MPRI 2.4, Functional programming and type systems
Metatheory of System F

Didier Rémy

Plan of the course

Simply typed lambda-calculus

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^ω!

Logical relations

Side effects, References, Value restriction

Type reconstruction

Overloading

Fomega: higher-kinds and higher-order types

Contents

Polymorphism in System F

Simply-typed λ-calculus

- no polymorphism
- many functions must be duplicated at different types

Via ML style (let-binding) polymorphism

- Considerable improvement by avoiding most of code duplication.
- ML has also local let-polymorphism (less critical).
- Still, ML is lacking existential types—compensated by modules and sometimes lacking higher-rank polymorphism

System F brings much more expressiveness

- Existential types—allows for type abstraction
- First-class universal types
- Allows for encoding of data structures and more programming patterns

Still, limited...

Limits of System F $\lambda f x y. (f\ x, f\ y)$

Map on pairs, say pair_map, has the following types:

Limits of System F $\lambda f x y. (f\ x, f\ y)$

Map on pairs, say pair_map, has the following types:

$$\forall \alpha_1. \forall \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
$$\forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Limits of System F $\lambda f x y. (f\ x, f\ y)$

Map on pairs, say pair_map, has the following types:

$$\forall \alpha_1. \forall \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
$$\forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic.

Limits of System F $\lambda f x y. (f\ x, f\ y)$

Map on pairs, say pair_map, has the following incompatible types:

$$\forall \alpha_1. \forall \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
$$\forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic.

Limits of System F $\lambda f x y.\,(f\ x, f\ y)$

Map on pairs, say pair_map, has the following incompatible types:

$$\forall \alpha_1. \forall \alpha_2. (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
$$\forall \alpha_1. \forall \alpha_2. (\forall \alpha. \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic.

It is missing the ability to describe the types of functions

- that are polymorphic in one parameter
- but whose domain and codomain are otherwise arbitrary

i.e. of the form $\forall \alpha. \tau[\alpha] \to \sigma[\alpha]$ for arbitrary one-hole types τ and σ.

Limits of System F $\lambda fxy.(f\ x, f\ y)$

Map on pairs, say pair_map, has the following incompatible types:

$$\forall\alpha_1.\forall\alpha_2.(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$
$$\forall\alpha_1.\forall\alpha_2.(\forall\alpha.\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

The first one requires x and y to admit a common type, while the second one requires f to be polymorphic.

It is missing the ability to describe the types of functions

- that are polymorphic in one parameter
- but whose domain and codomain are otherwise arbitrary

i.e. of the form $\forall\alpha.\tau[\alpha] \to \sigma[\alpha]$ for arbitrary one-hole types τ and σ.

We just need to abstract over such contexts, *i.e.*, over *type functions*:

$$\forall\varphi.\forall\psi.\forall\alpha_1.\forall\alpha_2.(\forall\alpha.\varphi\ \alpha \to \psi\ \alpha) \to \varphi\ \alpha_1 \to \varphi\ \alpha_2 \to \psi\ \alpha_1 \times \psi\ \alpha_2$$

From System F to System F^ω — Kinds

We introduce kinds κ for types (with a single kind $*$ to stay in System F)

Well-formedness of types becomes $\Gamma \vdash \tau : *$:

$$\frac{\vdash \Gamma \quad \alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa . \tau : *}$$

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau}$$

We add and check kinds on type abstractions and type applications:

$$\textsc{Tabs}\ \frac{\Gamma, \alpha : \kappa \vdash M : \tau}{\Gamma \vdash \Lambda \alpha :: \kappa . M : \forall \alpha :: \kappa . \tau} \qquad \textsc{Tapp}\ \frac{\Gamma \vdash M : \forall \alpha :: \kappa . \tau \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

So far, this is an equivalent formalization of System F

From System F to System F^ω — Type functions

Redefine kinds as $\kappa ::= * \mid \kappa \Rightarrow \kappa$

New types $\tau ::= \ldots \mid \lambda\alpha :: \kappa . \tau \mid \tau\ \tau$

WfTypeApp

$$\frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

WfTypeAbs

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha :: \kappa_1 . \tau : \kappa_1 \Rightarrow \kappa_2}$$

Typing of expressions is up to type equivalence:

TConv

$$\frac{\Gamma \vdash M : \tau \qquad \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

From System F to System F^ω — Type functions

Redefine kinds as $\kappa ::= * \mid \kappa \Rightarrow \kappa$

New types $\tau ::= \ldots \mid \lambda\alpha :: \kappa.\tau \mid \tau\ \tau$

WfTypeApp
$$\frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

WfTypeAbs
$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha :: \kappa_1.\tau : \kappa_1 \Rightarrow \kappa_2}$$

Typing of expressions is up to type equivalence:

TConv
$$\frac{\Gamma \vdash M : \tau \qquad \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

Remark

$$\Gamma \vdash M : \tau \implies \Gamma \vdash \tau : *$$

F^ω, static semantics (altogether on one slide)

Syntax

$$\begin{array}{lcl} \kappa & ::= & * \mid \kappa \Rightarrow \kappa \\ \tau & ::= & \alpha \mid \tau \to \tau \mid \forall \alpha :: \kappa . \tau \mid \lambda \alpha :: \kappa . \tau \mid \tau\ \tau \\ M & ::= & x \mid \lambda x : \tau . M \mid M\ M \mid \Lambda \alpha :: \kappa . M \mid M\ \tau \end{array}$$

Kinding rules

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha :: \kappa . \tau : *} \qquad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda \alpha :: \kappa_1 . \tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

Typing rules

Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Abs
$$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . M : \tau_1 \to \tau_2}$$

App
$$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

TAbs
$$\frac{\Gamma, \alpha : \kappa \vdash M : \tau}{\Gamma \vdash \Lambda \alpha :: \kappa . M : \forall \alpha :: \kappa . \tau}$$

TApp
$$\frac{\Gamma \vdash M : \forall \alpha :: \kappa . \tau \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau}$$

TEquiv
$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

F^ω, static semantics (altogether on one slide)

Syntax

With implicit kinds

$$\kappa ::= * \mid \kappa \Rightarrow \kappa$$
$$\tau ::= \alpha \mid \tau \to \tau \mid \forall \alpha.\tau \mid \lambda\alpha.\tau \mid \tau\ \tau$$
$$M ::= x \mid \lambda x{:}\tau.\,M \mid M\ M \mid \Lambda\alpha.\,M \mid M\ \tau$$

Kinding rules

$$\vdash \varnothing \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \quad x \notin \mathrm{dom}(\Gamma)}{\vdash \Gamma, x : \tau} \qquad \frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau_1 : * \quad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *}$$

$$\frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha.\tau : *} \qquad \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha.\tau : \kappa_1 \Rightarrow \kappa_2} \qquad \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \quad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_1}$$

Typing rules

$$\textsc{Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{Abs}\ \frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\,M : \tau_1 \to \tau_2} \qquad \textsc{App}\ \frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\ M_2 : \tau_2}$$

$$\textsc{TAbs}\ \frac{\Gamma, \alpha : \kappa \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.\,M : \forall \alpha.\tau} \qquad \textsc{TApp}\ \frac{\Gamma \vdash M : \forall \alpha.\tau \quad \Gamma \vdash \tau' : \kappa}{\Gamma \vdash M\ \tau' : [\alpha \mapsto \tau']\tau} \qquad \textsc{TEquiv}\ \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash \tau \equiv_\beta \tau'}{\Gamma \vdash M : \tau'}$$

F^ω, dynamic semantics

The semantics is unchanged (modulo kind annotations in terms)

$$V ::= \lambda x:\tau.\, M \mid \Lambda\alpha::\kappa.\, V$$
$$E ::= [\,]\, M \mid V\, [\,] \mid [\,]\, \tau \mid \Lambda\alpha::\kappa.\, [\,]$$

$$(\lambda x:\tau.\, M)\, V \longrightarrow [x \mapsto V]M$$
$$(\Lambda\alpha::\kappa.\, V)\, \tau \longrightarrow [\alpha \mapsto \tau]V$$

Context
$$\frac{M \longrightarrow M'}{E[M] \longrightarrow E[M']}$$

No type reduction

- We need not reduce types inside terms.
- β reduction on types is needed for type conversion (*i.e.* for typing) but such reduction need not be performed during term reduction.

