

The Core Wyvern Language

The Plaid Group

July 23, 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 $\text{dom}(\Sigma) = \text{dom}(S)$ and $\forall l \in \text{dom}(S) : \Gamma|\Sigma \vdash S(l) : \Sigma(l)$.*

Theorem 1 (Preservation). *If $\Gamma|\Sigma \vdash e:\tau$ and $\Gamma|\Sigma \vdash S$ and $e|S \rightsquigarrow e'|S'$, 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 $e|S \rightsquigarrow e'|S'$.*

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

$\tau ::= \tau \rightarrow \tau$		$e ::= x$
$\mid \{f_i:\tau_i^{i \in 1..n}\}$	$v ::= \lambda x:\tau.e$	$\mid \lambda x:\tau.e$
$\mid \mathbf{ref} \ \tau$	$\mid \{f_i = v_i^{i \in 1..n}\}$	$\mid e(e)$
$\mid t$	$\mid \ell$	$\mid \{f_i = e_i^{i \in 1..n}\}$
$\mid \mu t.\tau$	$\mid \mathbf{fold}[\tau] \ v$	$\mid e.f$
		$\mid \mathbf{fix} \ e$
		$\mid \mathbf{alloc} \ e$
		$\mid !e$
$\Gamma ::= \{\bar{x}:\bar{\tau}\}$	$S ::= \{\bar{l} = \bar{v}\}$	$\mid e := e$
$\Sigma ::= \{\bar{l}:\bar{\tau}\}$		$\mid \mathbf{fold}[\tau] \ e$
		$\mid \mathbf{unfold}[\tau] \ e$
		$\mid l$

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

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

Figure 1: Lambda Calculus with Extensions

$$\begin{array}{c}
\frac{}{M \vdash \tau <: \tau} \text{S-Refl} \quad \frac{M \vdash \tau_1 <: \tau_2 \quad M \vdash \tau_2 <: \tau_3}{M \vdash \tau_1 <: \tau_3} \text{S-Trans} \\
\\
\frac{M \vdash \tau_3 <: \tau_1 \quad M \vdash \tau_2 <: \tau_4}{M \vdash \tau_1 \rightarrow \tau_2 <: \tau_3 \rightarrow \tau_4} \text{S-Arrow} \\
\\
\frac{}{M \vdash \{f_i:\tau_i^{i \in 1..n+k}\} <: \{f_i:\tau_i^{i \in 1..n}\}} \text{S-RcdWidth} \\
\\
\frac{\text{for each } i \ M \vdash \tau_i <: \tau'_i}{M \vdash \{f_i:\tau_i^{i \in 1..n}\} <: \{f_i:\tau'_i^{i \in 1..n}\}} \text{S-RcdDepth} \\
\\
\frac{\{f_j:\tau_j^{j \in 1..n}\} \text{ is a permutation of } \{f'_i:\tau'_i^{i \in 1..n}\}}{M \vdash \{f_j:\tau_j^{j \in 1..n}\} <: \{f'_i:\tau'_i^{i \in 1..n}\}} \text{S-RcdPerm} \\
\\
\frac{M \vdash \tau <: \tau' \quad M \vdash \tau' <: \tau}{M \vdash \mathbf{ref} \ \tau <: \mathbf{ref} \ \tau'} \text{S-Ref} \\
\\
\frac{M, t <: t' \vdash \tau <: \tau'}{M \vdash \mu t. \tau <: \mu t'. \tau'} \text{S-Amber} \quad \frac{t <: t' \in M}{M \vdash t <: t'} \text{S-Assumption} \\
\\
\frac{\Gamma|\Sigma \vdash e:\tau' \quad \emptyset \vdash \tau' <: \tau}{\Gamma|\Sigma \vdash e:\tau} \text{T-Sub}
\end{array}$$

