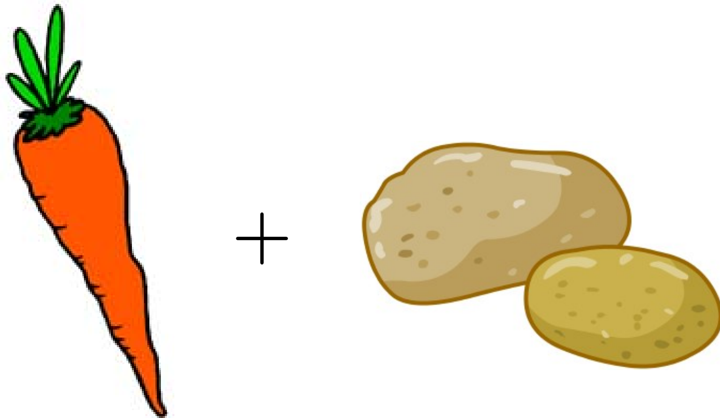
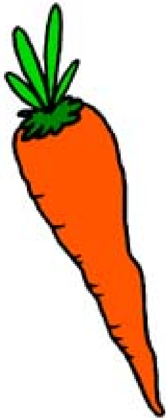
Universidade da Beira Interior

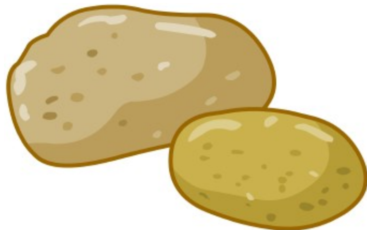
Desenho de Linguagens de Programação e de Compiladores

Simão Melo de Sousa

Aula 3 - Tipagem, sistemas de tipos e algoritmos







se avalio a expressão

```
"5" + 37
```

devo obter

- um erro durante a compilação ? (OCaml)
- um erro durante a execução ? (Python)
- o inteiro 42 ? (Visual Basic, PHP)
- a string "537" ? (Java)
- ou ainda outra coisa ?

e que tal para

```
37 / "5"
```

?

WAT?? (link)

e se adicionar duas expressões quaisquer

```
e1 + e2
```

como determinar se isto é “legal” e o que se deve fazer conforme o caso ?

a resposta é a **tipagem**, uma análise que associa um **tipo** a cada sub-expressão, com o objectivo de recusar programas sem coerência

certas linguagens de programação são qualificadas de **tipadas/tipificadas dinamicamente**, isto é, durante a execução do programa

exemplos : Lisp, PHP, Python

outros são **tipificadas estaticamente**, isto é durante a compilação do programa

exemplos : C, Java, OCaml

é o segundo caso que exploraremos nesta aula

slogan (R. Milner)

well typed programs do not go wrong

- a tipagem deve ser **decidível**
- a tipagem deve rejeitar programas absurdos como 1 2, cuja avaliação iria fracassar ; é a **segurança da tipagem** (*type safety*)
- a tipagem não deve rejeitar demasiados programas não-absurdos, *i.e.* o sistema de tipos deve ser **expressivo**

1. todas as sub-expressões sofrem anotações de tipo

   $\texttt{fun}\,(x : \texttt{int}) \to \texttt{let}\,(y : \texttt{int}) = (+ :)(((x : \texttt{int}), (1 : \texttt{int})) : \texttt{int}{\times}\texttt{int})\ \texttt{in}\ y$

   de verificação fácil, mas muito tedioso para o programador

2. anotar somente as declarações de variáveis (Pascal, C, Java, etc.)

   $$\texttt{fun}\ (x : \texttt{int}) \to \texttt{let}\ (y : \texttt{int}) = +(x, 1)\ \texttt{in}\ y$$

3. anotar somente os parâmetros das funções

   $$\texttt{fun}\ (x : \texttt{int}) \to \texttt{let}\ y = +(x, 1)\ \texttt{in}\ y$$

4. não obrigar anotações de tipo $\Rightarrow$ **inferência** completa (OCaml, Haskell, etc.)

um algoritmo de tipagem deve ter as propriedades seguintes

- **correção** : se o algoritmo responde "sim" então o programa está efectivamente bem tipado

