_i{:}{\tau'}_i^{i\in 1..n}\}}{\{\textbf{meth } m_j{:}\tau_j^{j\in 1..n}\} <: \{\textbf{meth } {m'}_i{:}{\tau'}_i^{i\in 1..n}\}} \; \textit{T-RcdPerm}$$

$$\frac{\tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4} \; \textit{T-Arrow}$$

$$\frac{\tau_1 <: \tau_2}{\textbf{meth } m : \tau_1 <: \textbf{meth } m : \tau_2} \; \textit{T-SubMeth}$$

$$\frac{\tau_1 <: \tau_1}{\textbf{var} f : \tau_1 <: \textbf{var} f : \tau_1} \; \textit{T-SubVar}$$

$$\frac{\sigma_1 <: \sigma_2}{\textbf{new}\sigma_1\{\overline{d_1}\} <: \textbf{new}\sigma_2\{\overline{d_2}\}} \; \textit{T-SubNew}$$

Figure 7: Subtyping Rules Core 2

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

Figure 8: Translation from Featherweight Wyvern to the Extended Lambda Calculus

[TODO: Ligia: Give translation rules to core 1. Need T-type rule. Need store and store type. Need subtyping. Where is μ used in the typing rules? show off: classes are first class(but point out distinction from smalltalk)- write one example, enforces uniform access principle; related work]

2.1 Example Program in the Method-Based Language

The code below uses **val** for readability; assume **val** x = e1; e2 is equivalent to (**fn** x => e2)(e1). We also use **fn** x : **type** => e in place of $\boldsymbol{\lambda} x{:}\tau.e$, and we write **rec** for $\boldsymbol{\mu}$.

```
type t { meth add : int -> t }
type ti { var f : int
    type t3 { int -> t }
    meth add : t3 }
val o : t = new ti
  var f : int = 1
  type t3 { int -> t }
  meth add : t3 =
    fn x : int =>
      this.f = this.f + x
      this
val o3 : t = o.add(2)
```

2.2 Translation of the Program to the Core Language

We assume global type abbreviations because the translated code would be unintelligible without thenm. They can be eliminated by capture-avoiding substitution. We also assume val declarations, which can be encoded with functions and function calls.

```
type unit = {} // standard prelude
type t { add : int -> t }

type ti { f : ref int
    add : int -> t }

val o =
  letrec this : ti = {
    add = fn _ : unit => fn x : int =>
      this.f = !this.f + x
      fold[t] this
    f = alloc 1
  } in fold[t] this

val o3 = (unfold[t] o).add()(2)
```

Tasks:

- write a rewriting rule for new expressions, translating methods to lambdas, translating types appropriately, and translating vars to refs

- give complete rewriting rules (R*) to the core language
- prove that well-typed source programs translate to well-typed core programs. Is it possible to prove a property related to the uniform access principle and/or state encapsulation?
- argue this is OO in the sense of Cook. Since the body of a method is evaluated on every access to an object, it seems to qualify.

3 Example Factorial

```
type t { meth factorial : int -> t }
type ti { var f : int
    meth factorial : int -> t }
val o : t = new ti
  var f : int = 1
  type t3 { int -> t }
  meth factorial : t3 =
    fn x : int =>
                  if x>1
           this.f = x * (factorial(x-1)).f
                  else
                       this.f = 1
       this
val o3 : t = o.factorial(10)
```

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

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

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

Figure 9: Syntax of Featherweight Wyvern with Classes

4 A Class-Based Language

This version adds classes and shows how to rewrite them in terms of more primitive constructs. It is a true subset of the real Wyvern language.

Note that {} are used in the abstract syntax as an abbreviation for an indented block.

$$\frac{}{\Gamma, \_ \vdash \textbf{type}\ t\ \{\overline{\tau_d}\} :: \textbf{type}\ t\ \{\overline{\tau_d}\}}\ \textit{DT-type}$$

$$\frac{\Gamma, \sigma_{this} \vdash \overline{d} :: \overline{\sigma_d} \quad \Gamma, \sigma_{this} \vdash \overline{cd} :: \overline{\sigma_{cd}} \quad \sigma_{this} = \{\overline{\sigma_{cd}}, \overline{\sigma_d}\}}{\Gamma, \_ \vdash \textbf{class}\ c\ \{\overline{cd}, \overline{d}\} :: \textbf{class}\ c\ \{\overline{cd}, \overline{d}\}}\ \textit{DT-class}$$

$$\frac{\Gamma, \sigma_{this} \vdash e : \tau_2 \quad \tau_2 <: \tau_1}{\Gamma, \sigma_{this} \vdash \textbf{class var}\ f : \tau_1 = e_1 :: \textbf{class var}\ f : \tau_1}\ \textit{DT-class-var}$$

$$\frac{\Gamma, \sigma_{this} \vdash e : \tau_2 \quad \tau_2 <: \tau_1}{\Gamma, \sigma_{this} \vdash \textbf{class meth}\ m : \tau_1 = e_1 :: \textbf{class meth}\ m : \tau_1}\ \textit{DT-class-meth}$$

$$\frac{\Gamma, \sigma_{this} \vdash \overline{d_1} :: \overline{\sigma_{d_1}} \quad \{\overline{\sigma_{d_1}}\} \cup \sigma_{this} <: \{\overline{\tau_d}\}}{\Gamma, \sigma_{this} \vdash \textbf{new}\ \{\overline{d_1}\} : \{\overline{\tau_d}\}}\ \textit{T-new}$$

$$\frac{\Gamma, \sigma_{this} \vdash e : \sigma_1 \quad \sigma_1 = \{\textbf{class var}\ f : \tau_1, ...\}}{\Gamma, \sigma_{this} \vdash e.f : \tau_1}\ \textit{T-class-field}$$

$$\frac{\Gamma, \sigma_{this} \vdash e : \tau_1 \quad \tau_1 = \{\textbf{class meth}\ m : \tau_1 = e_1, ...\}}{\Gamma, \sigma_{this} \vdash e.m : \tau_1}\ \textit{T-class-meth}$$