Figure 2: Subtyping Rules

$$\begin{array}{c}
\frac{x:\tau \in \Gamma}{\Gamma|\Sigma \vdash x:\tau} \textit{T-Var} \\
\\
\frac{\Gamma, x:\tau_1|\Sigma \vdash e_2:\tau_2}{\Gamma|\Sigma \vdash \lambda x:\tau_1. e_2:\tau_1 \rightarrow \tau_2} \textit{T-Abs} \quad \frac{\Gamma|\Sigma \vdash e_1:\tau_{11} \rightarrow \tau_{12} \quad \Gamma|\Sigma \vdash e_2:\tau_{11}}{\Gamma|\Sigma \vdash e_1(e_2):\tau_{12}} \textit{T-App} \\
\\
\frac{\text{for each } i \quad \Gamma|\Sigma \vdash e_i:\tau_i}{\Gamma|\Sigma \vdash \{f_i = e_i^{i \in 1..n}\}:\{f_i:\tau_i^{i \in 1..n}\}} \textit{T-Rcd} \quad \frac{\Gamma|\Sigma \vdash e_1:\{f_i:\tau_i^{i \in 1..n}\}}{\Gamma|\Sigma \vdash e_1.f_j:\tau_j} \textit{T-Proj} \\
\\
\frac{\Gamma|\Sigma \vdash e_1:\tau_1 \quad \Gamma, x:\tau_1|\Sigma \vdash e_2:\tau_2}{\Gamma|\Sigma \vdash \textbf{let } x = e_1 \textbf{ in } e_2:\tau_2} \textit{T-Let} \quad \frac{\Gamma|\Sigma \vdash e_1:\tau_1 \rightarrow \tau_1}{\Gamma|\Sigma \vdash \textbf{fix } e_1:\tau_1} \textit{T-Fix} \\
\\
\frac{\Sigma(l) = \tau}{\Gamma|\Sigma \vdash l:\textbf{ref } \tau} \textit{T-Loc} \quad \frac{\Gamma|\Sigma \vdash e:\tau}{\Gamma|\Sigma \vdash \textbf{alloc } e:\textbf{ref } \tau} \textit{T-Alloc} \quad \frac{\Gamma|\Sigma \vdash e:\textbf{ref } \tau}{\Gamma|\Sigma \vdash !e:\tau} \textit{T-Deref} \\
\\
\frac{\Gamma|\Sigma \vdash e_1:\textbf{ref } \tau \quad \Gamma|\Sigma \vdash e_2:\tau}{\Gamma|\Sigma \vdash e_1 := e_2:\tau} \textit{T-Assign} \\
\\
\frac{\tau = \mu t.\tau_1 \quad \Gamma|\Sigma \vdash e : [t \mapsto \tau]\tau_1}{\Gamma|\Sigma \vdash \textbf{fold}[\tau] e : \tau} \textit{T-Fold} \quad \frac{\tau = \mu t.\tau_1 \quad \Gamma|\Sigma \vdash e : \tau}{\Gamma|\Sigma \vdash \textbf{unfold}[\tau] e : [t \mapsto \tau]\tau_1} \textit{T-Unfold}
\end{array}$$

Figure 3: Typing Rules (Static Semantics)

