Type systems for programming languages

Didier Rémy

Academic year 2020-2021
Version of August 25, 2022

Contents

Chapter 6

Fomega: higher-kinds and higher-order types

6.1 Introduction

Polymorphism in System F Compare with simple types, which lacks polymorphism, and thus forces many functions to be duplicated at different types. ML style polymorphism is a considerable improvement by avoiding most of code duplication. Local let-bound polymorphism is also permitted in ML, but is less used in practice. However, core ML still lacks first-class polymorphism, which means higher-rank polymorphism and the lack of primitive existential types.

In ML, the module system allows for type abstraction, which for made the lack of existential types more sustainable—when programming in the large. First class-existential types are encodable in ML with first-class modules. They are now directly available via GADTs.

System F solves enables first-class existential and universal-types in the core language. This increases expressiveness by enabling encoding of data structures and many more programming patterns. Still, System F polymorphism is limited. . . .

Limits of System F: $\lambda f x y.\,(f\ x,\ f\ y)$ Although System F has higher-rank polymorphism, this is still sometimes limited. For example, the map function on pairs whose untyped code if $\lambda f.\,\lambda x.\,\lambda y.\,(fx, fy)$, says distrib_pair can be given the following *incompatible* types in System F:

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

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

However, System F 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. Hence, it cannot give a type to distrib_pair that subsumes both types above.

To solve this, we need to abstract over σ and τ, *i.e.* over *type functions*, of kind $\star \to \star$:

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

This is what System F^ω allows.

6.2 From System F to System F$^\omega$

Kinds To emphasize the small difference between System F and System F^ω, we first introduce kinds in the presentation of System F—without changing expressiveness. That is, we write κ for the single kind of types.

Well-formedness of types $\Gamma \vdash \tau$ may then be written as a well-kinding judgment $\Gamma \vdash \tau : *$, defined inductively as follows:

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

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

We accordingly add kind annotations on type abstractions and type applications:

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

So far, this is an equivalent formalization of System F.

Type functions We now add type functions, moving form System F to System F^ω. For that purpose, we redefine kinds, so as to introduce kinds of type functions:

$$\kappa ::= * \mid \kappa \Rightarrow \kappa$$

We may now introduce type function and type application in type expressions:

$$\tau ::= \ldots \mid \lambda\alpha{::}\,\kappa.\,\tau \mid \tau\;\tau$$

with the following kinding rules:

$$\textsc{WfTypeApp}\ \frac{\Gamma \vdash \tau_1 : \kappa_2 \Rightarrow \kappa_1 \qquad \Gamma \vdash \tau_2 : \kappa_2}{\Gamma \vdash \tau_1\;\tau_2 : \kappa_1} \qquad \textsc{WfTypeAbs}\ \frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda\alpha{::}\,\kappa_1.\,\tau : \kappa_1 \Rightarrow \kappa_2}$$

Type reduction Types are now equipped with β-reduction:

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

which is applicable in *any type context*.

Notice that type reduction is the same as (full reduction) in simply-typed λ-calculus when kinds and types now play the role of types and terms. It thus preserves well-kinding and kinds[1]. Hence, kinds are erasable. Kinds may only be checked when reading type expressions and ignored afterwards. As types, they do not contribute to the reduction.

Type reduction induces a notion of β-equivalence on types, which is decidable: for example, by normalization (which terminates) and comparison of normal forms—although an efficient implementation would reduce terms by need.

Typing of expressions is up to type equivalence, *i.e.* β-conversion:

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

Notice that well-typedness $\Gamma \vdash M : \tau$ ensures well-kindedness $\Gamma \vdash \tau : *$. Notice that decidability of type checking in System F^ω relies on decidability of type equivalence.

However, we need not reduce types inside terms. Type reduction is needed for type conversion during typechecking but such reduction need not be performed on terms.

6.2.1 Properties

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

- *Type soundness.* The proof is by *subject reduction* and *progress*.
- *Termination of reduction.* This holds in the absence of other constructs that can be use to introduce recursion, such as recursive types, recursive definitions or side effects (references, exceptions, control, *etc.*).
- *Typechecking is decidable.* This requires reduction at the level of types to check type equality. Checking type equality can be performed by putting types in normal forms using full reduction (on types)—or just in head normal forms. Normal forms for types exists as the language of type is a simply-typed λ-calculus (where kinds plays the role of types).

