SW7 Programming Paradigms

Advanced Type Systems I : Polymorphism (Subtyping, Overloading, Parametricity)

based on lecture materials by Jean-Christophe Filliâtre

given at ENS Paris and École Polytechnique

Léon Gondelman

AALBORG UNIVERISITY | COPENHAGEN | 2025

on the other hand, we cannot type the program 1 2

$$\frac{\Gamma \vdash 1 : \tau' \to \tau \qquad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1\ 2 : \tau}$$

nor the program $\texttt{fun}\ x \to x\ x$

$$\frac{\dfrac{\Gamma + x : \tau_1 \vdash x : \tau_3 \to \tau_2 \qquad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x\ x : \tau_2}}{\Gamma \vdash \texttt{fun}\ x \to x\ x : \tau_1 \to \tau_2}$$

since $\tau_1 = \tau_1 \to \tau_2$ has no solution (types are finite)

we can show that

$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{int} \to \texttt{int}$$

but also

$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{bool} \to \texttt{bool}$$

beware : this is not polymorphism

for a given occurrence of $\texttt{fun}\ x \to x$ we must **choose** a type

thus, the term $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ is not typable

because there is no type τ such that

$$f : \tau \to \tau \vdash (f\ 1, f\ \texttt{true}) : \tau_1 \times \tau_2$$

on the other hand,

$$((\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x))\ 42$$

is typable (exercise : show it)

in particular, we cannot give **a single type** that is satisfactory for a primitive such as `fst` ; we would have to choose between

$$\texttt{int} \times \texttt{int} \to \texttt{int}$$
$$\texttt{int} \times \texttt{bool} \to \texttt{int}$$
$$\texttt{bool} \times \texttt{int} \to \texttt{bool}$$
$$(\texttt{int} \to \texttt{int}) \times \texttt{int} \to \texttt{int} \to \texttt{int}$$
etc.

but we can instead give **a typing rule**
for the application of `fst` :

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}\ e : \tau_1}$$

Parametric polymorphism

it is restrictive to give a unique type to $\texttt{fun}\ x \to x$ in

$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ \ldots$$

likewise, we would like to be able to give $\ll$ several types $\gg$ to primitives such as `fst` or `snd`

one solution : **parametric polymorphism**

we extend the algebra of types :

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \textit{base types} \\
& \mid & \tau \to \tau & \textit{type of a function} \\
& \mid & \tau \times \tau & \textit{type of a pair} \\
& \mid & \alpha & \textit{type variable} \\
& \mid & \forall \alpha . \tau & \textit{polymorphic type}
\end{array}
$$

we write $\mathcal{L}(\tau)$ for the set of **free** type variables in τ, defined by

$$\begin{aligned}
\mathcal{L}(\texttt{int}) &= \emptyset \\
\mathcal{L}(\alpha) &= \{\alpha\} \\
\mathcal{L}(\tau_1 \to \tau_2) &= \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau_1 \times \tau_2) &= \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\forall \alpha.\tau) &= \mathcal{L}(\tau) \setminus \{\alpha\}
\end{aligned}$$

for an environment Γ, we also define

$$\mathcal{L}(\Gamma) = \bigcup_{x \in dom(\Gamma)} \mathcal{L}(\Gamma(x))$$

we write $\tau[\alpha \leftarrow \tau']$ for the substitution of α by τ' in τ, defined by

$$
\begin{array}{rcl}
\texttt{int}[\alpha \leftarrow \tau'] & = & \texttt{int} \\
\alpha[\alpha \leftarrow \tau'] & = & \tau' \\
\beta[\alpha \leftarrow \tau'] & = & \beta \quad \text{if } \beta \neq \alpha \\
(\tau_1 \to \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \to \tau_2[\alpha \leftarrow \tau'] \\
(\tau_1 \times \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \times \tau_2[\alpha \leftarrow \tau'] \\
(\forall \alpha.\tau)[\alpha \leftarrow \tau'] & = & \forall \alpha.\tau \\
(\forall \beta.\tau)[\alpha \leftarrow \tau'] & = & \forall \beta.\tau[\alpha \leftarrow \tau'] \quad \text{if } \beta \neq \alpha
\end{array}
$$