$$\begin{array}{c}
\frac{e_1|S \rightsquigarrow e'_1|S'}{e_1(e_2)|S \rightsquigarrow e'_1(e_2)|S'} \text{ E-App1} \quad \frac{e_2|S \rightsquigarrow e'_2|S'}{v_1(e_2)|S \rightsquigarrow v_1(e'_2)|S'} \text{ E-App2} \\
\\
\frac{}{(\lambda x:\tau.e)v|S \rightsquigarrow [x \mapsto v]e|S} \text{ E-AppAbs} \\
\\
\frac{e_j|S \rightsquigarrow e'_j|S'}{\frac{\{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'} \text{ E-Rcd}} \text{ E-Rcd} \\
\\
\frac{e|S \rightsquigarrow e'|S'}{e.f|S \rightsquigarrow e'.f|S'} \text{ E-Proj} \quad \frac{}{\{f_i = v_i^{i \in 1..n}\}.f_j|S \rightsquigarrow v_j|S} \text{ E-ProjRcd} \\
\\
\frac{}{\mathbf{let} \ x = v \ \mathbf{in} \ e|S \rightsquigarrow [x \mapsto v]e|S} \text{ E-LetV} \quad \frac{e_1|S \rightsquigarrow e'_1|S'}{\mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2|S \rightsquigarrow \mathbf{let} \ x = e'_1 \ \mathbf{in} \ e_2|S} \text{ E-Let} \\
\\
\frac{}{\mathbf{fix}(\lambda:\tau_1.e_2)|S \rightsquigarrow [x \mapsto (\mathbf{fix}(\lambda:\tau_1.e_2))]e_2|S} \text{ E-FixBeta} \quad \frac{e|S \rightsquigarrow e'|S'}{\mathbf{fix} \ e|S \rightsquigarrow \mathbf{fix} \ e'|S'} \text{ E-Fix} \\
\\
\frac{l \notin \text{dom}(S)}{\mathbf{alloc} \ v|S \rightsquigarrow l|(S, l \mapsto v)} \text{ E-AllocV} \quad \frac{e|S \rightsquigarrow e'|S'}{\mathbf{alloc} \ e|S \rightsquigarrow \mathbf{alloc} \ e'|S'} \text{ E-Alloc} \\
\\
\frac{S(l) = v}{!l|S \rightsquigarrow v|S} \text{ E-DerefLoc} \quad \frac{e|S \rightsquigarrow e'|S'}{!e|S \rightsquigarrow !e'|S'} \text{ E-Deref} \quad \frac{}{l := v|S \rightsquigarrow v|[l \mapsto v]S} \text{ E-Assign} \\
\\
\frac{e_1|S \rightsquigarrow e'_1|S'}{e_1 := e_2|S \rightsquigarrow e'_1 := e_2|S'} \text{ E-Assign1} \quad \frac{e_2|S \rightsquigarrow e'_2|S'}{e_1 := e_2|S \rightsquigarrow e_1 := e'_2|S'} \text{ E-Assign2} \\
\\
\frac{e|S \rightsquigarrow e'|S'}{\mathbf{fold}[\tau] \ e|S \rightsquigarrow \mathbf{fold}[\tau] \ e'|S'} \text{ E-Fold} \quad \frac{e|S \rightsquigarrow e'|S'}{\mathbf{unfold}[\tau] \ e|S \rightsquigarrow \mathbf{unfold}[\tau] \ e'|S'} \text{ E-Unfold} \\
\\
\frac{}{\mathbf{unfold}[\tau](\mathbf{fold}[\tau_1] \ v)|S \rightsquigarrow v|S} \text{ E-UnfoldFold}
\end{array}$$

Figure 4: Evaluation Rules (Dynamic Semantics)

$e ::= x$	$d ::= \mathbf{var} \ f : \tau = e$
$\quad \quad \lambda x : \tau. e$	$\quad \quad \mathbf{meth} \ m : \tau = e$
$\quad \quad e(e)$	$\quad \quad \mathbf{type} \ t = \{\overline{\tau_d}, attributes = e\}$
$\quad \quad \mathbf{new} \ \{\overline{d}\}$	
$\quad \quad e.f$	$\tau_d ::= \mathbf{meth} \ m : \tau$
$\quad \quad e.f = e$	
$\quad \quad e.m$	$\sigma ::= \tau$
	$\quad \quad \{\overline{\sigma_d}\}$
$\tau ::= t$	
$\quad \quad \tau \rightarrow \tau$	$\sigma_d ::= \mathbf{var} \ f : \tau$
	$\quad \quad \mathbf{type} \ t = \{\tau\}$
	$\quad \quad \tau_d$

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”.