[1] We have only proved subject reduction for CBV, though in the previous lessons

Syntax

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

Kinding rules

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

KVar
$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa}$$

KArrow
$$\frac{\Gamma \vdash \tau_1 : * \qquad \Gamma \vdash \tau_2 : *}{\Gamma \vdash \tau_1 \to \tau_2 : *}$$

WfTypeForall
$$\frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \forall \alpha{::}\kappa.\tau : *}$$

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

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

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

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

Dynamic semantics (unchanged, up to kind annotations in terms)

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

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

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

Figure 6.1: System F^ω, altogether

6.3 Expressiveness

System F^ω increases expressiveness and allows to solve the limitations of System F discussed in the introduction.

Just adding more polymorphism on ad hoc examples such as distrib_pair, exploiting abstraction over type operators, such as examples with monads, or the encoding of existential types, or more advanced encodings such as non regular datatypes, and type equality.

6.3.1 Distrib pair in System F^ω

We may now type the example of distrib_pair, whose implicitly typed definition is $\lambda f x y.\,(f\ x,\ f\ y)$ by abstracting over (one parameter) type *functions*, *i.e.* type functions of kind $\star \to \star$. That is, the explicitly typed version of distrib_pair is:

$$\begin{array}{l}\Lambda\varphi\ .\Lambda\psi\ .\Lambda\tau_1.\Lambda\alpha_2.\\ \quad \lambda(f : \forall\,(\alpha :: *)\,.\,\varphi\ \alpha \to \psi\ \alpha).\,\lambda x : \varphi\ \alpha_1.\,\lambda y : \varphi\ \alpha_2.\,(f\ \alpha_1\ x,\ f\ \alpha_2\ y)\end{array}$$

of type:

$$\begin{array}{l}\forall(\varphi :: * \Rightarrow *).\forall(\psi :: * \Rightarrow *).\forall(\alpha_1 :: *).\forall(\alpha_2 :: *).\\ \quad (\forall\,(\alpha :: *)\,.\,\varphi\ \alpha \to \psi\ \alpha) \to \varphi\ \alpha_1 \to \varphi\ \alpha_2 \to \psi\ \alpha_1 \times \psi\ \alpha_2\end{array}$$

We may recover, the two incomparable types it had in System F:

$$\begin{array}{l}\Lambda\,(\alpha_1 :: *)\,.\Lambda\,(\alpha_2 :: *)\,.\mathsf{distrib\_pair}\ (\lambda\,(\alpha :: *)\,.\,\alpha_1)\ (\lambda\,(\alpha :: *)\,.\,\alpha_2)\ \alpha_1\ \alpha_2\\ \quad : \forall\,(\alpha_1 :: *)\,.\,\forall\,(\alpha_2 :: *)\,.\,(\alpha_1 \to \alpha_2) \to \alpha_1 \to \alpha_1 \to \alpha_2 \times \alpha_2\end{array}$$

and

$$\begin{array}{l}\mathsf{distrib\_pair}\ (\lambda(\alpha :: *).\alpha)\ (\lambda(\alpha :: *).\alpha)\\ \quad : \forall\,(\alpha_1 :: *)\,.\,\forall\alpha_2.\,(\forall\,(\alpha :: *)\,.\,\alpha \to \alpha) \to \alpha_1 \to \alpha_2 \to \alpha_1 \times \alpha_2\end{array}$$

While the type of distrib_pair in System F^ω is much more general than in System F, it is still not principal. For example, φ and ψ could depend on two variables, *i.e.* be of kind $* \Rightarrow * \Rightarrow *$, or many other kinds.

6.3.2 Abstracting over type operators

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

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

(Notice that M is itself of higher kind.)

For example, a generic map function, can then be defined as follows:

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

Abstraction over type operators is available in Haskell—but without β-reduction: type application $\varphi\ \ \alpha$ is encoded as a first-order type $\mathsf{App}\,(\varphi\,,\alpha)$ where App is a binary (application) symbol of kind $(\kappa_1 \Rightarrow \kappa_2) \Rightarrow \kappa_1 \Rightarrow \kappa_2$.

Interestingly, this approach is compatible with type inference, which is based on first-order unification. However, there is no β-reduction at the level of types, that is:

$$\varphi\ \alpha = \psi\ \beta \iff \varphi\ = \psi\ \wedge \alpha = \beta$$

Therefore, this does not have the expressiveness of System F^ω at all.

Abstraction over type operators is also encodable with OCaml modules. See **?** (and also **?**). As in Haskell, the encoding does not handle type β-reduction and as a consequence is compatible with type inference at higher kinds.