Kinds are erasable

- Kinds are preserved by type and term reduction.
- Kinds may be ignored during reduction—or erased prior to reduction.

Properties

Main properties are preserved. Proofs are similar to those for System F.

Type soundness

- Subject reduction
- Progress

Termination of reduction

(In the absence of construct for recursion.)

Typechecking is decidable

- This requires reduction at the level of types to check type equality
- Can be done by putting types in normal forms using full reduction (on types only), or just head normal forms.

Type reduction

Used for typechecking to check type equivalence $\equiv$

Full reduction of the simply typed λ-calculus

$$(\lambda\alpha.\,\tau)\;\sigma \longrightarrow [\alpha \mapsto \tau]\sigma$$

applicable in *any type context*.

Type reduction preserve types: this is subject reduction for simply-typed λ-calculus (when terms are now used as types), but for *full reduction*

(we have only proved it for CBV).

It is a key that reduction terminates.

(which again, we have only proved for CBV.)

Contents

Expressiveness

More polymorphism

- pair_map

Abstraction over type operators

- monads
- encoding of existentials

Other encodings

- non regular datatypes
- equality
- *modules*

Pair map in F^ω $\lambda f x y.\, (f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

$$\Lambda\varphi :: * \Rightarrow *.\, \Lambda\psi :: * \Rightarrow *.\, \Lambda\alpha_1 :: *.\, \Lambda\alpha_2 :: *.$$
$$\lambda(f : \forall\alpha :: *.\, \varphi\alpha \to \psi\alpha).\, \lambda x : \varphi\alpha_1.\, \lambda y : \varphi\alpha_2.\, (f\ \alpha_1\ x, f\ \alpha_2\ y)$$

call it pair_map of type:

$$\forall\varphi :: * \Rightarrow *. \forall\psi :: * \Rightarrow *. \forall\alpha_1 :: *. \forall\alpha_2 :: *.$$
$$(\forall\alpha :: *.\, \varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

We may recover, in particular, the two types it has in System F:

$$\Lambda\alpha_1 :: *.\, \Lambda\alpha_2 :: *.\, \mathsf{pair\_map}\ (\lambda\alpha :: *.\, \alpha_1)\ (\lambda\alpha :: *.\, \alpha_2)\ \alpha_1\ \alpha_2$$
$$: \forall\alpha_1 :: *.\, \forall\alpha_2 :: *.\, (\ \forall\gamma.\ \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\mathsf{pair\_map}\ (\lambda\alpha :: *.\alpha)\ (\lambda\alpha :: *.\alpha)$$
$$: \forall\alpha_1 :: *.\, \forall\alpha_2.\, (\forall\alpha :: *.\, \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Pair map in F^ω (with implicit kinds) $\lambda fxy.(f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

$$\Lambda\varphi.\,\Lambda\psi.\,\Lambda\alpha_1.\,\Lambda\alpha_2.\quad \lambda(f : \forall\alpha.\,\varphi\alpha \to \psi\alpha).\,\lambda x : \varphi\alpha_1.\,\lambda y : \varphi\alpha_2.\,(f\ \alpha_1\ x, f\ \alpha_2\ y)$$

call it pair_map of type:

$$\forall\varphi.\forall\psi.\forall\alpha_1.\forall\alpha_2.\quad (\forall\alpha.\,\varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

We may recover, in particular, the two types it has in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\text{pair\_map}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\ \forall\gamma.\ \alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{pair\_map}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Pair map in F^ω (with implicit kinds) $\lambda f x y.\,(f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

$$\Lambda\varphi.\,\Lambda\psi.\,\Lambda\alpha_1.\,\Lambda\alpha_2.\;\lambda(f : \forall\alpha.\,\varphi\alpha \to \psi\alpha).\,\lambda x : \varphi\alpha_1.\,\lambda y : \varphi\alpha_2.\,(f\ \alpha_1\ x, f\ \alpha_2\ y)$$

call it pair_map of type:

$$\forall\varphi.\forall\psi.\forall\alpha_1.\forall\alpha_2.\;(\forall\alpha.\,\varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

We may recover, in particular, the two types it has in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\lambda f : \alpha_1 \to \alpha_2.\,\text{pair\_map}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2\ (\Lambda\gamma.\,f)$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{pair\_map}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)$$
$$: \forall\alpha_1.\,\forall\alpha_2.\,(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Pair map in F^ω (with implicit kinds) $\lambda f x y. (f\ x, f\ y)$

Abstract over (one parameter) type *functions* (*e.g.* of kind $\star \to \star$)

$$\Lambda\varphi.\, \Lambda\psi.\, \Lambda\alpha_1.\, \Lambda\alpha_2.\; \lambda(f : \forall\alpha.\, \varphi\alpha \to \psi\alpha).\, \lambda x : \varphi\alpha_1.\, \lambda y : \varphi\alpha_2.\, (f\ \alpha_1\ x, f\ \alpha_2\ y)$$

call it pair_map of type:

$$\forall\varphi.\forall\psi.\forall\alpha_1.\forall\alpha_2.\; (\forall\alpha.\, \varphi\alpha \to \psi\alpha) \to \varphi\alpha_1 \to \varphi\alpha_2 \to \psi\alpha_1 \times \psi\alpha_2$$

We may recover, in particular, the two types it has in System F:

$$\Lambda\alpha_1.\, \Lambda\alpha_2.\, \lambda f : \alpha_1 \to \alpha_2.\, \text{pair\_map}\ (\lambda\alpha.\, \alpha_1)\ (\lambda\alpha.\, \alpha_2)\ \alpha_1\ \alpha_2\ (\Lambda\gamma.\, f)$$
$$: \forall\alpha_1.\, \forall\alpha_2.\, (\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{pair\_map}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)$$
$$: \forall\alpha_1.\, \forall\alpha_2.\, (\forall\alpha.\, \alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Still, the type of pair_map is not principal: φ and ψ could depend on two variables, *i.e.* be of kind $* \Rightarrow * \Rightarrow *$, or many other kinds...

Abstracting over type operators

Type of monads Given a type operator φ, a monad is given by a pair of two functions of the following type (satisfying certain laws).

$$\begin{aligned}\mathcal{M} \;&\triangleq\; \lambda\,(\varphi :: * \Rightarrow *)\,.\\ &\quad\{\; \mathit{ret} : \forall\,(\alpha :: *)\,.\,\alpha \to \varphi\alpha;\\ &\quad\;\; \mathit{bind} : \forall\,(\alpha :: *)\,.\,\forall\,(\beta :: *)\,.\,\varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \;\}\\ \;&:\; (* \Rightarrow *) \Rightarrow *\end{aligned}$$

(Notice that $\mathcal{M}$ is itself of higher kind)

Abstracting over type operators

Type of monads Given a type operator φ, a monad is given by a pair of two functions of the following type (satisfying certain laws).

$$
\begin{array}{rl}
\mathcal{M} & \triangleq \lambda (\varphi :: * \Rightarrow *) .\\
& \quad \{ \; \mathit{ret} : \forall (\alpha :: *) . \alpha \to \varphi\alpha; \\
& \quad\;\; \mathit{bind} : \forall (\alpha :: *) . \forall (\beta :: *) . \varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \; \} \\
& : \; (* \Rightarrow *) \Rightarrow *
\end{array}
$$

(Notice that $\mathcal{M}$ is itself of higher kind)

A generic map function: can then be defined:

$$
\begin{array}{l}
\mathit{fmap} \\
\quad \triangleq \quad \Lambda (\varphi :: * \Rightarrow *) . \lambda m : \mathcal{M}\,\varphi. \\
\qquad\qquad \Lambda (\alpha :: *) . \Lambda (\beta :: *) . \lambda f : (\alpha \to \beta). \lambda x : \varphi\alpha. \\
\qquad\qquad\qquad m.\mathit{bind}\; \alpha\; \beta\; x\; (\lambda x : \alpha.\, m.\mathit{ret}\; \beta\; (f\; x)) \\
\quad : \quad \forall (\varphi :: * \Rightarrow *) . \mathcal{M}\,\varphi \to \forall (\alpha :: *) . \forall (\beta :: *) . (\alpha \to \beta) \to \varphi\alpha \to \varphi\beta
\end{array}
$$