$$\begin{array}{c}
\frac{\Gamma \vdash e : \tau}{\Gamma, \sigma \vdash \mathbf{var} \ f : \tau = e :: \mathbf{var} \ f : \tau} \text{DT-var} \\
\\
\frac{\Gamma, \text{this} : \sigma \vdash e : \tau}{\Gamma, \sigma \vdash \mathbf{meth} \ m : \tau = e :: \mathbf{meth} \ m : \tau} \text{DT-meth} \\
\\
\frac{\Gamma, \sigma \vdash \bar{d} :: \bar{\sigma}_d \quad \sigma = \{\bar{\sigma}_d\} \quad \bar{\tau}_d \subseteq \bar{\sigma}_d}{\Gamma \vdash \mathbf{new} \ \sigma\{\bar{d}\} : \{\bar{\tau}_d\}} \text{T-new} \\
\\
\frac{\Gamma \vdash [\tau/t]d}{\Gamma \vdash \text{type } t = \{\tau\}; d} \text{T-typeab} \\
\\
\frac{\Delta, \Gamma \vdash e : t1 \quad \Gamma, t : \{\bar{\tau}_d, t1\} \vdash d \quad t1 : \{\bar{\tau}_d\}}{\Delta, \Gamma \vdash \text{type } t = \{\bar{\tau}_d, \text{attributes} = e\}; d} \text{T-attrib} \\
\\
\frac{t : \{\bar{\tau}_d, t1\} \in \Delta \quad t1 : \{\bar{\tau}_d', -\} \in \Delta \quad \mathbf{meth} : \tau \in \bar{\tau}_d'}{\Delta, \Gamma \vdash t.m : \tau} \text{Type-meth} \\
\\
\frac{\Gamma, x : \tau \vdash e : \tau_1}{\Gamma \vdash \lambda x : \tau. e :: \tau \rightarrow \tau_1} \text{T-abs} \\
\\
\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \text{T-varx} \\
\\
\frac{\Gamma \vdash e : \sigma \quad \sigma = \{\mathbf{var} \ f : \tau_1, \dots\}}{\Gamma \vdash e.f : \tau_1} \text{T-field} \\
\\
\frac{\Gamma \vdash e : \tau \quad \tau = \{\dots \mathbf{meth} \ m : \tau_1 = e_1, \dots\}}{\Gamma \vdash e.m : \tau_1} \text{T-meth2} \\
\\
\frac{\Gamma \vdash e : \sigma \quad \sigma = \{\mathbf{var} \ f : \tau_1 = e_1, \dots\} \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e.f = e_2 : \tau_1} \text{T-assign} \\
\\
\frac{\Gamma \vdash e : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e(e_1) : \tau_2} \text{T-appl}
\end{array}$$

Figure 6: Static Semantics Rules Core 2

$$\begin{array}{c}
\frac{}{\tau <: \tau} \textit{T-refl} \\
\\
\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3} \textit{T-trans} \\
\\
\frac{}{\{\mathbf{meth} \ m_i : \tau_i^{i \in 1..n+k}\} <: \{\mathbf{meth} \ m_i : \tau_i^{i \in 1..n}\}} \textit{T-RcdWidth} \\
\\
\frac{\text{for each } i \ \tau_i <: \tau'_i}{\{\mathbf{meth} \ m_i : \tau_i^{i \in 1..n}\} <: \{\mathbf{meth} \ m_i : \tau'_i^{i \in 1..n}\}} \textit{T-RcdDepth} \\
\\
\frac{\{\mathbf{meth} \ m_j : \tau_j^{j \in 1..n}\} \text{ is a permutation of } \{\mathbf{meth} \ m'_i : \tau'_i^{i \in 1..n}\}}{\{\mathbf{meth} \ m_j : \tau_j^{j \in 1..n}\} <: \{\mathbf{meth} \ m'_i : \tau'_i^{i \in 1..n}\}} \textit{T-RcdPerm} \\
\\
\frac{\tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\tau_1 \rightarrow \tau_2 <: \tau_3 \rightarrow \tau_4} \textit{T-Arrow} \\
\\
\frac{\tau_1 <: \tau_2}{\mathbf{meth} \ m : \tau_1 <: \mathbf{meth} \ m : \tau_2} \textit{T-SubMeth} \\
\\
\frac{\tau_1 <: \tau_1}{\mathbf{var} \ f : \tau_1 <: \mathbf{var} \ f : \tau_1} \textit{T-SubVar} \\
\\
\frac{\sigma_1 <: \sigma_2}{\mathbf{new}_{\sigma_1}\{\bar{d}_1\} <: \mathbf{new}_{\sigma_2}\{\bar{d}_2\}} \textit{T-SubNew}
\end{array}$$