6.3.3 Encoding of existential types

We saw the encoding of existential types in System F:

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

Hence, existential types could be provided as a family of primitives

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

(and a similar encoding for $[\![\mathsf{unpack}_{\exists\alpha.\tau}]\!]$).

Unfortunately, this requires a different code for each type τ. To have a unique code, we need to abstract over τ which is not possible in System F, but quite natural in System F^ω.

We first extend existential types to abstraction over higher kinding variables, letting $[\![\exists\,(\alpha :: \kappa)\,.\,\tau]\!]$ mean $\forall\,(\beta :: *)\,.\,(\forall\,(\alpha :: \kappa)\,.\,\tau \to \beta) \to \beta$. In fact, we need not introduce a special construct $\exists\,(\alpha :: \kappa)\,.\,\tau$ for that purpose, but just a new type constant $\exists_\kappa$ of kind $\kappa \Rightarrow *$ and write $\exists_\kappa\,(\lambda\,(\alpha :: \kappa)\,.\,\tau)$ for $\exists\,(\alpha :: \kappa)\,.\,\tau$.

Then, we may abstract the encodings over some type variable φ of kind $\kappa \Rightarrow *$, as follows:

$$
\begin{array}{lcl}
[\![\exists\,(\alpha :: \kappa)\,.\,\tau]\!] & = & \forall\,(\beta :: *)\,.\,(\forall\,(\alpha :: \kappa)\,.\,\tau \to \beta) \to \beta \\
\exists_\kappa & = & \lambda\,(\varphi :: \kappa \Rightarrow *)\,.\,\forall\,(\beta :: *)\,.\,(\forall\,(\alpha :: \kappa)\,.\,\varphi\ \ \alpha \to \beta) \to \beta \\
\mathsf{pack}_\kappa & : & \forall\,(\varphi :: \kappa \Rightarrow *)\,.\,\forall\,(\alpha :: \kappa)\,.\,\varphi\ \ \alpha \to \exists_\kappa\,\varphi \\
 & = & \Lambda\,(\varphi :: \kappa \Rightarrow *)\,.\Lambda\,(\alpha :: \kappa)\,. \\
 & & \qquad \lambda x : \varphi\ \ \alpha.\,\Lambda\,(\beta :: *)\,.\lambda k : \forall\,(\alpha :: \kappa)\,.\,(\varphi\ \ \alpha \to \beta).\,k\ \alpha\ x \\
\mathsf{unpack}_\kappa & : & \forall\,(\varphi :: \kappa \Rightarrow *)\,.\,\exists_\kappa\,\varphi \to \forall\,(\beta :: *)\,.\,((\forall\,(\alpha :: \kappa)\,.\,([\![\tau]\!] \to \beta)) \to \beta) \\
 & = & \Lambda\,(\varphi :: \kappa \Rightarrow *)\,.\lambda x : \exists_\kappa\,\varphi\ .\,x
\end{array}
$$

The interest is that the encoding need not be defined at the metalevel, but directly provided as two terms of System F^ω—with may be defined once for all.

This idea of exploiting kinds

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

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

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

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

This is even nicer if System F^ω were extended with kind abstraction (see §6.4)**??**, as we could then just write:

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

where the right hand sides are no more syntactic forms but the application of the type constants $\hat\forall$ and $\hat\exists$ to a kind a type.

6.3.4 Church encoding of non-regular ADT

Regular ADTs can be encoded in System F. For instance, the type list datatype

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

has the following Church (CPS style) encoding:

$$
\begin{array}{lcl}
\mathsf{List} & \triangleq & \lambda\alpha.\,\forall\beta.\,\beta \to (\alpha \to \beta \to \beta) \to \beta \\
\mathsf{Nil} & \triangleq & \Lambda\alpha.\Lambda\beta.\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,n \\
\mathsf{Cons} & \triangleq & \Lambda\alpha.\lambda x : \alpha.\,\lambda\ell : \mathsf{List}\ \alpha. \\
 & & \quad \Lambda\beta.\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,c\ x\ (\ell\ \beta\ n\ c) \\
\mathsf{fold} & \triangleq & \Lambda\alpha.\Lambda\beta.\lambda n : \beta.\,\lambda c : (\alpha \to \beta \to \beta).\,\lambda\ell : \mathsf{List}\ \alpha.\,\ell\ \beta\ n\ c
\end{array}
$$

