Lambda calculus
(Advanced Functional Programming)

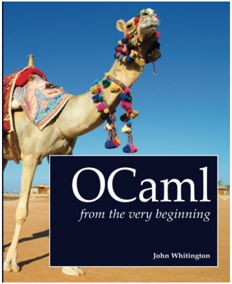
Jeremy Yallop

Computer Laboratory
University of Cambridge

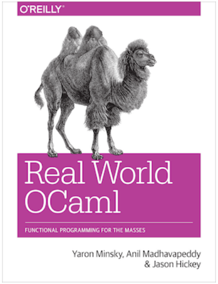
January 2015

Course outline

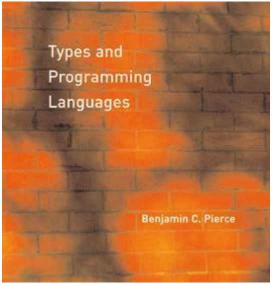
Books



OCaml from the very beginning
John Whitington
Coherent Press (2013)



Real World OCaml
Yaron Minsky,
Anil Madhavapeddy &
Jason Hickey
O'Reilly Media (2013)



Types and Programming Languages
Benjamin C. Pierce
MIT Press (2002)

Tooling

OPAM
OCaml package manager

Linux / OSX / VirtualBox

IOCaml

Fω

Fω interpreter

Philosophy and approach

- **practical**: with theory as necessary for understanding
- **real-world**: patterns and techniques from real applications
- **reusable**: general, widely applicable techniques
- **current**: mostly the topics of ongoing research

Philosophy and approach

- **practical**: with theory as necessary for understanding
- **real-world**: patterns and techniques from real applications
- **reusable**: general, widely applicable techniques
- **current**: mostly the topics of ongoing research
- **opinionated** (but you don't have to agree)

Mailing list

cl-acs-28@lists.cam.ac.uk

Announcements, questions and discussion. Feel free to post!

Have a question but feeling shy? Mail a lecturer instead and we'll anonymise and post your question:

jeremy.yallop@cl.cam.ac.uk leo.white@cl.cam.ac.uk

Exercises assessed and unassessed

Unassessed exercises:

Useful preparation for the assessed exercises, so we recommend that you work through them. Hand in for feedback, discuss freely on the mailing list.

Assessed exercises:

Mon 2 Feb	Mon 16 Feb	Mon 9 March
↓	↓	↓
Mon 9 Feb	Mon 2 March	Fri 24 April

Course structure

- **Technical background**
  Lambda calculus; type inference
- **Themes**
  Propositions as types; duality; parametricity and abstraction
- **(Fancy) types**
  Higher-rank and higher-kinded polymorphism; modules and functors; generalised algebraic types; rows
- **Applications**
  Monads and related concepts; domain-specific languages; datatype-generic programming; staged programming

Motivation & background

System Fω

Function composition in OCaml:

```
fun f g x -> f (g x)
```

Function composition in System Fω:

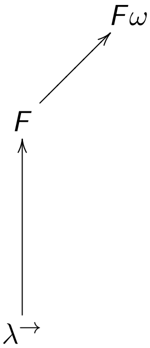
$$
\begin{array}{l}
\Lambda\alpha :: * . \\
\quad \Lambda\beta :: * . \\
\quad\quad \Lambda\gamma :: * . \\
\quad\quad\quad \lambda f : \alpha \to \beta . \\
\quad\quad\quad\quad \lambda g : \gamma \to \alpha . \\
\quad\quad\quad\quad\quad \lambda x :: \gamma . \; f \;\; (g \;\; x)
\end{array}
$$

What's the point of System Fω?

A framework for understanding language features and programming patterns:

- the elaboration language for type inference
- the proof system for reasoning with propositional logic
- the setting for dualities
- the background for parametricity properties
- the language underlying higher-order polymorphism in OCaml
- the elaboration language for modules
- the core calculus for GADTs

Roadmap



Inference rules

$$\frac{\text{premise 1} \qquad \text{premise 1} \qquad \ldots \qquad \text{premise N}}{\text{conclusion}}\ \text{rule name}$$