Figure 7: Subtyping Rules Core 2

$$\begin{array}{ll}
\text{trans}(\mathbf{meth} \ m : \tau) & \equiv \ m : \mathbf{unit} \rightarrow \tau \\
\text{trans}(\mathbf{var} \ f : \tau) & \equiv \ f : \mathbf{ref} \ \tau \\
\text{trans}(\mathbf{meth} \ m : \tau = e; \bar{d}) & \equiv \ m : \tau = \lambda_ : \mathbf{unit}.\text{trans}(e); \\
& \quad \text{trans}(\bar{d}) \\
\text{trans}(\mathbf{var} \ f : \tau = e; \bar{d}) & \equiv \ f : \tau := \mathbf{alloc} \ \text{trans}(e); \\
& \quad \text{trans}(\bar{d}) \\
\text{trans}(e.m) & \equiv \ (\mathbf{unfold} \ \text{trans}(e)).m() \\
\text{trans}(\mathbf{new} \ \{\bar{d}\}) & \equiv \ \mathbf{letrec} \ \text{this} : \sigma = \\
& \quad \{\text{trans}(\bar{d})\} \\
& \quad \mathbf{in} \ \mathbf{fold} \ \text{this} \\
& \quad \text{where } \Gamma, \sigma \vdash \{\bar{d}\} : \sigma \\
\text{trans}(e.f = e_1) & \equiv \ \text{trans}(e).f = \text{trans}(e_1) \\
\text{trans}(e.f) & \equiv \ !\text{trans}(e).f \\
\text{trans}(\mathbf{type} \ t = \{\bar{\tau}_d\}; \bar{d}) & \equiv \ \text{trans}([\mu t.\text{trans}(\bar{\tau}_d)/t]\bar{d})
\end{array}$$

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** $x \Rightarrow e2$) $(e1)$. We also use **fn** $x : \text{type} \Rightarrow e$ in place of $\lambda x:\tau.e$, and we write **rec** for μ .

```
1 type t { meth add : int -> t }
2 type ti { var f : int
3         type t3 { int -> t }
4         meth add : t3 }
5 val o : t = new ti
6   var f : int = 1
7   type t3 { int -> t }
8   meth add : t3 =
9     fn x : int =>
10       this.f = this.f + x
11     this
12 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 them. They can be eliminated by capture-avoiding substitution. We also assume **val** declarations, which can be encoded with functions and function calls.

```
1 type unit = {} //standard prelude
2 type t { add : int -> t }
3
4 type ti { f : ref int
5         add : int -> t }
6
7 val o =
8   letrec this : ti = {
9     add = fn _ : unit => fn x : int =>
10       this.f = !this.f + x
11       fold[t] this
12     f = alloc 1
13   } in fold[t] this
14
15 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

```

1  type t { meth factorial : int -> t }
2  type ti { var f : int
3      meth factorial : int -> t }
4  val o : t = new ti
5      var f : int = 1
6      type t3 { int -> t }
7      meth factorial : t3 =
8          fn x : int =>
9              if x>1
10                 this.f = x * (factorial(x-1)).f
11                 else
12                     this.f = 1
13         this
14  val o3 : t = o.factorial(10)