In fact, we may give use following signature in System F^ω:

$$
\begin{array}{lcl}
\mathsf{List} & \triangleq & \lambda\alpha.\,\forall\varphi\ .\varphi\ \alpha \to (\alpha \to \varphi\ \alpha \to \varphi\ \alpha) \to \varphi\ \alpha \\
\mathsf{Nil} & \triangleq & \Lambda\alpha.\Lambda\varphi\ .\lambda n : \varphi\ \alpha.\,\lambda c : (\alpha \to \varphi\ \alpha \to \varphi\ \alpha).\,n \\
\mathsf{Cons} & \triangleq & \Lambda\alpha.\lambda x : \alpha.\,\lambda\ell : \mathsf{List}\ \alpha. \\
 & & \quad \Lambda\varphi\ .\lambda n : \varphi\ \alpha.\,\lambda c : (\alpha \to \varphi\ \alpha \to \varphi\ \alpha).\,c\ x\ (\ell\ \varphi\ \ n\ c) \\
\mathsf{fold} & \triangleq & \Lambda\alpha.\Lambda\varphi\ .\lambda n : \varphi\ \alpha.\,\lambda c : (\alpha \to \varphi\ \alpha \to \varphi\ \alpha).\,\lambda\ell : \mathsf{List}\ \alpha.\,\ell\ \varphi\ \ n\ c
\end{array}
$$

This seems more abstract since β is now $\varphi\ \alpha$ which may depend on α.

Actually not! Be aware of useless over-generalization! For regular ADTs, since all uses of φ are applied to the same α, this interface is actually no more general that the previous one. However, this additional degree of liberty will be the key to then encoding of non regular ADTs.

> *A simpler example of over generalization is the type of the identity. $\forall\alpha.\,\alpha \to \alpha$ could be generalized as $\forall\varphi\ .\forall\alpha.\,\varphi\ \alpha \to \varphi\ \alpha$: it is easy to check that there are retyping functions (typable in System F^ω that are $\beta\eta$ convertible to the identity). By contrast type abstraction at higher-rank was a key for the typing of* distrib_pair.

Let us consider Okasaki's Seq non-regular ADT:

```
type  Seq α =
   | Nil  : ∀α. Seq α
   | Zero : ∀α. Seq (α×α) → Seq α
   | One  : ∀α. α → Seq (α×α) → Seq α
```

```
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
```

Figure 6.2: Leibnitz equality with GADT in OCaml

This may be encoded in System F^ω as:

$$
\begin{array}{rcl}
\mathsf{Seq} & \triangleq & \lambda\alpha.\,\forall F.\,F\alpha \to (F(\alpha\times\alpha) \to F\alpha) \to (\alpha \to F(\alpha\times\alpha) \to F\alpha) \to F\alpha \\
\mathsf{Nil} & \triangleq & \lambda n.\,\lambda z.\,\lambda s.\,n \\
 & & \Lambda\alpha.\Lambda F.\lambda n : F\alpha.\,\lambda z : F(\alpha\times\alpha) \to F\alpha.\,\lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\,n \\
\mathsf{Zero} & \triangleq & \lambda\ell.\,\lambda n.\,\lambda z.\,\lambda s.\,z\ (\ell\ n\ z\ s) \\
 & & \Lambda\alpha.\lambda\ell : \mathsf{Seq}\ \alpha.\,\Lambda F.\lambda n : F\alpha.\,\lambda z : F(\alpha\times\alpha) \to F\alpha.\,\lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\,z\ (\ell\ F\ n\ z\ s) \\
\mathsf{One} & \triangleq & \lambda x.\,\lambda\ell.\,\lambda n.\,\lambda z.\,\lambda s.\,s\ x\ (\ell\ n\ z\ s) \\
 & & \Lambda\alpha.\lambda x : \alpha. \\
 & & \qquad \lambda\ell : \mathsf{Seq}\ \alpha.\,\Lambda F.\lambda n : F\alpha.\,\lambda z : F(\alpha\times\alpha) \to F\alpha.\,\lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\,s\ x\ (\ell\ F\ n\ z\ s) \\
\mathsf{fold} & \triangleq & \lambda n.\,\lambda z.\,\lambda s.\,\lambda\ell.\,\ell\ n\ z\ s \\
 & & \Lambda\alpha.\Lambda F.\lambda n : F\alpha.\,\lambda z : F(\alpha\times\alpha) \to F\alpha.\,\lambda s : \alpha \to F(\alpha\times\alpha) \to F\alpha.\,\lambda\ell : \mathsf{Seq}\ \alpha.\,\ell\ F\ n\ z\ s
\end{array}
$$

Indeed, higher-rank is mandatory as for each constructor φ is applied to both α and $\alpha\times\alpha$. This is why non-regular ADTs cannot be encoded in System F.