the rules are **exactly** the same as before, plus

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\tau}$$

and

$$\frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

the resulting system is called **system F** (J.-Y. Girard / J. C. Reynolds)

$$\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \texttt{fun}\ x \to x : \alpha \to \alpha}}{\vdash \texttt{fun}\ x \to x : \forall \alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\ldots \vdash f : \forall \alpha.\alpha \to \alpha}{\ldots \vdash f : \texttt{int} \to \texttt{int}} \quad \vdots}{\ldots \vdash f\ 1 : \texttt{int}} \qquad \dfrac{\vdots}{\ldots \vdash f\ \texttt{true} : \texttt{bool}}}{f : \forall \alpha.\alpha \to \alpha \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

$$\vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}$$

we can now give a satisfactory type to primitives

$$\mathit{fst} : \quad \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$$

$$\mathit{snd} : \quad \forall\alpha.\forall\beta.\alpha \times \beta \to \beta$$

we can build a derivation of

$$\Gamma \vdash \texttt{fun}\ x \to x\ x : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$$

(exercise : do it)

the condition $\alpha \notin \mathcal{L}(\Gamma)$ in the rule

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\tau}$$

is crucial

without it, we would have

$$\frac{\dfrac{\dfrac{\Gamma + x : \alpha \vdash x : \alpha}{\Gamma + x : \alpha \vdash x : \forall \alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \alpha \to \forall \alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \forall \alpha.\alpha \to \forall \alpha.\alpha}$$

and we would in particular accept the program

$$(\texttt{fun}\ x \to x)\ 1\ 2$$

bad news

for terms without annotations, the two problems

- **inference** : given e, does there exist τ such that $\vdash e : \tau$?
- **checking** : given e and τ, do we have $\vdash e : \tau$?

are not decidable

J. B. Wells. *Typability and type checking in the second-order lambda-calculus are equivalent and undecidable*, 1994.

to obtain a decidable type inference, we are going to restrict the power of system F

$\Rightarrow$ the **Hindley–Milner** system, used in OCaml, SML, Haskell, etc.

we restrict the $\forall$ quantification to the head of types (prenex quantification)

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \textit{base types} \\
& \mid & \tau \to \tau & \textit{type of a function} \\
& \mid & \tau \times \tau & \textit{type of a pair} \\
& \mid & \alpha & \textit{type variable} \\
\\
\sigma & ::= & \tau & \textit{schemes} \\
& \mid & \forall \alpha . \sigma &
\end{array}
$$

the environment Γ associates a type scheme with each identifier and the typing relation now has the form $\Gamma \vdash e : \sigma$

in Hindley–Milner, the following types are still accepted

$$\forall\alpha.\alpha \to \alpha$$
$$\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$$
$$\forall\alpha.\texttt{bool} \times \alpha \times \alpha \to \alpha$$
$$\forall\alpha.(\alpha \to \alpha) \to \alpha$$

but not types such as

$$(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$$

note : in OCaml syntax, prenex quantification is implicit

```
# fst;;
```

```
- : 'a * 'b -> 'a = <fun>
```