```

e	$::=$	x		σ	$::=$	τ
		$\lambda x:\tau.e$				$\{\overline{\sigma_{cd}}\}$
		$e(e)$	d	$::=$		
		$\mathbf{new} \{\overline{d}\}$			$\mathbf{var} f : \tau = e$	
		$e.f$			$\mathbf{meth} m : \tau = e$	
		$e.f = e$			$\mathbf{type} t \{\overline{\tau_d}\}$	$\sigma_{cd} ::=$
		$e.m$			$\mathbf{class} c \{\overline{cd}; \overline{d}\}$	$\mathbf{class} \mathbf{var} f : \tau$
						$\mathbf{class} \mathbf{meth} m : \tau$
						σ_d
τ	$::=$	t	cd	$::=$	$\mathbf{class} \mathbf{var} f : \tau = e$	$\sigma_d ::=$
		$\tau \rightarrow \tau$			$\mathbf{class} \mathbf{meth} m : \tau = e$	$\mathbf{var} f : \tau$
						$\mathbf{type} t \{\overline{\tau_d}\}$
						$\mathbf{class} c \{\overline{\sigma_{cd}}, \overline{\sigma_d}\}$
τ_d	$::=$	$\mathbf{meth} m : \tau$				τ_d

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.

$$\begin{array}{c}
\frac{}{\Gamma, _ \vdash \mathbf{type} \ t \ \{\overline{\tau_d}\} :: \mathbf{type} \ t \ \{\overline{\tau_d}\}} \text{DT-type} \\
\\
\frac{\Gamma, \sigma_{this} \vdash \overline{d} :: \overline{\sigma_d} \quad \Gamma, \sigma_{this} \vdash \overline{cd} :: \overline{\sigma_{cd}} \quad \sigma_{this} = \{\overline{\sigma_{cd}}, \overline{\sigma_d}\}}{\Gamma, _ \vdash \mathbf{class} \ c \ \{\overline{cd}, \overline{d}\} :: \mathbf{class} \ c \ \{\overline{cd}, \overline{d}\}} \text{DT-class} \\
\\
\frac{\Gamma, \sigma_{this} \vdash e : \tau_2 \quad \tau_2 <: \tau_1}{\Gamma, \sigma_{this} \vdash \mathbf{class} \ \mathbf{var} \ f : \tau_1 = e_1 :: \mathbf{class} \ \mathbf{var} \ f : \tau_1} \text{DT-class-var} \\
\\
\frac{\Gamma, \sigma_{this} \vdash e : \tau_2 \quad \tau_2 <: \tau_1}{\Gamma, \sigma_{this} \vdash \mathbf{class} \ \mathbf{meth} \ m : \tau_1 = e_1 :: \mathbf{class} \ \mathbf{meth} \ m : \tau_1} \text{DT-class-meth} \\
\\
\frac{\Gamma, \sigma_{this} \vdash \overline{d_1} :: \overline{\sigma_{d_1}} \quad \{\overline{\sigma_{d_1}}\} \cup \sigma_{this} <: \{\overline{\tau_d}\}}{\Gamma, \sigma_{this} \vdash \mathbf{new} \ \{\overline{d_1}\} : \{\overline{\tau_d}\}} \text{T-new} \\
\\
\frac{\Gamma, \sigma_{this} \vdash e : \sigma_1 \quad \sigma_1 = \{\mathbf{class} \ \mathbf{var} \ f : \tau_1, \dots\}}{\Gamma, \sigma_{this} \vdash e.f : \tau_1} \text{T-class-field} \\
\\
\frac{\Gamma, \sigma_{this} \vdash e : \tau_1 \quad \tau_1 = \{\mathbf{class} \ \mathbf{meth} \ m : \tau_1 = e_1, \dots\}}{\Gamma, \sigma_{this} \vdash e.m : \tau_1} \text{T-class-meth}
\end{array}$$