6.3.5 Encoding GADT—with explicit coercions

We have seen that GADT can be encoded with a single equality type, existential types and non regular datatypes. Figure 6.2 gives an implementation of Leibnitz equality with a GADT in OCaml. We may then use a value of type (τ, σ) Eq.eq as a proof of equality of the types τ and σ.

Leibnitz equality can also be defined in System F^ω (Figure 6.3. In the figure, we have overlined proof terms and their types (respectively on the left and right columns) so as to help check typechecking.

We only implemented parts of the coercions of System Fc: we do not have decomposition of equalities (the inverse of Lift), as this requires injectivity of the type operator, which is not given.

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

Figure 6.3: Leibnitz equality in System F^ω

Equivalences and liftings must be written explicitly, while they are implicit with GADTs.

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

6.4 Beyond F^ω

Let us define the rank of a kind as usual: the base kind $*$ is of rank 1 and $\mathsf{rank}\;(\kappa_1\Rightarrow\kappa_2)$ is recursively defined as $\max(1+\mathsf{rank}\;\kappa_1,\mathsf{rank}\;\kappa_2)$. Hence, type functions of kind $*\Rightarrow*$ taking type parameters of base kind have rank 1 and type functions taking such type functions as arguments have rank 2.

We may define a hierarchy $F^1\subseteq F^2\subseteq F^3\ldots\subseteq F^\omega$ of type systems of increasing expressiveness, where F^n only uses kinds of rank n, whose whose limit is *System* F^ω—and where System F is just F^0 (ranks are sometimes shifted by one, starting with $F=F^2$).

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

Kind abstraction In section §**??**, we have used abstraction over kinds. Strictly speaking, this goes beyond System F^ω, but this all properties are preserved.

$$
\begin{array}{l}
L\forall(\varphi::*\Rightarrow*).\forall(\psi::*\Rightarrow*).\forall(\alpha_1::*).\forall(\alpha_2::*). \\
\quad(\forall\,(\alpha::*)\,.\,\varphi\;\alpha\to\psi\;\alpha)\to\varphi\;\alpha_1\to\varphi\;\alpha_2\to\psi\;\alpha_1\times\psi\;\alpha_2
\end{array}
$$

One applications is the use of constants instead of encodings as in section §**??**. Another application could be having even more general types. See for example, this discussion on distrib_pair (§6.3.1).

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

$$\begin{array}{ll} \mathsf{filled} & : * \Rightarrow \mathsf{field} \\ \mathsf{empty} & : \mathsf{field} \end{array} \qquad\qquad \mathsf{box} \ : \mathsf{field} \Rightarrow *$$

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

This allows to build values v of type box θ where θ of kind field statically tells whether v is filled with a value of type τ or empty. This is used in OCaml for rows of object types, although kinds are hidden from the user using superficial syntax:

```
let get (x : < get : 'a; .. >) : 'a = x#get
```

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

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 **?**.) 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 **?**.

Equirecursive kinds 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. Without further restrictions reduction of types does not terminate and type equality is ill-defined.

A solution proposed by Pottier 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. This extension has been used to show that references in System F can be translated away in System F^ω with guarded recursive kinds.

Encoding of functors In OCaml functions are by default generative: when a functor return abstract types, two applications of this functor to same structures will produce new incompatible abstract types. By contrast, applicative functors would return two structures with compatible abstract types.

While generative functors can be encoded in System F with existential types (as long as we ignore parametric types—or treat add as primitive). The idea to give functor F a type of the

form $\forall\alpha.\tau[\alpha] \to \exists\beta..\sigma[\alpha,\beta]$. Then, if X, Y has type $\tau[\rho]$, 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).

Indeed, the program

```
let Y = unpack FX in
let Z = unpack FX in
Y = Z
```

is ill-typed.