$$\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$$

```
# List.fold_left;;
```

```
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

$$\forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \texttt{list} \to \alpha$$

$$\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\cdots \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \to \texttt{int}}\cdots$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma} \qquad \frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

we note that only the `let` construct makes it possible to introduce a polymorphic type into the environment

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma_2}$$

in particular, we can always type

$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

with $f : \forall\alpha.\alpha \to \alpha$ in the context in order to type $(f\ 1, f\ \texttt{true})$

on the other hand, the typing rule

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2}$$

does not introduce a polymorphic type (otherwise $\tau_1 \to \tau_2$ would be ill-formed)

in particular, we can no longer type

$$\texttt{fun}\ x \to x\ x$$

to program type checking or type inference, we proceed by induction on the structure of the program

but for a given expression, three rules can apply : the rule of the monomorphic system, the rule

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \sigma}$$

or else the rule

$$\frac{\Gamma \vdash e : \forall \alpha . \sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

how do we choose ? will we have to proceed by trial and error ?

syntax-directed Hindley–Milner system

we are going to modify the presentation of the Hindley–Milner system so that it is **syntax-directed**, *i.e.*, for every expression, at most one rule applies

the rules have the same expressive power : every closed term is typable in one system if and only if it is typable in the other

syntax-directed Hindley–Milner system

$$\frac{x \in dom(\Gamma) \quad \tau \leq \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\cdots \qquad \frac{\tau \leq type(op)}{\Gamma \vdash op : \tau}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : Gen(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

two operations appear

- **instantiation**, in the rule

$$\frac{x \in dom(\Gamma) \quad \tau \leq \Gamma(x)}{\Gamma \vdash x : \tau}$$

the relation $\tau \leq \sigma$ is read $\ll \tau$ is an instance of $\sigma \gg$ and is defined by

$$\tau \leq \forall\alpha_1 ... \alpha_n.\tau' \quad \text{iff} \quad \exists\tau_1 ... \exists\tau_n.\ \tau = \tau'[\alpha_1 \leftarrow \tau_1, ..., \alpha_n \leftarrow \tau_n]$$

example : $\texttt{int} \times \texttt{bool} \rightarrow \texttt{int} \leq \forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha$

- and **generalization**, in the rule

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + x : \mathit{Gen}(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

  where

$$\mathit{Gen}(\tau_1, \Gamma) \stackrel{\text{def}}{=} \forall\alpha_1 ... \forall\alpha_n.\tau_1 \quad \text{where} \quad \{\alpha_1, ..., \alpha_n\} = \mathcal{L}(\tau_1) \backslash \mathcal{L}(\Gamma)$$

$$\dfrac{\dfrac{\dfrac{\alpha \leq \alpha}{x : \alpha \vdash x : \alpha}}{\emptyset \vdash \texttt{fun}\ x \to x : \alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{\texttt{int}\to\texttt{int} \leq \forall\alpha.\alpha\to\alpha}{\Gamma \vdash f : \texttt{int}\to\texttt{int}} \quad \vdots}{\Gamma \vdash f\ 1 : \texttt{int}} \qquad \dfrac{\dfrac{\texttt{bool}\to\texttt{bool} \leq \forall\alpha.\alpha\to\alpha}{\Gamma \vdash f : \texttt{bool}\to\texttt{bool}} \quad \vdots}{\Gamma \vdash f\ \texttt{true} : \texttt{bool}}}{\Gamma \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}$$

with $\Gamma \stackrel{\text{def}}{=} \emptyset + f : \mathit{Gen}(\alpha \to \alpha, \emptyset)$
$= f : \forall\alpha.\alpha \to \alpha$

to infer the type of an expression, problems remain

- in `fun` $x \to e$, what type should we give to x ?
- for a variable x, which instance of $\Gamma(x)$ should we choose ?

there is a solution : **algorithm W** (Milner, Damas)

two ideas

- we use **fresh type variables** to represent unknown types
  - for the type of x in `fun` $x \to e$
  - to instantiate the variables of the scheme $\Gamma(x)$