Figure 10: Static Semantics Rules Core 3

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

Figure 11: Translation of a Class from FWC to FW

```

1 class Option
2   var quantity : int = 0
3   var price : int = 0
4   meth exercise : int =
5     this.quantity * this.price
6
7   class var totalQuantityIssued : int = 0
8   class meth issue : int -> int -> Option =
9     fn q : int =>
10      fn p : int =>
11        totalQuantityIssued =
12          totalQuantityIssued + q
13      new
14        var quantity : int = q
15        var price : int = p
16
17 var optn : Option = Option.issue(100, 50)
18 var ret : int = optn.exercise

```

Figure 12: An Option Class in Featherweight Wyvern with Classes

4.1 Example Program and Translation by Darya

```

1  type Option =
2    meth exercise : int
3
4  type OptionC =
5    meth issue : int -> int -> Option
6
7  var Option : OptionC =
8    new
9      var totalQuantityIssued : int = 0
10     meth issue : int -> int -> Option =
11       fn q : int =>
12         fn p : int =>
13           totalQuantityIssued =
14             totalQuantityIssued + q
15         new
16           var quantity : int = q
17           var price : int = p
18           meth exercise : int =
19             this.quantity * this.price
20
21  var optn : Option = Option.issue(100, 50)
22  var ret : int = optn.exercise

```

Figure 13: Option Class Translated to Featherweight Wyvern


```

1  type t
2    meth add : int -> t
3    meth get : int
4
5  class c
6    var f : int
7    meth add : int -> c =
8      fn x : int =>
9        this.f = this.f + x
10     this
11   meth get : int
12   meth equals : c -> bool =
13     fn other : c =>
14       this.f == other.get // cannot access other.f in type c
15                           // because this core doesn't support ADTs
16   class var nc : int = 1
17   class meth make : int -> c =
18     fn x:int =>
19       nc = nc + 1
20     new
21       var f:int = x
22       meth get : int = this.f
23
24   val o : c = c.make(4)
25   val o2 : t = o.add(2)
26   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.

```

1  type t = rec t2.
2    meth add : int -> t2
3    meth get : int
4
5  type c = rec c2.
6    meth add : int -> c2
7    meth get : int
8    meth equals : c2 -> bool
9
10 type c_internal = rec ci2. // not necessary, but a convenient abbreviation
11   var f : int
12   meth add : int -> c
13   meth get : int
14   meth equals : c -> bool
15
16 type c_class = rec cl2. // not necessary
17   meth make : int -> c
18
19 type c_class_internal = rec cli2. // not necessary
20   var nc : int
21   meth make : int -> c
22
23 val c : c_class
24   = new c_class_internal
25     var nc : int = 1
26     meth make : int -> c =
27       fn x:int =>
28         nc = nc + 1
29         new c_internal
30           var f : int = x
31           meth get : int = this.f
32           meth add : int -> c =
33             fn x : int =>
34               this.f = this.f + x
35               this
36           meth equals : c -> bool =
37             fn other : c =>
38               this.f == other.get // cannot access other.f in type c
39
40 val o : c = c.make(4)
41 val o2 : t = o.add(2)
42 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`

e	$::=$	x $\lambda x:\tau.e$ $e(e)$ new $\{\bar{d}\}$ $e.f$ $e.f = e$ $e.m$
τ	$::=$	$p\ t$ $\tau \rightarrow \tau$
p	$::=$	ϵ $x.$ $p\ f.$ <i>f must be constant</i>
u	$::=$	module $url : \tau\ \bar{i}\ \bar{d}$
i	$::=$	import url import $url : \tau$ as x
d	$::=$	public _{opt} var $f : \tau = e$ public _{opt} val $f : \tau = e$ public _{opt} def $m : \tau = e$ public _{opt} type $t = \{\bar{\tau}_d\}$
τ_d	$::=$	constant _{opt} def $m : \tau$ type $t = \{\tau\}$
σ	$::=$	τ $\{\bar{\sigma}_d\}$
σ_d	$::=$	var $f : \tau$ τ_d

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)