Abstracting over type operators

Type of monads Given a type operator φ, a monad is given by a pair of two functions of the following type (satisfying certain laws).

$$\begin{aligned}
\mathcal{M} &\triangleq \lambda\varphi.\\
&\quad \{\ \mathit{ret} : \forall\alpha.\,\alpha \to \varphi\alpha;\\
&\quad\ \ \mathit{bind} : \forall\alpha.\,\forall\beta.\,\varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta\ \}\\
&: (\ast \Rightarrow \ast) \Rightarrow \ast
\end{aligned}$$

(Notice that $\mathcal{M}$ is itself of higher kind)

A generic map function: can then be defined:

$$\begin{aligned}
&\mathit{fmap}\\
&\quad\triangleq\ \Lambda\varphi.\,\lambda m : \mathcal{M}\,\varphi.\\
&\qquad\quad \Lambda\alpha.\,\Lambda\beta.\,\lambda f : (\alpha \to \beta).\,\lambda x : \varphi\alpha.\\
&\qquad\qquad m.\mathit{bind}\ \alpha\ \beta\ x\ (\lambda x : \alpha.\, m.\mathit{ret}\ \beta\ (f\ x))\\
&\quad :\ \forall\varphi.\,\mathcal{M}\,\varphi \to \forall\alpha.\,\forall\beta.\,(\alpha \to \beta) \to \varphi\alpha \to \varphi\beta
\end{aligned}$$

Abstracting over type operators

Type of monads Given a type operator φ, a monad is given by a pair of two functions of the following type (satisfying certain laws).

$$\begin{aligned}\mathcal{M} \;&\triangleq\; \lambda\varphi.\\ &\qquad \{\, \textit{ret} : \forall\alpha.\, \alpha \to \varphi\alpha;\\ &\qquad\;\; \textit{bind} : \forall\alpha.\, \forall\beta.\, \varphi\alpha \to (\alpha \to \varphi\beta) \to \varphi\beta \,\}\\ &: \;(* \Rightarrow *) \Rightarrow *\end{aligned}$$

(Notice that $\mathcal{M}$ is itself of higher kind)

A generic map function: can then be defined:

$$\begin{aligned}&\textit{fmap}\\ &\quad\triangleq\quad \lambda m.\\ &\qquad\qquad \lambda f.\, \lambda x.\\ &\qquad\qquad\qquad m.\textit{bind}\; x\; (\lambda x.\, m.\textit{ret}\; (f\; x))\\ &\quad:\quad \forall\varphi.\, \mathcal{M}\,\varphi \to \forall\alpha.\, \forall\beta.\, (\alpha \to \beta) \to \varphi\alpha \to \varphi\beta\end{aligned}$$

Abstracting over type operators

Available in Haskell —without β-reduction

- $\varphi\alpha$ is treated as a type $\text{app}(\varphi, \alpha)$ where $\text{app} : (\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2$
- No β-reduction at the level of types: $\varphi\alpha = \psi\beta \iff \varphi = \psi \wedge \alpha = \beta$
- Compatible with type inference (first-order unification)
- Since there is no type β-reduction, this is not F^{ω}.

Encodable in OCaml with modules

- See [Yallop and White, 2014] (and also [Kiselyov])
- As in Haskell, the encoding does not handle type β-reduction
- As a counterpart, this allows for type inference at higher kinds (as in Haskell).

Encoding of existentials

Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \quad = \quad ?$$

Encoding of existentials — Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Encoding of existentials — Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

Encoding of existentials — Limits of System F

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x\!:\![\![\tau]\!].\,\Lambda\beta.\,\lambda k\!:\!\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

In System F^ω, we may defined

$$[\![\mathit{pack}\;\;]\!] \;=\; ?$$

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau\to\beta)\to\beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!]\to\beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

In System F^ω, we may defined

$$[\![\mathit{pack}\;\;]\!] \;=\; \Lambda\,(\varphi :: * \Rightarrow *)\,.\,?$$

Encoding of existentials in System F^ω

We saw

$$[\![\exists\, \alpha.\, \tau]\!] \;=\; \forall \beta.\, (\forall \alpha.\, \tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\, \lambda x\!:\![\![\tau]\!].\, \Lambda\beta.\, \lambda k\!:\!\forall\alpha.\, ([\![\tau]\!] \to \beta).\, k\ \alpha\ x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

In System F^ω, we may defined

$$[\![\mathit{pack}\ ]\!] \;=\; \Lambda\, (\varphi :: \ast \Rightarrow \ast)\, .\, \Lambda\, (\alpha :: \ast)\, .\; \lambda x : \varphi\ \alpha.\, \Lambda\, (\beta :: \ast)\, .\, \lambda k : \forall\, (\alpha :: \ast)\, .\, (\varphi\ \alpha \to \beta).\, k\ \alpha\ x$$

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

In System F^ω, we may defined

$$[\![\mathit{pack}\;\;]\!] \;=\; \Lambda\varphi.\,\Lambda\alpha.\;\lambda x : \varphi\;\alpha.\,\Lambda\beta.\,\lambda k : \forall\alpha.\,(\varphi\;\alpha \to \beta).\,k\;\alpha\;x \qquad \text{(omitting kinds)}$$

Encoding of existentials in System F^ω

We saw

$$[\![\exists\,\alpha.\,\tau]\!] \;=\; \forall\beta.\,(\forall\alpha.\,\tau \to \beta) \to \beta$$

Hence,

$$[\![\mathit{pack}_{\exists\alpha.\tau}]\!] \;\triangleq\; \Lambda\alpha.\,\lambda x{:}[\![\tau]\!].\,\Lambda\beta.\,\lambda k{:}\forall\alpha.\,([\![\tau]\!] \to \beta).\,k\;\alpha\;x$$

This requires a different code for each type τ

To have a unique code, we just abstract over $\lambda\alpha.\,\tau$, *i.e.* φ:

In System F^ω, we may defined

$$[\![\mathit{pack}_{\kappa}]\!] \;=\; \Lambda\,(\varphi :: \kappa \Rightarrow \ast)\,.\,\Lambda\,(\alpha :: \kappa)\,.\,\lambda x : \varphi\;\alpha.\,\Lambda\,(\beta :: \ast)\,.\,\lambda k : \forall\,(\alpha :: \kappa)\,.\,(\varphi\;\alpha \to \beta).\,k\;\alpha\;x$$

Allows existentials at higher kinds!

Exploiting kinds

Once we have type functions, the language of types could be reduced to λ-calculus with constants (plus arrow types kept as primitive):

$$\tau = \alpha \mid \lambda\alpha{:}\kappa.\,\tau \mid \tau\ \tau \mid \tau \to \tau \mid g$$

where type constants $g \in \mathcal{G}$ are given with their kind and syntactic sugar:

$$\begin{array}{lll} \times & :: & \ast \Rightarrow \ast \Rightarrow \ast \\ + & :: & \ast \Rightarrow \ast \Rightarrow \kappa \\ \forall_\kappa & :: & (\kappa \Rightarrow \ast) \Rightarrow \ast \\ \exists_\kappa & :: & (\kappa \Rightarrow \ast) \Rightarrow \ast \end{array} \qquad \begin{array}{lll} (\tau \times \tau) & \triangleq & (\times)\ \tau_1\ \tau_2 \\ (\tau + \tau) & \triangleq & (+)\ \tau_1\ \tau_2 \\ \forall\varphi : \kappa.\,\tau & \triangleq & \forall_\kappa(\lambda\varphi{:}\kappa \Rightarrow \ast.\,\tau) \\ \exists\varphi : \kappa.\,\tau & \triangleq & \exists_\kappa(\lambda\varphi{:}\kappa \Rightarrow \ast.\,\tau) \end{array}$$