- we determine the values of these variables later, by **unification of types** at the time of typing an application

let τ_1 and τ_2 be two types containing type variables $\alpha_1, \ldots, \alpha_n$

does there exist an instantiation f of the variables α_i such that $f(\tau_1) = f(\tau_2)$?

this is called **unification**

example 1 :

$$\tau_1 = \alpha \times \beta \to \texttt{int}$$
$$\tau_2 = \texttt{int} \times \texttt{bool} \to \gamma$$
$$\text{solution :} \quad \alpha = \texttt{int} \ \wedge \ \beta = \texttt{bool} \ \wedge \ \gamma = \texttt{int}$$

example 2 :

$$\tau_1 = \alpha \times \texttt{int} \to \alpha \times \texttt{int}$$
$$\tau_2 = \gamma \to \gamma$$
$$\text{solution :} \quad \gamma = \alpha \times \texttt{int}$$

example 3 :

$$\tau_1 = \alpha \to \texttt{int}$$
$$\tau_2 = \beta \times \gamma$$
no solution

example 4 :

$$\tau_1 = \alpha \to \texttt{int}$$
$$\tau_2 = \alpha$$
no solution

unifier(τ_1, τ_2) determines whether there exists an instance of the type variables of τ_1 and τ_2 such that $\tau_1 = \tau_2$

$$\textit{unifier}(\tau, \tau) = \text{success}$$

$$\textit{unifier}(\tau_1 \to \tau_1', \tau_2 \to \tau_2') = \textit{unifier}(\tau_1, \tau_2) \ ; \ \textit{unifier}(\tau_1', \tau_2')$$

$$\textit{unifier}(\tau_1 \times \tau_1', \tau_2 \times \tau_2') = \textit{unifier}(\tau_1, \tau_2) \ ; \ \textit{unifier}(\tau_1', \tau_2')$$

$$\textit{unifier}(\alpha, \tau) = \begin{array}{l} \text{if } \alpha \notin \mathcal{L}(\tau), \text{ replace } \alpha \text{ by } \tau \text{ everywhere} \\ \text{otherwise, failure} \end{array}$$

$$\textit{unifier}(\tau, \alpha) = \textit{unifier}(\alpha, \tau)$$

$$\textit{unifier}(\tau_1, \tau_2) = \text{failure in all other cases}$$

(the lab session is there for you to implement the algorithm W in OCaml)

idea of algorithm W on an example

let us consider the expression $\boxed{\texttt{fun}\ x \to +(\mathit{fst}\ x, 1)}$

– we give x the type α_1, a fresh type variable

– the primitive $+$ has type $\texttt{int} \times \texttt{int} \to \texttt{int}$

– let us type the expression $(\mathit{fst}\ x, 1)$

  – *fst* has the type scheme $\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$

  – we therefore give it the type $\alpha_2 \times \beta_1 \to \alpha_2$ for two new variables

  – the application *fst* x forces us to unify α_1 and $\alpha_2 \times \beta_1$, $\Rightarrow$ $\alpha_1 = \alpha_2 \times \beta_1$

– $(\mathit{fst}\ x, 1)$ therefore has type $\alpha_2 \times \texttt{int}$

– the application $+(\mathit{fst}\ x, 1)$ unifies $\texttt{int} \times \texttt{int}$ and $\alpha_2 \times \texttt{int}$, $\Rightarrow \alpha_2 = \texttt{int}$

in the end, we obtain the type $\texttt{int} \times \beta_1 \to \texttt{int}$,
and if we generalize it (in a `let`) we obtain $\boxed{\forall\beta.\texttt{int} \times \beta \to \texttt{int}}$

we define a function W that takes as arguments an environment Γ and an expression e and returns the inferred type for e

- if e is a variable x,
  return a trivial instance of $\Gamma(x)$
- if e is a constant c,
  return a trivial instance of its type (think of [] : α `list`)
- if e is a primitive op,
  return a trivial instance of its type
- if e is a pair (e_1, e_2),
  compute $\tau_1 = W(\Gamma, e_1)$
  compute $\tau_2 = W(\Gamma, e_2)$
  return $\tau_1 \times \tau_2$

- if e is a function $\texttt{fun}\ x \to e_1$,
  let α be a fresh variable
  compute $\tau = W(\Gamma + x : \alpha, e_1)$
  return $\alpha \to \tau$

- if e is an application $e_1\ e_2$,
  compute $\tau_1 = W(\Gamma, e_1)$
  compute $\tau_2 = W(\Gamma, e_2)$
  let α be a fresh variable
  unifier$(\tau_1, \tau_2 \to \alpha)$
  return α

- if e is $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$,
  compute $\tau_1 = W(\Gamma, e_1)$
  return $W(\Gamma + x : \mathit{Gen}(\tau_1, \Gamma), e_2)$

Théorème (Damas, Milner, 1982)

Algorithm W is correct, complete and determines the principal type :

- *if* $W(\emptyset, e) = \tau$ *then* $\emptyset \vdash e : \tau$
- *if* $\emptyset \vdash e : \tau$ *then* $\tau \leq \forall \bar{\alpha}.W(\emptyset, e)$

Théorème (type safety)

The Hindley–Milner system is safe.
(If $\emptyset \vdash e : \tau$*, then the reduction of* e *is infinite*
or terminates on a value.)

algorithmic considerations

there are several ways to implement unification

- by explicitly manipulating a **substitution**