To allow two identical applications of the functor F to be compatible, a solution is to give F a type of the form: $\exists\varphi\,.\,\forall\alpha.\tau[\alpha] \to \sigma[\alpha,\varphi\ \alpha]$. Then, when F is applied it is first open and given type $\forall\alpha.\tau[\alpha] \to \sigma[\alpha,\psi\ \alpha]$ for some unknown ψ. Then the result of the application to an argument of type $\sigma[\rho]$ will have type $\sigma[\rho,\psi\ \rho]$ where the abstract type (application) $\psi\ \rho$ describes the abstract types created bye the application. Hence two applications to the same argument will have the same type, as the same abstract type ψ has just been open once and all occurrences of $\psi\ \rho$ are equal hence compatible.

The code would look like

```
let ψ , f = unpack F in
let Y = FX in
let Z = FX in
Y = Z
```

The encoding heavily relies on higher-rank types and may only be implemented in System F^ω. See **?** and **?** for details.

System F^ω in OCaml. Second-order polymorphism is not primitive but encodable in OCaml, using 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
```

or 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
```

Both solutions are quite verbose, though. Besides, second-order types are not first-class.

In principle, one can also reach higher-rank types OCaml, using first-class modules. However, this is not currently possible, due to (unnecessary) restrictions in the module language.

Modular explicits, a prototype extension[2], leaves some of these restrictions, easing abstraction over first-class modules and allow a light-weight encoding of System F^ω—with still some boiler-plate glue code. The encoding of distrib_pair with modular explicit is presented in Figure 6.4 with its two specialized instances.

[2]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

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)
```

Figure 6.4: distrib_pair with modular implicits

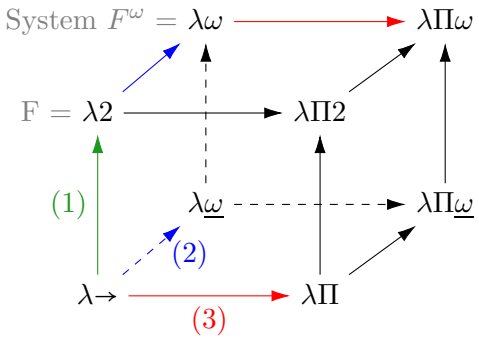
Higher-order polymorphism a la System F^ω is now also accessible in Scala-3. For instance, the monad example (with some variation on the signature) can be defined as:

```
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-$\sigma_2$scala-$\sigma_2$3.

Still, this feature of Scala-3 is not emphasized and was not directly accessible in previous versions of Scala. Besides, Scala's syntax and other complex features of Scala are obfuscating.

What's next? The next step in expressiveness are Dependent types, as illustrated in the Barendregt's λ-cube:



(1) Term abstraction on Types, as in System F;

(2) Type abstraction on Types, as in System F^{ω};

(3) Type abstraction on Terms: dependent types $\lambda\Pi$, $\lambda\Pi 2, \lambda\Pi\omega$.

A form of dependent types is available in Haskell, but not in OCaml.

Bibliography

▷ A tour of scala: Implicit parameters. Part of scala documentation.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, March 1996.

▷ Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2–3):81–116, December 1995.

▷ Amal Ahmed and Matthias Blume. Typed closure conversion preserves observational equivalence. In *ACM International Conference on Functional Programming (ICFP)*, pages 157–168, September 2008.

▷ Lennart Augustsson. Implementing Haskell overloading. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 65–73, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X.

▷ Nick Benton and Andrew Kennedy. Exceptional syntax journal of functional programming. *J. Funct. Program.*, 11(4):395–410, 2001.

▷ Jean-Philippe Bernardy, Patrik Jansson, and Koen Claessen. *Testing Polymorphic Properties*, pages 125–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11957-6. doi: 10.1007/978-$\sigma_2$3-$\sigma_2$642-$\sigma_2$11957-$\sigma_2$6_8.

▷ Richard Bird and Lambert Meertens. Nested datatypes. In *International Conference on Mathematics of Program Construction (MPC)*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998.

Nikolaj Skallerud Bjørner. Minimal typing derivations. In *In ACM SIGPLAN Workshop on ML and its Applications*, pages 120–126, 1994.

Daniel Bonniot. *Typage modulaire des multi-méthodes*. PhD thesis, École des Mines de Paris, November 2005.

▷ Daniel Bonniot. Type-checking multi-methods in ML (a modular approach). In *Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 2002.

▷ Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equality and subtyping. *Fundamenta Informaticæ*, 33:309–338, 1998.

▷ Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

Luca Cardelli. An implementation of f¦:. Technical report, DEC Systems Research Center, 1993.

Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science Series. Birkäuser, Boston, 1997.