In fact F^ω could be extended with kind abdstraction:

$$\begin{array}{lll} \hat\forall & :: & \forall\kappa.(\kappa \Rightarrow \ast) \Rightarrow \ast \\ \hat\exists & :: & \forall\kappa.(\kappa \Rightarrow \ast) \Rightarrow \ast \end{array} \qquad \begin{array}{lll} \forall\varphi : \kappa.\,\tau & \triangleq & \hat\forall\ \kappa\ (\lambda\varphi : \kappa \Rightarrow \ast.\,\tau) \\ \exists\varphi : \kappa.\,\tau & \triangleq & \hat\exists\ \kappa\ (\lambda\varphi : \kappa \Rightarrow \ast.\,\tau) \end{array}$$

When kinds are inferred:

$$\begin{array}{lll} \forall\varphi.\,\tau & \triangleq & \hat\forall\ (\lambda\varphi.\,\tau) \\ \exists\varphi.\,\tau & \triangleq & \hat\exists\ (\lambda\varphi.\,\tau) \end{array}$$

Church encoding of regular ADT — List

$$\begin{array}{l} \texttt{type} \quad \mathit{List}\ \alpha = \\ \quad \mid \mathit{Nil} \quad : \forall\alpha.\ \mathit{List}\ \alpha \\ \quad \mid \mathit{Cons} : \forall\alpha.\ \alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha \end{array}$$

Church encoding (CPS style) in System F

$$\mathit{List} \triangleq \lambda\alpha.\ \forall\beta.\ \beta \to (\alpha \to \beta \to \beta) \to \beta$$

$$\mathit{Nil} \triangleq \lambda n.\ \lambda c.\ n$$

$$\mathit{Cons} \triangleq \lambda x.\ \lambda\ell.\ \lambda n.\ \lambda c.\ c\ x\ (\ell\ \beta\ n\ c)$$

$$\mathit{fold} \triangleq \lambda n.\ \lambda c.\ \lambda\ell.\ \ell\ \beta\ n\ c$$

Church encoding of regular ADT — List

$$
\begin{array}{l}
\texttt{type} \quad \mathit{List}\,\alpha = \\
\quad \mid \mathit{Nil} \;\; : \forall\alpha.\, \mathit{List}\,\alpha \\
\quad \mid \mathit{Cons} : \forall\alpha.\, \alpha \to \mathit{List}\,\alpha \to \mathit{List}\,\alpha
\end{array}
$$

Church encoding (CPS style) in System F

$$\mathit{List} \triangleq \lambda\alpha.\,\forall\beta.\,\beta \to (\alpha \to \beta \to \beta) \to \beta$$

$$\mathit{Nil} \triangleq \lambda n.\,\lambda c.\,n \;:\; \forall\alpha.\,\mathit{List}\,\alpha$$

$$\mathit{Cons} \triangleq \lambda x.\,\lambda\ell.\,\lambda n.\,\lambda c.\,c\;x\;(\ell\;\beta\;n\;c) \;:\; \forall\alpha.\,\alpha \to \mathit{List}\,\alpha \to \mathit{List}\,\alpha$$

$$\mathit{fold} \triangleq \lambda n.\,\lambda c.\,\lambda\ell.\,\ell\;\beta\;n\;c$$

Church encoding of regular ADT — List

```
type   List α =
     | Nil  : ∀α. List α
     | Cons : ∀α. α → List α → List α
```

Church encoding (CPS style) in System F

$$\mathit{List} \triangleq \lambda\alpha.\,\forall\beta.\,\beta \to (\alpha \to \beta \to \beta) \to \beta$$

$$\mathit{Nil} \triangleq \Lambda\alpha.\,\Lambda\beta.\,\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,n$$

$$\mathit{Cons} \triangleq \Lambda\alpha.\,\lambda x : \alpha.\,\lambda \ell : \mathit{List}\,\alpha.\; \Lambda\beta.\,\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,c\;x\;(\ell\;\beta\;n\;c)$$

$$\mathit{fold} \triangleq \Lambda\alpha.\,\Lambda\beta.\,\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,\lambda \ell : \mathit{List}\,\alpha.\,\ell\;\beta\;n\;c$$

Church encoding of regular ADT — List

$$
\begin{array}{ll}
\texttt{type} & \mathit{List}\,\alpha = \\
\mid \mathit{Nil} & : \forall \alpha.\, \mathit{List}\,\alpha \\
\mid \mathit{Cons} & : \forall \alpha.\, \alpha \to \mathit{List}\,\alpha \to \mathit{List}\,\alpha
\end{array}
$$

Church encoding (CPS style) enhanced in F^ω ?

$$\mathit{List} \triangleq \lambda \alpha.\, \forall \varphi.\, \varphi\alpha \to (\alpha \to \varphi\alpha \to \varphi\alpha) \to \varphi\alpha$$

$$\mathit{Nil} \triangleq \Lambda \alpha.\, \Lambda \varphi.\, \lambda n : \varphi\alpha.\, \lambda c : (\alpha \to \varphi\alpha \to \varphi\alpha).\, n$$

$$\mathit{Cons} \triangleq \Lambda \alpha.\, \lambda x : \alpha.\, \lambda \ell : \mathit{List}\,\alpha.\, \Lambda \varphi.\, \lambda n : \varphi\alpha.\, \lambda c : (\alpha \to \varphi\alpha \to \varphi\alpha).\, c\, x\, (\ell\, \varphi\, n\, c)$$

$$\mathit{fold} \triangleq \Lambda \alpha.\, \Lambda \varphi.\, \lambda n : \varphi\alpha.\, \lambda c : (\alpha \to \varphi\alpha \to \varphi\alpha).\, \lambda \ell : \mathit{List}\,\alpha.\, \ell\, \varphi\, n\, c$$

Actually not enhanced !

Be aware of useless over-generalization!

For regular ADTs, all uses of φ are $\varphi\alpha$.

Hence, $\forall \alpha.\, \forall \varphi.\, \tau[\varphi\alpha]$ is not more general than $\forall \alpha.\, \forall \beta.\, \tau[\beta]$

Church encoding of *non*-regular ADTs — Okasaki's Seq

$$
\begin{array}{ll}
\texttt{type} & \mathit{Seq}\;\alpha = \\
& \mid \mathit{Nil} \;: \forall\alpha.\, \mathit{Seq}\;\alpha \\
& \mid \mathit{Zero} : \forall\alpha.\, \mathit{Seq}\;(\alpha\times\alpha) \to \mathit{Seq}\;\alpha \\
& \mid \mathit{One} \;: \forall\alpha.\, \alpha \to \mathit{Seq}\;(\alpha\times\alpha) \to \mathit{Seq}\;\alpha
\end{array}
$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\, \forall\varphi.\, (\forall\alpha.\, \varphi\,\alpha) \to (\forall\alpha.\, \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to (\forall\alpha.\, \alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to \varphi\alpha$$

$$\mathit{Nil} \triangleq \lambda n.\, \lambda z.\, \lambda s.\, n$$

$$\mathit{Zero} \triangleq \lambda \ell.\, \lambda n.\, \lambda z.\, \lambda s.\, z\;(\ell\; n\; z\; s)$$

$$\mathit{One} \triangleq \lambda x.\, \lambda \ell.\, \lambda n.\, \lambda z.\, \lambda s.\, s\; x\;(\ell\; n\; z\; s)$$

$$\mathit{fold} \triangleq \lambda n.\, \lambda z.\, \lambda s.\, \lambda \ell.\, \ell\; n\; z\; s$$