Inference rules

$$\frac{\text{premise 1} \qquad \text{premise 1} \qquad \ldots \qquad \text{premise N}}{\text{conclusion}}\ \text{rule name}$$

$$\frac{\begin{array}{c}\text{all } M \text{ are } P\\ \text{all } S \text{ are } M\end{array}}{\text{all } S \text{ are } P}\ \text{modus barbara}$$

Inference rules

$$\frac{\text{premise 1} \qquad \text{premise 1} \qquad \ldots \qquad \text{premise N}}{\text{conclusion}}\ \text{rule name}$$

$$\frac{\begin{array}{c}\text{all } M \text{ are } P \\ \text{all } S \text{ are } M\end{array}}{\text{all } S \text{ are } P}\ \text{modus barbara}$$

$$\frac{\begin{array}{c}\text{all programs are buggy} \\ \text{all functional programs are programs}\end{array}}{\text{all functional programs are buggy}}\ \text{modus barbara}$$

Typing rules

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B} \to\text{-elim}$$

Terms, types, kinds

Kinds: K, K_1, K_2, ...

K is a kind

Environments: Γ

Γ is an environment

Types: A, B, C, ...

$\Gamma \vdash A :: K$

Terms: L, M, N, ...

$\Gamma \vdash M : A$

$$\lambda^{\rightarrow}$$

(simply typed lambda calculus)

$\lambda^{\rightarrow}$ by example

In $\lambda^{\rightarrow}$:

$\lambda x : A . x$

$\lambda f : B \rightarrow C .$
$\quad \lambda g : A \rightarrow B .$
$\quad\quad \lambda x : A . f \; (g \; x)$

In OCaml:

```
fun x -> x

fun f g x -> f (g x)
```

Kinds in $\lambda^{\rightarrow}$

$$\frac{}{* \text{ is a kind}}\ *\text{-kind}$$

Kinding rules (type formation) in $\lambda^{\rightarrow}$

$$\frac{}{\Gamma \vdash \mathcal{B} :: *} \text{kind-}\mathcal{B}$$

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A \rightarrow B :: *} \text{kind-}\rightarrow$$

A kinding derivation

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B} \qquad \dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B}}{\Gamma \vdash \mathcal{B} \to \mathcal{B} :: *}\,\text{kind-}\to \qquad \dfrac{}{\Gamma \vdash \mathcal{B} :: *}\,\text{kind-}\mathcal{B}}{\Gamma \vdash (\mathcal{B} \to \mathcal{B}) \to \mathcal{B} :: *}\,\text{kind-}\to$$

Environment formation rules

$$\frac{}{\cdot \text{ is an environment}}\ \Gamma\text{-}\cdot$$

$$\frac{\Gamma \text{ is an environment} \qquad \Gamma \vdash A :: *}{\Gamma, x : A \text{ is an environment}}\ \Gamma\text{-}:$$

Typing rules (term formation) in $\lambda^{\rightarrow}$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \text{tvar}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \rightarrow B}\ \rightarrow\text{-intro} \qquad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ \rightarrow\text{-elim}$$

A typing derivation for the identity function

$$\frac{\cdot, x : A \vdash x : A}{\cdot \vdash \lambda x : A . x : A \to A} \to\text{-intro}$$

Products by example

In $\lambda^{\rightarrow}$ with products:

$\lambda p : (A \rightarrow B) \times A .$
$\quad$ **fst** p (**snd** p)

$\lambda x : A . \langle x , x \rangle$

$\lambda f : A \rightarrow C .$
$\quad \lambda g . B \rightarrow C .$
$\quad\quad \lambda p . A \times B .$
$\quad\quad\quad \langle$ f **fst** p , g **snd** p $\rangle$

$\lambda p . A \times B . \langle$ **snd** p , **fst** p $\rangle$

In OCaml:

```
fun (f, p) -> f p

fun x -> (x, x)

fun f g (x, y) -> (f x, g y)

fun (x, y) -> (y, x)
```

Kinding and typing rules for products

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A \times B :: *} \text{ kind-}\times$$

$$\frac{\begin{array}{c}\Gamma \vdash M : A \\ \Gamma \vdash N : B\end{array}}{\Gamma \vdash \langle M, N \rangle : A \times B} \times\text{-intro}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{fst}\ M : A} \times\text{-elim-1}$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \mathsf{snd}\ M : B} \times\text{-elim-2}$$