▷ Henry Cejtin, Matthew Fluet, Suresh Jagannathan, and Stephen Weeks. The MLton compiler, 2007.

▷ Arthur Charguéraud and François Pottier. Functional translation of a calculus of capabilities. In *ACM International Conference on Functional Programming (ICFP)*, pages 213–224, September 2008.

▷ Juan Chen and David Tarditi. A simple typed intermediate language for object-oriented languages. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–49, January 2005.

▷ Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 54–65, June 2007.

▷ Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

Julien Crétin and Didier Rémy. Extending System F with Abstraction over Erasable Coercions. In *Proceedings of the 39th ACM Conference on Principles of Programming Languages*, January 2012.

▷ Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–70, New York, NY, USA, 2007. ACM. ISBN 1-59593-575-4.

Joshua Dunfield. Greedy bidirectional polymorphism. In *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 15–26, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-509-3. doi: http://doi.acm.org/10.1145/1596627.1596631.

▷ Ken-etsu Fujita and Aleksy Schubert. Existential type systems with no types in terms. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, pages 112–126, 2009. doi: 10.1007/978-$\sigma_2$3-$\sigma_2$642-$\sigma_2$02273-$\sigma_2$9_10.

Jun Furuse. Extensional polymorphism by flow graph dispatching. In Ohori (2003), pages 376–393. ISBN 3-540-20536-5.

▷ Jun Furuse. Extensional polymorphism by flow graph dispatching. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003b.

▷ Jacques Garrigue. Relaxing the value restriction. In *Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, pages 196–213. Springer, April 2004.

Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université Paris 7, June 1972.

▷ Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.

▷ Dan Grossman. Quantified types in an imperative language. *ACM Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

▷ Bob Harper and Mark Lillibridge. ML with callcc is unsound. Message to the TYPES mailing list, July 1991.

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

Robert Harper and Benjamin C. Pierce. Design considerations for ML-style module systems. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 8, pages 293–345. MIT Press, 2005.

▷ Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

▷ J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

▷ Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *ACM SIGPLAN Conference on History of Programming Languages*, June 2007.

Gérard Huet. *Résolution d'équations dans des langages d'ordre* 1, 2, ..., ω. PhD thesis, Université Paris 7, September 1976.

▷ John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

▷ Mark P. Jones. Simplifying and improving qualified types. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 160–169, New York, NY, USA, 1995a. ACM. ISBN 0-89791-719-7.

Mark P. Jones. Typing Haskell in Haskell. In *In Haskell Workshop*, 1999a.

Mark P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995b. ISBN 0-521-47253-9.

▷ Mark P. Jones. Typing Haskell in Haskell. In *Haskell workshop*, October 1999b.

▷ Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, 1997.

▷ Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(01):1, 2006.

Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *LFP '92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 193–204, New York, NY, USA, 1992. ACM. ISBN 0-89791-481-3. doi: http://doi.acm.org/10.1145/141471.141540.

▷ Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. ML typability is DEXPTIME-complete. In *Colloquium on Trees in Algebra and Programming*, volume 431 of *Lecture Notes in Computer Science*, pages 206–220. Springer, May 1990.

▷ Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, 1965.

▷ Konstantin Läufer and Martin Odersky. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems*, 16(5):1411–1430, September 1994.

▷ Didier Le Botlan and Didier Rémy. Recasting MLF. *Information and Computation*, 207(6):726–785, 2009. ISSN 0890-5401. doi: 10.1016/j.ic.2008.12.006.

▷ Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. PhD thesis, Université Paris 7, June 1992.

▷ Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 42–54, January 2006.

▷ Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, 2000. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/349214.349230.

▷ John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 47–57, January 1988.

▷ Harry G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 382–401, 1990.

▷ David McAllester. A logical algorithm for ML type inference. In *Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Computer Science*, pages 436–451. Springer, June 2003.

Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5.

▷ Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

▷ Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 271–283, January 1996.

▷ John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76 (2–3):211–249, 1988.

▷ John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

▷ Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 63–74, January 2009.

J. Garrett Morris and Mark P. Jones. Instance chains: type class programming without overlapping instances. In *ICFP '10: Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 375–386, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: http://doi.acm.org/10.1145/1863543.1863596.

▷ Greg Morrisett and Robert Harper. Typed closure conversion for recursively-defined functions (extended abstract). In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 10 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

▷ Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

▷ Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer, April 1984.

▷ 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.

▷ Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. Functional logic overloading. pages 233–244, 2002. doi: http://doi.acm.org/10.1145/565816.503294.

▷ Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7.

▷ Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

▷ Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 41–53, 2001.