Church encoding of *non*-regular ADTs — Okasaki's Seq

$$
\begin{aligned}
&\texttt{type}\quad \mathit{Seq}\ \alpha = \\
&\quad \mid \mathit{Nil} : \forall\alpha.\ \mathit{Seq}\ \alpha \\
&\quad \mid \mathit{Zero} : \forall\alpha.\ \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha \\
&\quad \mid \mathit{One} : \forall\alpha.\ \alpha \to \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha
\end{aligned}
$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\ \forall\varphi.\ (\forall\alpha.\ \varphi\,\alpha) \to (\forall\alpha.\ \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to (\forall\alpha.\ \alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to \varphi\alpha$$

$$\mathit{Nil} \triangleq \lambda n.\lambda z.\lambda s.\, n \quad : \quad \forall\alpha.\ \ \mathit{Seq}\ \alpha$$

$$\mathit{Zero} \triangleq \lambda\ell.\lambda n.\lambda z.\lambda s.\, z\ (\ell\ n\ z\ s) \quad : \quad \forall\alpha.\ \ \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha$$

$$\mathit{One} \triangleq \lambda x.\lambda\ell.\lambda n.\lambda z.\lambda s.\, s\ x\ (\ell\ n\ z\ s) \quad : \quad \forall\alpha.\ \ \alpha \to \mathit{Seq}\ (\alpha\times\alpha) \to \mathit{Seq}\ \alpha$$

$$\mathit{fold} \triangleq \lambda n.\lambda z.\lambda s.\lambda\ell.\, \ell\ n\ z\ s$$

Church encoding of *non*-regular ADTs — Okasaki's Seq

$$
\begin{array}{l}
\texttt{type}\quad \mathit{Seq}\,\alpha = \\
\quad \mid \mathit{Nil} : \forall\alpha.\,\mathit{Seq}\,\alpha \\
\quad \mid \mathit{Zero} : \forall\alpha.\,\mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha \\
\quad \mid \mathit{One} : \forall\alpha.\,\alpha \to \mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha
\end{array}
$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\,\forall\varphi.\,(\forall\alpha.\,\varphi\,\alpha) \to (\forall\alpha.\,\varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to (\forall\alpha.\,\alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to \varphi\alpha$$

$$\mathit{Nil} \triangleq \Lambda\alpha.\,\Lambda\varphi.\,\lambda n : \forall\alpha.\,\varphi\,\alpha.\,\lambda z : \forall\alpha.\,\varphi(\alpha\times\alpha) \to \varphi\,\alpha.\,\lambda s : \forall\alpha.\,\alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha.\,n$$

$$\mathit{Zero} \triangleq \Lambda\alpha.\,\lambda\ell : \mathit{Seq}\,(\alpha\times\alpha).\ \cdots$$

$$\mathit{One} \triangleq \Lambda\alpha.\,\lambda x : \alpha.\,\lambda\ell : \mathit{Seq}\,(\alpha\times\alpha).\ \Lambda\varphi.\,\lambda n : \forall\alpha.\,\varphi\,\alpha.\,\lambda z : \forall\alpha.\,\varphi(\alpha\times\alpha) \to \varphi\,\alpha.\,\lambda s : \forall\alpha.\,\alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha.\ s\ x\ (\ell\ \varphi\ n\ z\ s)$$

$$\mathit{fold} \triangleq \Lambda\alpha.\,\Lambda\varphi.\,\lambda n : \forall\alpha.\,\varphi\,\alpha.\,\lambda z : \forall\alpha.\,\varphi(\alpha\times\alpha) \to \varphi\,\alpha.\,\lambda s : \forall\alpha.\,\alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha.\ \lambda\ell : \mathit{Seq}\,\alpha.\,\ell\ \varphi\ n\ z\ s$$

Church encoding of *non*-regular ADTs — Okasaki's Seq

$$\begin{array}{l} \texttt{type}\quad \mathit{Seq}\,\alpha = \\ \quad \mid \mathit{Nil} : \forall\alpha.\, \mathit{Seq}\,\alpha \\ \quad \mid \mathit{Zero} : \forall\alpha.\, \mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha \\ \quad \mid \mathit{One} : \forall\alpha.\, \alpha \to \mathit{Seq}\,(\alpha\times\alpha) \to \mathit{Seq}\,\alpha \end{array}$$

Encoded as:

$$\mathit{Seq} \triangleq \lambda\alpha.\, \forall\varphi.\, (\forall\alpha.\, \varphi\,\alpha) \to (\forall\alpha.\, \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to (\forall\alpha.\, \alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha) \to \varphi\alpha$$

$$\mathit{Nil} \triangleq \Lambda\alpha.\, \Lambda\varphi.\, \lambda n : \forall\alpha.\, \varphi\,\alpha.\, \lambda z : \forall\alpha.\, \varphi(\alpha\times\alpha) \to \varphi\,\alpha.\, \lambda s : \forall\alpha.\, \alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha.\, n$$

$$\mathit{Zero} \triangleq \Lambda\alpha.\, \lambda\ell : \mathit{Seq}\,(\alpha\times\alpha).\ \cdots$$

$$\begin{array}{l} \mathit{One} \triangleq \Lambda\alpha.\, \lambda x : \alpha.\, \lambda\ell : \mathit{Seq}\,(\alpha\times\alpha). \\ \qquad \Lambda\varphi.\, \lambda n : \forall\alpha.\, \varphi\,\alpha.\, \lambda z : \forall\alpha.\, \varphi(\alpha\times\alpha) \to \varphi\,\alpha.\, \lambda s : \forall\alpha.\, \alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha. \\ \qquad\qquad s\ x\ (\ell\ \varphi\ n\ z\ s) \end{array}$$

$$\begin{array}{l} \mathit{fold} \triangleq \Lambda\alpha.\, \Lambda\varphi.\, \lambda n : \forall\alpha.\, \varphi\,\alpha.\, \lambda z : \forall\alpha.\, \varphi(\alpha\times\alpha) \to \varphi\,\alpha.\, \lambda s : \forall\alpha.\, \alpha \to \varphi(\alpha\times\alpha) \to \varphi\,\alpha. \\ \qquad \lambda\ell : \mathit{Seq}\,\alpha.\, \ell\ \varphi\ n\ z\ s \end{array}$$

Cannot be simplified! Indeed φ is applied to both α and $\alpha\times\alpha$.

Non regular ADTs cannot be encoded in System F.

Equality — Encoded with GADT

```
module Eq : EQ = struct
  type (α, β) eq = Eq : (α, α) eq

  let coerce (type a) (type b) (ab : (a,b) eq) (x : a) : b = let Eq = ab in x

  let refl : (α, α) eq = Eq

  (* all these are propagation and automatic with GADTs *)
  let symm (type a) (type b) (ab : (a,b) eq) : (b,a) eq = let Eq = ab in ab
  let trans (type a) (type b) (type c)
      (ab : (a,b) eq) (bc : (b,c) eq) : (a,c) eq = let Eq = ab in bc

  let lift (type a) (type b) (ab : (a,b) eq) : (a list, b list) eq =
    let Eq = ab in Eq

end
```

Equality — Leibnitz equality in F^ω

$$\mathit{Eq}\ \alpha\ \beta \equiv \forall\varphi.\ \varphi\alpha \to \varphi\beta$$

$$\begin{aligned}
\mathit{Eq} &\triangleq \lambda\alpha.\ \lambda\beta.\ \forall\varphi.\ \varphi\alpha \to \varphi\beta \\
\mathit{coerce} &\triangleq \lambda p.\ \lambda x.\ p\ x \\
&: \forall\alpha.\ \forall\beta.\ \mathit{Eq}\ \alpha\ \beta \to \alpha \to \beta \\
\mathit{refl} &\triangleq \lambda x.\ x \\
&: \forall\alpha.\ \forall\varphi.\ \varphi\alpha \to \varphi\alpha \ \equiv\ \forall\alpha.\ \mathit{Eq}\ \alpha\ \alpha \\
\mathit{symm} &\triangleq \lambda p.\ p\ (\mathit{refl}) \\
&: \forall\alpha.\ \forall\beta.\ \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \alpha \\
\mathit{trans} &\triangleq \lambda p.\ \lambda q.\ q\ p \\
&: \forall\alpha.\ \forall\beta.\ \forall\gamma.\ \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \gamma \to \mathit{Eq}\ \alpha\ \gamma \\
\mathit{lift} &\triangleq \lambda p.\ p\ (\mathit{refl}) \\
&: \forall\alpha.\ \forall\beta.\ \forall\varphi.\ \mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)
\end{aligned}$$