Sums by example

In $\lambda^{\to}$ with sums:

```
λf:A→C.
  λg:B→C.
    λs:A+B.
      case s of
        x.f x
      | y.g y
```

```
λs:A+B.
  case s of
    x.inr [B] x
  | y.inr [A] y
```

In OCaml:

```
fun f g s ->
  match s with
    Inl x -> f x
  | Inr y -> g y
```

```
function
  Inl x -> Inr x
| Inr y -> Inl y
```

Kinding and typing rules for sums

$$\frac{\Gamma \vdash A :: * \qquad \Gamma \vdash B :: *}{\Gamma \vdash A + B :: *}\ \text{kind-}+$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl}\ [B]\ M : A + B}\ +\text{-intro-1}$$

$$\frac{\Gamma \vdash N : B}{\Gamma \vdash \text{inr}\ [A]\ N : A + B}\ +\text{-intro-2}$$

$$\frac{\begin{array}{c}\Gamma \vdash L : A + B \\ \Gamma, x : A \vdash M : C \\ \Gamma, y : B \vdash N : C\end{array}}{\Gamma \vdash \text{case}\ L\ \text{of}\ x.M \mid y.N : C}\ +\text{-elim}$$

System F

(polymorphic lambda calculus)

System F by example

$$\Lambda\alpha :: * . \lambda \mathsf{x} : \alpha . \mathsf{x}$$

$$
\begin{array}{l}
\Lambda\alpha :: * . \\
\quad \Lambda\beta :: * . \\
\quad\quad \Lambda\gamma :: * . \\
\quad\quad\quad \lambda \mathsf{f} : \beta \to \gamma . \\
\quad\quad\quad\quad \lambda \mathsf{g} : \alpha \to \beta . \\
\quad\quad\quad\quad\quad \lambda \mathsf{x} : \alpha . \mathsf{f}\ (\mathsf{g}\ \mathsf{x})
\end{array}
$$

$$\Lambda\alpha :: * . \Lambda\beta :: * . \lambda \mathsf{p} : (\alpha \to \beta) \times \alpha . \mathbf{fst}\ \mathsf{p}\ (\mathbf{snd}\ \mathsf{p})$$

New kinding rules for System F

$$\frac{\Gamma, \alpha{::}K \vdash A :: *}{\Gamma \vdash \forall\alpha{::}K.A :: *}\ \text{kind-}\forall$$

$$\frac{\alpha{::}K \in \Gamma}{\Gamma \vdash \alpha :: K}\ \text{tyvar}$$

New environment rule for System F

$$\frac{\Gamma \text{ is an environment} \qquad K \text{ is a kind}}{\Gamma, \alpha{::}K \text{ is an environment}} \ \Gamma\text{-}{::}$$

New typing rules for System F

$$\frac{\Gamma, \alpha{::}K \vdash M : A}{\Gamma \vdash \Lambda\alpha{::}K.M : \forall\alpha{::}K.A} \ \forall\text{-intro}$$

$$\frac{\Gamma \vdash M : \forall\alpha{::}K.A \qquad \Gamma \vdash B :: K}{\Gamma \vdash M\ [B] : A[\alpha := B]} \ \forall\text{-elim}$$

Existential types

What's the point of existentials?

- $\forall$ and $\exists$ in logic are closely connected to polymorphism and existentials in type theory
- As in logic, $\forall$ and $\exists$ for types are closely related
- Module types can be viewed as a kind of existential type
- OCaml's variant types now support existential variables

Existential intuition

Existentials
correspond to
abstract types

Kinding rules for existentials

$$\frac{\Gamma, \alpha{::}K \vdash A :: *}{\Gamma \vdash \exists \alpha{::}K.A :: *}\ \text{kind-}\exists$$

Typing rules for existentials

$$\frac{\Gamma \vdash M : A[\alpha := B] \qquad \Gamma \vdash \exists\alpha{::}K.A :: *}{\Gamma \vdash \text{pack } B, M \text{ as } \exists\alpha{::}K.A : \exists\alpha{::}K.A}\ \exists\text{-intro}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \exists\alpha{::}K.A \\ \Gamma, \alpha{::}K, x : A \vdash M' : B\end{array}}{\Gamma \vdash \text{open } M \text{ as } \alpha, x \text{ in } M' : B}\ \exists\text{-elim}$$