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

Didier Rémy

October 6, 2021

Plan of the course

Metatheory of System F

ADTs, Recursive types, Existential types, GATDs

Going higher order with F^ω!

Logical relations

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 toplevel polymorphism

- Already, extremely useful! (avoiding dupplication of code)
- 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 *distrib_pair*, 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 *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$^\omega$ Kinds

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

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

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

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

Add and check kinds on type abstractions and 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$^\omega$ Type functions

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

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

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)

With implicit kinds

Syntax

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

Kinding rules

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

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

Typing rules

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

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

F^ω, dynamic semantics

The semantics is unchanged (modulo kind annotations in terms)

$$\begin{array}{lcl} V & ::= & \lambda x\!:\!\tau.\,M \mid \Lambda\alpha :: \kappa.\,V \\ E & ::= & [\,]\,M \mid V\,[\,] \mid [\,]\,\tau \mid \Lambda\alpha :: \kappa.\,[\,] \end{array}$$

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

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

No type reduction

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

Kinds are erasable

- Reduction preserves kinds.
- Kinds are just ignored during the reduction (they need not be reduced). In fact, kinds can be 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, but for *full reduction* (we have only proved it for CBV).

It is a key that reduction terminates.
(Again, we have only proved it for CBV.)

Contents

Expressiveness

More polymorphism

- distrib_pair

Abstraction over type operators

- monads
- encoding of existentials

Encodings

- non regular datatypes
- equality

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

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

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

call it distrib_pair 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 had in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\text{distrib\_pair}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2\\ \quad : \forall\alpha_1.\,\forall\alpha_2.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{distrib\_pair}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)\\ \quad : \forall\alpha_1.\,\forall\alpha_2.\,(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

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

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

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

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

call it distrib_pair 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 had in System F:

$$\Lambda\alpha_1.\,\Lambda\alpha_2.\,\text{distrib\_pair}\ (\lambda\alpha.\,\alpha_1)\ (\lambda\alpha.\,\alpha_2)\ \alpha_1\ \alpha_2\\ \quad : \forall\alpha_1.\,\forall\alpha_2.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2$$

$$\text{distrib\_pair}\ (\lambda\alpha.\alpha)\ (\lambda\alpha.\alpha)\\ \quad : \forall\alpha_1.\,\forall\alpha_2.\,(\forall\alpha.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2$$

Still, the type of distrib_pair is not principal. φ and ψ could depend on two variables, *i.e.* be of kind $\ast \Rightarrow \ast \Rightarrow \ast$, 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).

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

(Notice that M is itself of higher kind)

A generic map function: can then be defined:

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

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).

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

(Notice that M is itself of higher kind)

A generic map function: can then be defined:

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

Abstracting over type operators

Available in Haskell —without β-reduction

- $\varphi\alpha$ is treated as a type $App(\varphi, \alpha)$ where $App : (\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2$
- 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 does enable 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.

Encoding of existentials — Limits of System F

We saw

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

Hence,

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

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

In System F^ω, we may defined

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

Allows abstraction at higher kinds!

Encoding of existentials — Limits of System F

We saw

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

Hence,

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

This requires a different code for each type τ

To have a unique code, we need to abstract over the type function $\lambda\alpha.\,\tau$:

In System F^ω, we may defined

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

Allows abstraction at higher kinds!

Exploiting kinds

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

$$\tau = \alpha \mid \lambda\alpha.\tau \mid \tau\ \tau \mid \tau \to \tau \mid g$$

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

$$\begin{array}{lcl} \times & :: & * \Rightarrow * \Rightarrow * \\ + & :: & * \Rightarrow * \Rightarrow \kappa \\ \forall & :: & (\kappa \Rightarrow *) \Rightarrow * \\ \exists & :: & (\kappa \Rightarrow *) \Rightarrow * \end{array} \qquad \begin{array}{lcl} (\tau \times \tau) & \triangleq & (\times)\ \tau_1\ \tau_2 \\ (\tau + \tau) & \triangleq & (+)\ \tau_1\ \tau_2 \\ \forall\varphi.\tau & \triangleq & \forall(\lambda\varphi.\tau) \\ \exists\varphi.\tau & \triangleq & \exists(\lambda\varphi.\tau) \end{array}$$

Church encoding of regular ADT — List

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

Church encoding (CPS style) in System F

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

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

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

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

Actually not !

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 regular ADT — List

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

Church encoding (CPS style) in System F

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

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

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

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

Actually not !