Equality — Leibniz equality in F^ω

$Eq\ \alpha\ \beta \equiv \forall\varphi.\,\varphi\alpha \to \varphi\beta$

$$
\begin{array}{lcl}
Eq & \triangleq & \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta \\
coerce & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : Eq\ \alpha\ \beta.\,\lambda x : \alpha.\ p\ (\lambda\gamma.\,\gamma)\ x \\
 & : & \forall\alpha.\,\forall\beta.\,Eq\ \alpha\ \beta \to \alpha \to \beta \\
refl & \triangleq & \Lambda\alpha.\,\Lambda\varphi.\,\lambda x : \varphi\alpha.\ x \\
 & : & \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \ \equiv\ \forall\alpha.\,Eq\ \alpha\ \alpha \\
symm & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : Eq\ \alpha\ \beta.\ p\ (\lambda\gamma.\,Eq\ \gamma\ \alpha)\ (refl\ \alpha) \\
 & : & \forall\alpha.\,\forall\beta.\,Eq\ \alpha\ \beta \to Eq\ \beta\ \alpha \\
trans & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\gamma.\,\lambda p : Eq\ \alpha\ \beta.\,\lambda q : Eq\ \beta\ \gamma.\ q\ (Eq\ \alpha)\ p \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\gamma.\,Eq\ \alpha\ \beta \to Eq\ \beta\ \gamma \to Eq\ \alpha\ \gamma \\
lift & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\varphi.\,\lambda p : Eq\ \alpha\ \beta.\ p\ (\lambda\gamma.\,Eq\ (\varphi\alpha)\ (\varphi\gamma))\ (refl\ (\varphi\alpha)) \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\varphi.\,Eq\ \alpha\ \beta \to Eq\ (\varphi\alpha)\ (\varphi\beta)
\end{array}
$$

Equality

Leibnitz equality in F^ω

$\mathit{Eq}\ \alpha\ \beta \equiv \forall \varphi.\, \varphi\alpha \to \varphi\beta$

$$\mathit{Eq} \triangleq \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta$$

$$\mathit{coerce} \triangleq \Lambda\alpha.\,\Lambda\beta.\,\lambda p : \mathit{Eq}\ \alpha\ \beta.\,\lambda x : \alpha.\ p\ (\lambda\gamma.\,\gamma)\ x$$
$$: \forall\alpha.\,\forall\beta.\,\mathit{Eq}\ \alpha\ \beta \to \alpha \to \beta$$

$$\mathit{refl} \triangleq \Lambda\alpha.\,\Lambda\varphi.\,\lambda x : \varphi\alpha.\ x$$
$$: \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \ \equiv\ \forall\alpha.\,\mathit{Eq}\ \alpha\ \alpha$$

$$\mathit{symm} \triangleq \Lambda\alpha.\,\Lambda\beta.\,\lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\,\mathit{Eq}\ \gamma\ \alpha)\ (\mathit{refl}\ \alpha)$$
$$: \forall\alpha.\,\forall\beta.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \alpha$$

$: \mathit{Eq}\ \alpha\ \alpha \to \mathit{Eq}\ \beta\ \alpha$

$$\mathit{trans} \triangleq \Lambda\alpha.\,\Lambda\beta.\,\Lambda\gamma.\,\lambda p : \mathit{Eq}\ \alpha\ \beta.\,\lambda q : \mathit{Eq}\ \beta\ \gamma.\ q\ (\mathit{Eq}\ \alpha)\ p$$
$$: \forall\alpha.\,\forall\beta.\,\forall\gamma.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ \beta\ \gamma \to \mathit{Eq}\ \alpha\ \gamma$$

$$\mathit{lift} \triangleq \Lambda\alpha.\,\Lambda\beta.\,\Lambda\varphi.\,\lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\,\mathit{Eq}\ (\varphi\alpha)\ (\varphi\gamma))\ (\mathit{refl}\ (\varphi\alpha))$$
$$: \forall\alpha.\,\forall\beta.\,\forall\varphi.\,\mathit{Eq}\ \alpha\ \beta \to \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)$$

Equality

Leibnitz equality in F^ω

$\mathit{Eq}\ \alpha\ \beta \equiv \forall\varphi.\ \varphi\alpha \rightarrow \varphi\beta$

$$\mathit{Eq} \triangleq \lambda\alpha.\lambda\beta.\forall\varphi.\varphi\alpha \rightarrow \varphi\beta$$

$$\mathit{coerce} \triangleq \Lambda\alpha.\Lambda\beta.\lambda p : \mathit{Eq}\ \alpha\ \beta.\lambda x : \alpha.\ p\ (\lambda\gamma.\gamma)\ x$$
$$: \forall\alpha.\forall\beta.\mathit{Eq}\ \alpha\ \beta \rightarrow \alpha \rightarrow \beta$$

$$\mathit{refl} \triangleq \Lambda\alpha.\Lambda\varphi.\lambda x : \varphi\alpha.\ x$$
$$: \forall\alpha.\forall\varphi.\varphi\alpha \rightarrow \varphi\alpha \quad\equiv\quad \forall\alpha.\mathit{Eq}\ \alpha\ \alpha$$

$$\mathit{symm} \triangleq \Lambda\alpha.\Lambda\beta.\lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\mathit{Eq}\ \gamma\ \alpha)\ (\mathit{refl}\ \alpha)$$
$$: \forall\alpha.\forall\beta.\mathit{Eq}\ \alpha\ \beta \rightarrow \mathit{Eq}\ \beta\ \alpha$$

$$\mathit{trans} \triangleq \Lambda\alpha.\Lambda\beta.\Lambda\gamma.\lambda p : \mathit{Eq}\ \alpha\ \beta.\lambda q : \mathit{Eq}\ \beta\ \gamma.\ q\ (\mathit{Eq}\ \alpha)\ p$$
$$: \forall\alpha.\forall\beta.\forall\gamma.\mathit{Eq}\ \alpha\ \beta \rightarrow \mathit{Eq}\ \beta\ \gamma \rightarrow \mathit{Eq}\ \alpha\ \gamma$$

$q\ (\mathit{Eq}\ \alpha) : \mathit{Eq}\ \alpha\ \beta \rightarrow \mathit{Eq}\ \alpha\ \gamma$

$$\mathit{lift} \triangleq \Lambda\alpha.\Lambda\beta.\Lambda\varphi.\lambda p : \mathit{Eq}\ \alpha\ \beta.\ p\ (\lambda\gamma.\mathit{Eq}\ (\varphi\alpha)\ (\varphi\gamma))\ (\mathit{refl}\ (\varphi\alpha))$$
$$: \forall\alpha.\forall\beta.\forall\varphi.\mathit{Eq}\ \alpha\ \beta \rightarrow \mathit{Eq}\ (\varphi\alpha)\ (\varphi\beta)$$

Equality

Leibnitz equality in F^ω

$Eq\ \alpha\ \beta \equiv \forall \varphi.\, \varphi\alpha \to \varphi\beta$

$$
\begin{array}{lcl}
Eq & \triangleq & \lambda\alpha.\,\lambda\beta.\,\forall\varphi.\,\varphi\alpha \to \varphi\beta \\
coerce & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : Eq\ \alpha\ \beta.\,\lambda x : \alpha.\ p\ (\lambda\gamma.\,\gamma)\ x \\
 & : & \forall\alpha.\,\forall\beta.\,Eq\ \alpha\ \beta \to \alpha \to \beta \\
refl & \triangleq & \Lambda\alpha.\,\Lambda\varphi.\,\lambda x : \varphi\alpha.\ x \\
 & : & \forall\alpha.\,\forall\varphi.\,\varphi\alpha \to \varphi\alpha \ \equiv\ \forall\alpha.\,Eq\ \alpha\ \alpha \\
symm & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\lambda p : Eq\ \alpha\ \beta.\ p\ (\lambda\gamma.\,Eq\ \gamma\ \alpha)\ (refl\ \alpha) \\
 & : & \forall\alpha.\,\forall\beta.\,Eq\ \alpha\ \beta \to Eq\ \beta\ \alpha \\
trans & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\gamma.\,\lambda p : Eq\ \alpha\ \beta.\,\lambda q : Eq\ \beta\ \gamma.\ q\ (Eq\ \alpha)\ p \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\gamma.\,Eq\ \alpha\ \beta \to Eq\ \beta\ \gamma \to Eq\ \alpha\ \gamma \\
lift & \triangleq & \Lambda\alpha.\,\Lambda\beta.\,\Lambda\varphi.\,\lambda p : Eq\ \alpha\ \beta.\ p\ (\lambda\gamma.\,Eq\ (\varphi\alpha)\ (\varphi\gamma))\ (refl\ (\varphi\alpha)) \\
 & : & \forall\alpha.\,\forall\beta.\,\forall\varphi.\,Eq\ \alpha\ \beta \to Eq\ (\varphi\alpha)\ (\varphi\beta)
\end{array}
$$

$: Eq\ (\varphi\alpha)\ (\varphi\alpha) \to Eq\ (\varphi\alpha)\ (\varphi\beta)$

Equality — Leibnitz equality in F^ω

We implemented parts of the coercions of System Fc.