```
type tvar = int
type subst = typ TVmap.t
```

- by using **destructive** type variables

```
type tvar = { id: int; mutable def: typ option; }
```

there are also several ways to code algorithm W

- with **explicit schemes** and by computing $Gen(\tau, \Gamma)$

```
type schema = { tvars: TVset.t; typ: typ; }
```

- with **levels**

$$\frac{\Gamma \vdash_{n+1} e_1 : \tau_1 \qquad \Gamma + x : (\tau_1, n) \vdash_n e_2 : \tau_2}{\Gamma \vdash_n \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

see tutorial

we can extend mini-ML in many ways

- constructed types (n-tuples, lists, sum and product types)
- references (see the slides below)
- recursion (not shown in this slides)

we have already seen pairs

lists pose no difficulty

$$\begin{aligned} \texttt{[]} &: \quad \forall\alpha.\, \alpha\ \texttt{list} \\ \texttt{::} &: \quad \forall\alpha.\, \alpha \times \alpha\ \texttt{list} \to \alpha\ \texttt{list} \end{aligned}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{list} \qquad \Gamma \vdash e_2 : \tau_1 \qquad \Gamma + x : \tau + y : \tau\ \texttt{list} \vdash e_3 : \tau_1}{\Gamma \vdash \texttt{match}\ e_1\ \texttt{with}\ \texttt{[]} \to e_2 \mid \texttt{::}(x, y) \to e_3 : \tau_1}$$

this generalizes easily to sum and product types

for references, we might naively think that it is enough to add the primitives

$$\begin{aligned} \texttt{ref} &: \quad \forall\alpha.\, \alpha \to \alpha\ \texttt{ref} \\ \texttt{!} &: \quad \forall\alpha.\, \alpha\ \texttt{ref} \to \alpha \\ \texttt{:=} &: \quad \forall\alpha.\, \alpha\ \texttt{ref} \to \alpha \to \texttt{unit} \end{aligned}$$

unfortunately, this is incorrect !

$$\texttt{let } r = \texttt{ref } (\texttt{fun } x \to x) \texttt{ in} \qquad r : \forall\alpha.(\alpha \to \alpha)\ \texttt{ref}$$
$$\texttt{let } _ = r \texttt{ := } (\texttt{fun } x \to x\ 1) \texttt{ in}$$
$$!r\ \texttt{true} \qquad \textit{boom}!$$

this is the problem of **polymorphic references**

to avoid this problem, there is an extremely simple solution, namely a syntactic restriction on the `let` construct

Définition (*value restriction*, Wright 1995)

A program satisfies the **value restriction** *criterion if every* `let` *subexpression whose type is generalized is of the form*

$$\texttt{let } x = v_1 \texttt{ in } e_2$$

where v_1 *is a* **value**.

in practice, we continue to write

$$\texttt{let } r = \texttt{ref } (\texttt{fun } x \to x) \texttt{ in } \ldots$$

but the type of r is not generalized

as if we had written

$$(\texttt{fun } r \to \ldots)\ (\texttt{ref } (\texttt{fun } x \to x))$$

in OCaml, a nongeneralized variable has the form '_a

```
# let x = ref (fun x -> x);;

val x : ('_a -> '_a) ref
```

the *value restriction* is also slightly relaxed to allow safe expressions such as applications of constructors

```
# let l = [fun x -> x];;

val l : ('a -> 'a) list = [<fun>]
```

a few small annoyances remain

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let f = id id;;
val f : '_a -> '_a = <fun>
# f 1;;
- : int = 1
# f true;;
This expression has type bool but is here used with type int
# f;;
- : int -> int = <fun>
```

the solution : expand so as to make a function appear, *i.e.*, a value

```
# let f x = id id x;;

val f : 'a -> 'a = <fun>
```

(this is called η-expansion)

polymorphic references

in the presence of the module system, reality is even more complex

given a module `M`

```
module M : sig
  type 'a t
  val create : int -> 'a t
end
```

am I allowed to generalize the type of `M.create 17` ?

the answer depends on the nature of the type `'a t` : no for an array, yes for a list, etc.

in OCaml, a **variance** indication makes it possible to distinguish the two

```
type +'a t   (* we can generalize *)
type  'a u   (* we cannot *)
```

read *Relaxing the value restriction*, J. Garrigue, 2004

two books related to this course

- Benjamin C. Pierce, *Types and Programming Languages*
  (also related to lecture 2)

- Xavier Leroy and Pierre Weis, *Le langage Caml*
  (the last chapter explains inference with levels)