- **completude** : se o programa está bem tipado então o algoritmo deve responder "sim"

e eventualmente

- **principalidade** : o tipo calculado para uma dada expressão é o mais geral possivel dos tipos *candidatos* (os tipos possíveis para uma dada situação)

consideremos a tipagem de mini-ML

1. tipagem monomórfica
2. tipagem polimórfica, inferência de tipos

lembrete

$$
\begin{array}{rcll}
e & ::= & x & \text{identificador} \\
  & | & c & \text{constante } (1,\ 2,\ \ldots,\ \mathit{true},\ \ldots) \\
  & | & \mathit{op} & \text{primitiva } (+,\ \times,\ \mathit{fst},\ \ldots) \\
  & | & \texttt{fun}\ x \rightarrow e & \text{função} \\
  & | & e\ e & \text{applicação} \\
  & | & (e, e) & \text{par} \\
  & | & \texttt{let}\ x = e\ \texttt{in}\ e & \text{ligação/declaração local}
\end{array}
$$

damo-nos **tipos simples**, cuja sintaxe abstracta é

$$\begin{array}{rcll} \tau & ::= & \texttt{int} \mid \texttt{bool} \mid \dots & \textit{tipos de base} \\ & \mid & \tau \rightarrow \tau & \textit{tipo de uma função} \\ & \mid & \tau \times \tau & \textit{tipo de um par} \end{array}$$

o **juizo** de tipagem que vamos definir tem por notação

$$\Gamma \vdash e : \tau$$

e lê-se « no ambiente/contexto Γ, a expressão e tem por tipo τ »

o ambiente Γ associa um tipo $\Gamma(x)$ a toda a variável x *livre* em e

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

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

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

$\Gamma + x : \tau$ é o ambiente Γ' definido por $\Gamma'(x) = \tau$ e $\Gamma'(y) = \Gamma(y)$ senão

$$\dfrac{\dfrac{\dfrac{\vdots \qquad \dfrac{\vdots}{x : \texttt{int} \vdash (x,1) : \texttt{int} \times \texttt{int}}}{x : \texttt{int} \vdash +(x,1) : \texttt{int}}}{\emptyset \vdash \texttt{fun}\ x \rightarrow +(x,1) : \texttt{int} \rightarrow \texttt{int}} \qquad \dfrac{\dfrac{}{... \vdash f : \texttt{int} \rightarrow \texttt{int}} \quad \dfrac{}{... \vdash 2 : \texttt{int}}}{f : \texttt{int} \rightarrow \texttt{int} \vdash f\ 2 : \texttt{int}}}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \rightarrow +(x,1)\ \texttt{in}\ f\ 2 : \texttt{int}}$$

Por outro lado, não podemos tipar o programa 1 2

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

nem o programa $\texttt{fun}\ x \rightarrow x\ x$

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

porque $\tau_1 = \tau_1 \rightarrow \tau_2$ não tem soluções (no contexto em que os tipos são construções finitas)

podemos mostrar

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

mas também

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

attenção : isto não é polimorfismo

para uma ocorrência particular de $\texttt{fun } x \rightarrow x$ é preciso **escolher** um tipo

assim, o termo $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ não é tipável

porque não há tipo τ tal que

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

no entanto,

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

é tipável (exercício : demonstrá-lo)

em particular, não podemos dar um tipo satisfatório para uma primitiva como `fst` ; para tal deveríamos ter a capacidade em escolher entre

$$\begin{array}{l} \texttt{int} \times \texttt{int} \rightarrow \texttt{int} \\ \texttt{int} \times \texttt{bool} \rightarrow \texttt{int} \\ \texttt{bool} \times \texttt{int} \rightarrow \texttt{bool} \\ (\texttt{int} \rightarrow \texttt{int}) \times \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int} \\ \text{etc.} \end{array}$$

de igual modo para as primitivas *opif* e *opfix*

não podemos tipar de forma geral *opfix* mas podemos dar uma regra para uma construção `let rec` que seria primitiva

$$\frac{\Gamma + x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

e se pretendemos excluir os *valores* recursivos, podemos alterar a regra da seguinte forma

$$\frac{\Gamma + (f : \tau \rightarrow \tau_1) + (x : \tau) \vdash e_1 : \tau_1 \qquad \Gamma + (f : \tau \rightarrow \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

diferença entre regra de tipagem e algoritmo de tipagem

quando temos um tipo `fun` $x \to e$, como encontrar o tipo para x ?