- We do not have decomposition of equalities (the inverse of *Lift*).
- This requires injectivity of type operators, which is not given.
- Equivalences and liftings must be written explicitly, while they are implicit with GADTs.

Some GATDs can be encoded, using equality plus existential types.

Contents

A hierarchy of type systems

Kinds have a rank:

- the base kind $*$ is of rank 1
- kinds $* \Rightarrow *$ and $* \Rightarrow * \Rightarrow *$ have rank 2. They are the kinds of type functions taking type parameters of base kind.
- kind $(* \Rightarrow *) \Rightarrow *$ has rank 3—it is a type function whose parameter is itself a simple type function (of rank 1).
- more generally, $\mathit{rank}\,(\kappa_1 \Rightarrow \kappa_2) = \max(1 + \mathit{rank}\,\kappa_1, \mathit{rank}\,\kappa_2)$

This defines a sequence $F^1 \subseteq F^2 \subseteq F^3 \ldots \subseteq F^\omega$ of type systems of increasing expressiveness, where F^n only uses kinds of rank n, whose limit is F^ω and where System F is F^1.

(*Ranks are sometimes shifted by one, starting with* $F = F^2$.)

Most examples in practice (and those we wrote) are in F^2, just above F.

F^ω with several base kinds

We could have several base kinds, *e.g.* $*$ and *field* with type constructors:

$$\begin{aligned} \textit{filled} &: * \Rightarrow \textit{field} \qquad\qquad & \textit{box} &: \textit{field} \Rightarrow * \\ \textit{empty} &: \textit{field} \end{aligned}$$

Prevents ill-formed types such as $\textit{box}\,(\alpha \to \textit{filled}\,\alpha)$.

This allows to build values v of type $\textit{box}\,\theta$ where θ of kind *field* statically tells whether v is *filled* with a value of type τ or *empty*.

Application:

This is used in OCaml for rows of object types, but kinds are hidden to the user:

```
let get (x : ⟨ get : α; .. ⟩) : α = x#get
```

The dots ".." here stand for a variable of another base kind (representing a *row* of types).

System F^ω with equirecursive types

Checking equality of equirecursive types in System F is already non obvious, since unfolding may require α-conversion to avoid variable capture. (See also [Gauthier and Pottier, 2004].)

With higher-order types, it is even trickier, since unfolding at functional kinds could expose new type redexes.

Besides, the language of types would be the simply type λ-calculus with a fix-point operator: type reduction would not terminate.

Therefore type equality would be undecidable, as well as type checking.

A solution is to restrict to recursion at the base kind $\ast$. This allows to define recursive types but not recursive type functions.

Such an extension has been proven sound and and decidable, but only for the weak form or equirecursive types (with the unfolding but not the uniqueness rule)—see [Cai et al., 2016].

System F^ω with equirecursive kinds

Instead, recursion could also occur just at the level of kinds, allowing kinds to be themselves recursive.

Then, the language of types is the simply type λ-calculus with recursive types, equivalent to the untyped λ-calculus—every term is typable. Reduction of types does not terminate and type equality is ill-defined.

A solution proposed by Pottier [2011] is to force recursive kinds to be productive, reusing an idea from an [Nakano, 2000, 2001] for controlling recursion on terms, but pushing it one level up. Type equality becomes well-defined and semi-decidable.

The extension has been used to show that references in System F can be translated away in F^ω with guarded recursive kinds.

Encoding ML modules with *generative* functors

Generative functors can be encoded with existential types.

A functor F has a type of the form:

$$\forall \bar{\alpha}.\, \tau[\bar{\alpha}] \to \exists \bar{\beta}.\, \sigma[\bar{\alpha}, \bar{\beta}]$$

Where:

- $\tau[\bar{\alpha}]$ represents the signature of the argument with some abstract types $\bar{\alpha}$.
- $\exists \bar{\beta}.\, \sigma[\bar{\alpha}, \bar{\beta}]$ represents the signature of the result of the functor application.
- That is, the abstract types $\bar{\alpha}$ are those taken from and shared with the argument.
- Conversely $\bar{\beta}$ are the abstract types created by the application, and have fresh identities independent of the argument.
- Two successive applications with the *same* argument (hence the same α) will create two signatures with incompatible abstract types $\bar{\beta}_1$ and $\bar{\beta}_2$, once the existential is open.

Encoding ML modules with *generative* functors

Generative functors can be encoded with existential types.

A functor F has a type of the form:

$$\forall \bar\alpha.\, \tau[\bar\alpha] \to \exists \bar\beta.\, \sigma[\bar\alpha, \bar\beta]$$

Two applications of F with the same argument:

let module $\mathsf{Z}_1 = \mathsf{F(X)}$ **in**
let module $\mathsf{Z}_2 = \mathsf{F(X)}$ **in** ...

must be understood as:

let $\bar\beta_1$, $\mathsf{Z}_1 =$ unpack (F(X)) **in**
let $\bar\beta_2$, $\mathsf{Z}_2 =$ unpack (F(X)) **in** ...

creating two structures Z_1 and Z_2 with incompatible abstract types $\bar\beta_1$ and $\bar\beta_2$ that cannot interoperate.

(Typically, they contain a field ℓ of respective types β_1 and β_2 so that $Z.\ell = Z'.\ell$ is ill-typed.)

Encoding ML modules with *generative* functors

Generative functors can be encoded with existential types.

A functor F has a type of the form:

$$\forall \bar{\alpha}.\, \tau[\bar{\alpha}] \to \exists \bar{\beta}.\, \sigma[\bar{\alpha}, \bar{\beta}]$$

In the absence of parametric types (such as *list* α), the encoding can be done in System F.

Encoding ML modules with *applicative* functors

Applicative functors can be encoded with *higher-order* existential types.

A functor F has a type of the form:

$$\exists \bar{\varphi}.\, \forall \bar{\alpha}.\, \tau[\bar{\alpha}] \to \sigma[\bar{\alpha}, \bar{\varphi}\bar{\alpha}]$$

Compared with:

$$\forall \bar{\alpha}.\, \tau[\bar{\alpha}] \to \exists \bar{\beta}.\, \sigma[\bar{\alpha}, \bar{\beta}]$$

Encoding ML modules with *applicative* functors

Applicative functors can be encoded with *higher-order* existential types.

A functor F has a type of the form:

$$\exists \bar{\varphi}.\, \forall \bar{\alpha}.\, \tau[\bar{\alpha}] \to \sigma[\bar{\alpha}, \bar{\varphi}\bar{\alpha}]$$

Compared with:

$$\forall \bar{\alpha}.\, \tau[\bar{\alpha}] \to \exists \bar{\beta}.\, \sigma[\bar{\alpha}, \bar{\beta}]$$

That is:

- $\sigma[\bar{\alpha}, \bar{\varphi}\bar{\alpha}]$ represents the signature of the result of the functor application.
- $\bar{\varphi}\bar{\alpha}$ are the abstract types created by the application. Each $\varphi\bar{\alpha}$ is a new abstract type—one we know nothing about, as it is the application of an abstract type to $\bar{\alpha}$.
- However, two successive applications with the *same* argument (hence the same $\bar{\alpha}$) will create two *compatible* structures whose signatures have the same *shared* abstract types $\bar{\varphi}\bar{\alpha}$.