Atsushi Ohori, editor. *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, volume 2895 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-20536-5.

▷ Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

▷ Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: a new foundation for generic programming. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 35–44, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. doi: 10.1145/2254064.2254070.

▷ Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. Online lecture notes, January 2009.

▷ Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Manuscript, April 2004.

▷ Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 71–84, January 1993.

Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62697.

▷ Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

▷ Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000.

▷ Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

▷ François Pottier. Notes du cours de DEA "Typage et Programmation", December 2002.

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.

François Pottier. Syntactic soundness proof of a type-and-capability system with hidden state. *Journal of Functional Programming*, 23(1):38–144, January 2013.

François Pottier. Hindley-Milner elaboration in applicative style. In *Proceedings of the 2014 ACM SIGPLAN International Conference on Functional Programming (ICFP'14)*, September 2014.

▷ François Pottier and Nadji Gauthier. Polymorphic typed defunctionalization and concretization. *Higher-Order and Symbolic Computation*, 19:125–162, March 2006.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. Submitted for publication, October 2012.

François Pottier and Jonathan Protzenko. Programming with permissions in Mezzo. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 173–184, September 2013.

▷ François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

▷ François Pottier and Didier Rémy. The essence of ML type inference. Draft of an extended version. Unpublished, September 2003.

▷ Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *Proceedings of the tenth International Conference on Functional Programming*, September 2005.

▷ Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 321–346. Springer, April 1994a.

▷ Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming: Types, Semantics and Language Design*. MIT Press, 1994b.

▷ Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

Didier Rémy and Boris Yakobowski. Efficient Type Inference for the MLF language: a graphical and constraints-based approach. In *The 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, pages 63–74, Victoria, BC, Canada, September 2008. doi: http://doi.acm.org/10.1145/1411203.1411216.

▷ John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, April 1974.

▷ John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing 83*, pages 513–523. Elsevier Science, 1983.

▷ John C. Reynolds. Three approaches to type structure. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*, volume 185 of *Lecture Notes in Computer Science*, pages 97–138. Springer, March 1985.

François Rouaix. Safe run-time overloading. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 355–366, 1990. doi: http://doi.acm.org/10.1145/96709.96746.

▷ Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. Cochis: Deterministic and coherent implicits. Technical report, KU Leuven, May 2017.

▷ Christian Skalka and François Pottier. Syntactic type soundness for HM(X). In *Workshop on Types in Programming (TIP)*, volume 75 of *Electronic Notes in Theoretical Computer Science*, July 2002.

Lau Skorstengaard. An Introduction to Logical Relations. *arXiv e-prints*, art. arXiv:1907.11133, July 2019.

Geoffrey S. Smith. Principal type schemes for functional programs with overloading and subtyping. In *Science of Computer Programming*, 1994.

Morten Heine Sørensen and Pawel Urzyczyn. *Studies in Logic and the Foundations of Mathematics*, chapter Lectures on the Curry-Howard Isomorphism. Elselvir Science Inc, 2006.

▷ Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In Sofiène Tahar, Otmame Ait-Mohamed, and César Muñoz, editors, *TPHOLs 2008: Theorem Proving in Higher Order Logics, 21th International Conference*, Lecture Notes in Computer Science. Springer, August 2008.

▷ Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

▷ Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1–2):11–49, April 2000.

▷ Peter J. Stuckey and Martin Sulzmann. A theory of overloading. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 167–178, New York, NY, USA, 2002. ACM. ISBN 1-58113-487-8.

▷ W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):pp. 198–212, 1967. ISSN 00224812.

▷ Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 11(2):245–296, 1994.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

▷ Jerzy Tiuryn and Pawel Urzyczyn. The subtyping problem for second-order types is undecidable. *Information and Computation*, 179(1):1–18, 2002.

▷ Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, September 2004.

▷ Andrew Tolmach and Dino P. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, July 1998.

▷ Philip Wadler. Theorems for free! In *Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 347–359, September 1989.

▷ Philip Wadler. The Girard-Reynolds isomorphism (second edition). *Theoretical Computer Science*, 375(1–3):201–226, May 2007.

▷ Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 60–76, January 1989.

Mitchell Wand. Corrigendum: Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, 1988.

▷ J. B. Wells. The essence of principal typings. In *International Colloquium on Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 913–925. Springer, 2002.

▷ J. B. Wells. The undecidability of Mitchell's subtyping relation. Technical Report 95-019, Computer Science Department, Boston University, December 1995.

▷ J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1–3):111–156, 1999.

▷ Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, December 1995.

▷ Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.