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:

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

$$
\begin{aligned}
\mathit{Nil} &\triangleq \lambda n.\,\lambda z.\,\lambda s.\, n \\
\mathit{Zero} &\triangleq \lambda \ell.\,\lambda n.\,\lambda z.\,\lambda s.\, z\ (\ell\ n\ z\ s) \\
\mathit{One} &\triangleq \lambda x.\,\lambda \ell.\,\lambda n.\,\lambda z.\,\lambda s.\, s\ x\ (\ell\ n\ z\ s)
\end{aligned}
$$

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

Cannot be simplified! Indeed φ is applied to both α and $\alpha \times \alpha$.
Non regular ADTs cannot be encoded in System F.

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

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

Encoded as:

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

$$\begin{aligned} \mathit{Nil} &\triangleq \lambda n.\,\lambda z.\,\lambda s.\, n \\ \mathit{Zero} &\triangleq \lambda\ell.\,\lambda n.\,\lambda z.\,\lambda s.\, z\ (\ell\ n\ z\ s) \\ \mathit{One} &\triangleq \lambda x.\,\lambda\ell.\,\lambda n.\,\lambda z.\,\lambda s.\, s\ x\ (\ell\ n\ z\ s) \end{aligned}$$

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

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 ('a, 'b) eq = Eq : ('a, 'a) eq

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

  let refl : ('a, 'a) eq = Eq

  (* all these are propagation are 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 Leibniz equality in F^ω

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

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

Equality

Leibnitz equality in F^ω

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

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

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 the type operator, 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.

A hierarchy of type systems

Kinds have a rank:

- the base kind $*$ is of rank 0
- kinds $* \Rightarrow *$ and $* \Rightarrow * \Rightarrow *$ have rank 1. They are the kinds of type functions taking type parameters of base kind.
- kind $(* \Rightarrow *) \Rightarrow *$ has rank 2—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 $F^0 \subseteq F^1 \subseteq F^2 \ldots \subseteq F^\omega$ 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^0.

Note that ranks are often shifted by one, starting with $F = F^1$ or even by 2, starting with $F = F^2$.

Most examples in practice (and those we wrote) are in F^1, just above F.

Extensions

Abstraction over kinds?

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

Motivation: distrib_pair does not have a principal type.

F^ω with several base kinds

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

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

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

This allows to build values v of type $\mathit{box}\,\theta$ 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 : 'a; .. >) : 'a = x#get
```

The dots ".." stands 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 alpha-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 $*$. 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 become 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.

System F^ω For applicative functors

Generative ML modules (without parametric types) can be encoding in System F with existential types.

- A functor F has a type of the form: $\forall\alpha.\,\tau[\alpha] \to \exists\beta.\,.\sigma[\alpha,\beta]$
- If X, Y has type $\tau[\rho]$, then two successive applications F(X) and F(X) have types $\exists\beta.\,[\rho,\beta]$ with different abstract types β and cannot interoperate (on components involving β).

  let Y = *unpack* $F\,X$ *in*
  let Z = *unpack* $F\,X$ *in* is ill-typed
  $Y = Z$

However, *applicative* modules require the use of F^ω to keep track of type equalities! See [Rossberg et al., 2014] and [Rossberg, 2018].

- A functor F has a type of the form: $\exists\varphi.\,\forall\alpha.\,\tau[\alpha] \to \sigma[\alpha, \varphi\alpha]$
  or when open $\forall\alpha.\,\tau[\alpha] \to \sigma[\alpha,\psi\rho]$ for some unknown ψ.
- Then if X has type $\tau[\rho]$, two successive applications F(X) and F(X) have the same type $\sigma[\rho,\varphi\rho]$ sharing the abstract type (application) $\psi\rho$.
- Hence, the two applications can interoperate,

System F^ω in OCamll

Second-order polymorphism in OCaml

- Via polymorphic methods

```
let id = object method f : 'a. 'a → 'a = fun x → x end
let y (x : <f : 'a. 'a → 'a>) = x#f x in y id
```

- Via first-class modules

```
module type S = sig val f : 'a → 'a 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
```

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 OCamll

Second-order polymorphism in OCaml

- Via polymorphic methods

```
let id = object method f : 'a. 'a → 'a = fun x → x end
let y (x : <f : 'a. 'a → 'a>) = x#f x in y id
```

- Via first-class modules

```
module type S = sig val f : 'a → 'a 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
```

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 now accessible 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 emphrasized

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

What's next? Dependent types!

Barendregt's λ-cube

$$\begin{tikzcd}
 & F^{\omega} = \lambda\omega \arrow[rr] & & \lambda\Pi\omega \\
\mathsf{F} = \lambda 2 \arrow[ur] \arrow[rr] & & \lambda\Pi 2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu, dashed] \arrow[rr, dashed] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda_{st} \arrow[uu, "(1)"] \arrow[ur, dashed, "(2)"'] \arrow[rr, "(3)"'] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

(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.)

▷ 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.