Encoding ML modules with *applicative* functors

Applicative functors can be encoded with *higher-order* existential types.

A functor F has a type of the form:

$$\exists \bar{\varphi}. \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \sigma[\bar{\alpha}, \bar{\varphi}\bar{\alpha}]$$

Compared with:

$$\forall \bar{\alpha}. \tau[\bar{\alpha}] \to \exists \bar{\beta}. \sigma[\bar{\alpha}, \bar{\beta}]$$

The two applications of F:

let $\bar{\beta}_1$, Z_1 = unpack (F(X)) **in**
let $\bar{\beta}_2$, Z_2 = unpack (F(X)) **in** ...

becomes:

let $\bar{\varphi}$, F = unpack F **in**
let Z_1 = F(X) **in**
let Z_2 = F(X) **in** ...

Encoding ML modules with *applicative* functors

Applicative functors can be encoded with *higher-order* existential types.

A functor F has a type of the form:

$$\exists \bar{\varphi}. \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \sigma[\bar{\alpha}, \bar{\varphi}\bar{\alpha}]$$

More generally:

$$\exists \bar{\varphi}. \forall \bar{\alpha}. \tau[\bar{\alpha}] \to \exists \bar{\beta}. \sigma[\bar{\alpha}, \bar{\varphi}\bar{\alpha}, \bar{\beta}]$$

Where we can have

- applicative abstract types
- generative abstract types

simultaneously.

Encoding ML modules with *applicative* functors

Applicative functors can be encoded with *higher-order* existential types.

A functor F has a type of the form:

$$\exists \bar\varphi.\, \forall \bar\alpha.\, \tau[\bar\alpha] \to \sigma[\bar\alpha, \bar\varphi\bar\alpha]$$
$$\forall \bar\alpha.\, \tau[\bar\alpha] \to \exists \bar\beta.\, \sigma[\bar\alpha, \bar\beta]$$

Or we may just alternate between generative and applicative functors.

Encoding ML modules with *applicative* functors

Applicative functors can be encoded with *higher-order* existential types.

A functor F has a type of the form:

$$\exists \bar\varphi.\, \forall \bar\alpha.\, \tau[\bar\alpha] \to \sigma[\bar\alpha, \bar\varphi\bar\alpha]$$
$$\forall \bar\alpha.\, \tau[\bar\alpha] \to \exists \bar\beta.\, \sigma[\bar\alpha, \bar\beta]$$

Summary:

- An abstract type of higher-order kind can be used to generate new (partially) abstract types !
- F^ω may encode of *applicative* functors using this mechanism to generate abstract types that can be shared.

  See [Rossberg et al., 2014] and [Rossberg, 2018], and also [Blaudeau, 2021] for ongoing work.

System F^ω in OCaml

Second-order polymorphism in OCaml

- Via polymorphic methods

  let id = **object method** f : $\alpha.\ \alpha \to \alpha$ = **fun** x → x end
  let y (x : ⟨f : $\alpha.\ \alpha \to \alpha$⟩) = x#f x **in** y id

System F^ω in OCaml

Second-order polymorphism in OCaml

- Via polymorphic methods

  let id = **object method** f : $\alpha.\ \alpha \to \alpha$ = **fun** x → x end
  let y (x : ⟨f : $\alpha.\ \alpha \to \alpha$⟩) = x#f x **in** y id

- Via first-class modules

  module type S = sig **val** f : $\alpha \to \alpha$ end
  let id = (**module** struct **let** f x = x end : S)
  let y (x : (**module** S)) = **let module** X = (**val** x) **in** X.f x **in** y id

System F^{ω} in OCaml

Second-order polymorphism in OCaml

- Via polymorphic methods
- Via first-class modules

Higher-order types in OCaml

- In principle, they could be encoded with first-class modules.
- Not currently possible, due to (unnecessary) restrictions.
- Modular explicits, an extension that allows a better integration of abstraction over first-class modules will remove these limitations and allow a light-weight encoding of F^{ω}—with boiler-plate glue code.

System F^ω in OCaml . . . with modular explicits

Available at git@github.com:mrmr1993/ocaml.git

```
module type s = sig type t end
module type op = functor (A:s) → s

let dp {F:op} {G:op} {A:s} {B:s} (f:{C:s} → F(C).t → G(C).t)
    (x : F(A).t) (y : F(B).t) : G(A).t ∗ G(B).t = f {A} x, f {B} y
```

And its two specialized versions:

```
let dp1 (type a) (type b) (f : {C:s} → C.t → C.t) : a → b → a ∗ b =
  let module F(C:s) = C in let module G = F in
  let module A = struct type t = a end in
  let module B = struct type t = b end in
  dp {F} {G} {A} {B} f

let dp2 (type a) (type b) (f : a → b) : a → a → b ∗ b =
  let module A = struct type t = a end in
  let module B = struct type t = b end in
  let module F(C:s) = A in let module G(C:s) = B in
  dp {F} {G} {A} {B} (fun {C:s} → f)
```

System F^ω in Scala-3

Higher-order polymorphism a la System F^ω is available in Scala-3.

The monad example (with some variation on the signature) is:

```
trait Monad [F[_]] {
  def pure [A] (x: A) : F[A]
  def flatMap [A, B] (fa: F[A]) (f: A ⇒ F[B]) : F[B]
}
```

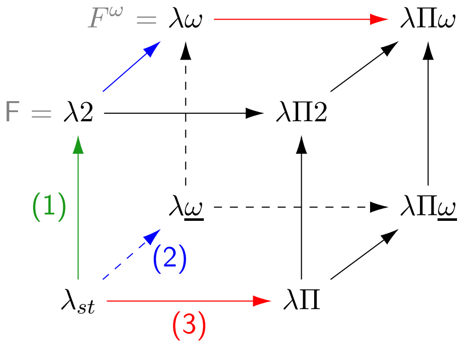
See https://www.baeldung.com/scala/dotty-scala-3

Still, this feature of Scala-3 is not emphasized

- It was not directly available in previous versions of Scala.
- Scala's syntax and other complex features of Scala are obfuscating.

What's next? Dependent types!

Barendregt's λ-cube



(1) Term abstraction on Types (example: System F)

(2) Type abstraction on Types (example: F^ω)

(3) *Type abstraction on Terms (dependent types)*

Bibliography I

(Most titles have a clickable mark "▷" that links to online versions.)

▷ Clément Blaudeau. OCaml modules: formalization, insights and improvements. Master's thesis, École polytechnique fédérale de Lausanne, September 2021.

▷ Yufei Cai, Paolo G. Giarrusso, and Klaus Ostermann. System F-omega with equirecursive types for datatype-generic programming. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 30–43. ACM, 2016. doi: 10.1145/2837614.2837660.

▷ Nadji Gauthier and François Pottier. Numbering matters: First-order canonical forms for second-order recursive types. In *Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming (ICFP'04)*, pages 150–161, September 2004. doi: http://doi.acm.org/10.1145/1016850.1016872.

▷ Oleg Kiselyov. Higher-kinded bounded polymorphism. web page.

Bibliography II

▷ Sophie Malecki. Proofs in system fω can be done in system fω1. In Dirk van Dalen and Marc Bezem, editors, *Computer Science Logic*, pages 297–315, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69201-0.

▷ Hiroshi Nakano. A modality for recursion. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 255–266, June 2000.

▷ Hiroshi Nakano. Fixed-point logic with the approximation modality and its Kripke completeness. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 2215 of *Lecture Notes in Computer Science*, pages 165–182. Springer, October 2001.

François Pottier. A typed store-passing translation for general references. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, Austin, Texas, January 2011. Supplementary material.

▷ Andreas Rossberg. 1ml - core and modules united. *J. Funct. Program.*, 28:e22, 2018. doi: 10.1017/S0956796818000205.

Bibliography III

- Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. *J. Funct. Program.*, 24(5):529–607, 2014. doi: 10.1017/S0956796814000264.
- Jeremy Yallop and Leo White. Lightweight higher-kinded polymorphism. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming*, pages 119–135, Cham, 2014. Springer International Publishing. ISBN 978-3-319-07151-0.