Figure 10: Static Semantics Rules Core 3

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

Figure 11: Translation of a Class from FWC to FW

```
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 12: An `Option` Class in Featherweight Wyvern with Classes

4.1 Example Program and Translation by Darya

```
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 13: `Option` Class Translated to Featherweight Wyvern

```
type t
  meth add : int -> t
  meth get : int

class c
  var f : int
  meth add : int -> c =
    fn x : int =>
      this.f = this.f + x
      this
  meth get : int
  meth equals : c -> bool =
    fn other : c =>
      this.f == other.get  // cannot access other.f in type c
                           // because this core doesn't support ADTs
  class var nc : int = 1
  class meth make : int -> c =
    fn x:int =>
      nc = nc + 1
      new
        var f:int = x
        meth get : int = this.f

val o : c = c.make(4)
val o2 : t = o.add(2)
var x6 : int = o.get
```

Figure 14: Example Program in Featherweight Wyvern

4.2 Example Program in the Class-Based Language (By Jonathan)

4.3 Translation of the Program to the Core Method-Based Language (By Jonathan)

Limitations: this language only supports objects, not ADTs. For ADTs we need bounded type members.

```
type t = rec t2.
  meth add : int -> t2
  meth get : int

type c = rec c2.
  meth add : int -> c2
  meth get : int
  meth equals : c2 -> bool

type c_internal = rec ci2.  // not necessary, but a convenient abbreviation
  var f : int
  meth add : int -> c
  meth get : int
  meth equals : c -> bool

type c_class = rec cl2.  // not necessary
  meth make : int -> c

type c_class_internal = rec cli2.  // not necessary
  var nc : int
  meth make : int -> c

val c : c_class
  = new c_class_internal
    var nc : int = 1
    meth make : int -> c =
      fn x:int =>
        nc = nc + 1
        new c_internal
          var f : int = x
          meth get : int = this.f
          meth add : int -> c =
            fn x : int =>
              this.f = this.f + x
              this
          meth equals : c -> bool =
            fn other : c =>
              this.f == other.get  // cannot access other.f in type c

val o : c = c.make(4)
val o2 : t = o.add(2)
var x6 : int = o.get
```

Figure 15: Example Program in FW Translated to OO Core without Classes

4.4 Tasks

- write some examples!
- define stripClass, rewriteNew, and a way of computing τ_i
- add lots of conveniences as sugar
- in rule R-class, meth c needs to return the same object each time, so cache it in a field.
- give complete rewriting rules (R*) to the core language
- give complete typing rules, and prove that well-typed source programs translate to well-typed core programs. Is it possible to prove a property related to the uniform access principle and/or state encapsulation?
- consider "class type t = ..."
- no abstract class members
- no class class members because a class is always a class member; if you want a class in an object use a val + type

$$
\begin{array}{lcll}
u & ::= & \textbf{module}\ \mathit{url} : \tau\ \bar{i}\ \bar{d} & \\
\\
i & ::= & \textbf{import}\ \mathit{url} & \\
 & | & \textbf{import}\ \mathit{url} : \tau\ \textbf{as}\ x & \\
\\
d & ::= & \textbf{public}_{\mathit{opt}}\ \textbf{var}\ f : \tau = e & \\
 & | & \textbf{public}_{\mathit{opt}}\ \textbf{val}\ f : \tau = e & \\
 & | & \textbf{public}_{\mathit{opt}}\ \textbf{def}\ m : \tau = e & \\
 & | & \textbf{public}_{\mathit{opt}}\ \textbf{type}\ t\ = \{\overline{\tau_d}\} & \\
\\
\tau_d & ::= & \textbf{constant}_{\mathit{opt}}\ \textbf{def}\ m : \tau & \\
 & | & \textbf{type}\ t\ = \{\tau\} & \\
\\
\sigma & ::= & \tau & \\
 & | & \{\overline{\sigma_d}\} & \\
\\
\sigma_d & ::= & \textbf{var}\ f : \tau & \\
 & | & \tau_d &
\end{array}
$$

$$
\begin{array}{lcll}
e & ::= & x & \\
 & | & \boldsymbol{\lambda} x{:}\tau.e & \\
 & | & e(e) & \\
 & | & \textbf{new}\ \{\bar{d}\} & \\
 & | & e.f & \\
 & | & e.f = e & \\
 & | & e.m & \\
\\
\tau & ::= & p\,t & \\
 & | & \tau \rightarrow \tau & \\
\\
p & ::= & \epsilon & \\
 & | & x. & \\
 & | & p\ f. & \textit{f must be constant}
\end{array}
$$

Figure 16: Featherweight Wyvern Module Syntax

5 Module System

This language extends the method-based language with a simple module system.

6 Example

WRITE ME!!!

7 Next Steps

Things to add, in approximate order:

1. prop in types, and field syntax for dereference/assignment of properties (sugar; easy)

2. classes, class members, and letrec (mostly sugar, but letrec will need to be added to the core as it cannot be done by translation–or can it? may need to stop and implement type members and type parameters)

3. type members and type parameters (affects core; requires pure methods)

4. option types (sugar)

5. abstract types, with and without type bounds; class members in types (as sugar)

Longer-term considerations, in approximate order:

1. tags, pattern matching, and tag tests (need singleton types - affects core)

2. modules (design not yet clear)

3. inheritance, delegation, or another reuse mechanism (design not yet clear)

4. formalizing the concrete syntax, and translation into abstract syntax (indentation)