é toda a diferença entre as **regras de tipagem**, que definam a relação ternária

$$\Gamma \vdash e : \tau$$

e o **algoritmo de tipagem** que verifica que uma certa expressão e é bem tipável num certo ambiente Γ

consideremos a abordagem em que só os parâmetros das funções comportam anotações de tipo.

programemo-la em OCaml

sintaxe abstracta dos tipos

```
type typ =
  | Tint
  | Tarrow   of typ * typ
  | Tproduct of typ * typ
```

o constructor `Fun` toma como argumento suplementar o tipo do parâmetro

```
type expression =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * typ * expression (* única alteração *)
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

o ambiente Γ é implementado por uma estrutura persistente

utilizemos aqui o módulo `Map` de OCaml

```
module Smap = Map.Make(String)

type env = typ Smap.t
```

(performance : árvore equilibradas $\Rightarrow$ inserção e pesquisa em $O(\log n)$)

```
let rec type_expr env = function
  | Const _ -> Tint
  | Var x -> Smap.find x env
  | Op "+" -> Tarrow (Tproduct (Tint, Tint), Tint)
  | Pair (e1, e2) ->
      Tproduct (type_expr env e1, type_expr env e2)
```

para a função, o tipo da variável é dado

```
  | Fun (x, ty, e) ->
      Tarrow (ty, type_expr (Smap.add x ty env) e)
```

para a variável, o tipo é calculado

```
  | Let (x, e1, e2) ->
      type_expr (Smap.add x (type_expr env e1) env) e2
```

(note o interesse da **persistência** de `env`)

as únicas verificações encontram-se na aplicação

```
  | App (e1, e2) -> begin match type_expr env e1 with
      | Tarrow (ty2, ty) ->
          if type_expr env e2 = ty2 then ty
          else failwith "error : bad argument type"
      | _ ->
          failwith "error : function expected"
    end
```

exemplos

```
# type_expr
    (Let ("f",
      Fun ("x", Tint, App (Op "+", Pair (Var "x", Const 1))),
      App (Var "f", Const 2)));;
```

```
- : typ = Tint
```

```
# type_expr (Fun ("x", Tint, App (Var "x", Var "x")));;
```

```
Exception: Failure "error : function expected".
```

```
# type_expr (App (App (Op "+", Const 1), Const 2));;
```

```
Exception: Failure "error : bad argument type".
```

- não fazemos

```
failwith "type error"
```

mas sim, indicamos a origem do problema com a maior precisão possível

- mantemos os tipos para as próximas fases do compilador

primeiro, enfeitamos as árvores **na entrada** da tipagem com uma localização para o ficheiro fonte

```
type loc = ...
```

```
type expression =






  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * typ * expression
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

primeiro, enfeitamos as árvores **na entrada** da tipagem com uma localização para o ficheiro fonte

```
type loc = ...
```

```
type expression = {
  desc: desc;
  loc : loc;
}
and desc =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * typ * expression
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

declaramos uma excepção da forma

```
exception Error of loc * string
```

esta é lançada da seguinte forma

```
let rec type_expr env e = match e.desc with
  | ...
  | App (e1, e2) -> begin match type_expr env e1 with
    | Tarrow (ty2, ty) ->
        if type_expr env e2 <> ty2 then
          raise (Error (e2.loc, "bad argument type"));
        ...
```

e esta é recuperada da seguinte forma, por exemplo no programa principal

```
try
  let p = Parser.parse file in
  let t = Typing.program p in
  ...
with Error (loc, msg) ->
  Format.eprintf "File '%s', line ...\n" file loc;
  Format.eprintf "error: %s@." msg;
  exit 1
```

de outra lado, decoramos as árvores **na saída** da tipagem com os tipos calculados

```
type texpression = {
  tdesc: tdesc;
  typ  : typ;
}
and tdesc =
  | Tvar   of string
  | Tconst of int
  | Top    of string
  | Tfun   of string * typ * texpression
  | Tapp   of texpression * texpression
  | Tpair  of texpression * texpression
  | Tlet   of string * texpression * texpression
```

é um **novo tipo** para as expressões

a função de tipagem tem então por tipo

```
val type_expr: expression -> texpression
```

a função de tipagem **reconstruí** árvores, desta vez tipadas

```
let rec type_expr env e =
  let d, ty = compute_type env e in
  { tdesc = d; typ = ty }

and compute_type env e = match e.desc with
  | Const n ->
      Tconst n, Tint
  | Var x ->
      Tvar x, Smap.find x env
  | Pair (e1, e2) ->
      let te1 = type_expr env e1 in
      let te2 = type_expr env e2 in
      Tpair (te1, te2), Tproduct (te1.typ, te2.typ)
  | ...
```

type safety:

well typed programs do not go wrong

mostramos a correcção da tipagem **relativamente** à semântica por redução (operacional *small-step*)

Theorema (segurança da tipagem, como uma propriedade de correcção)

Se $\emptyset \vdash e : \tau$, *então a redução de e é infinita ou então termina num valor.*

ou, de forma equivalente, *se termina então devolve um valor* :

Theorema

Se $\emptyset \vdash e : \tau$ *e* $e \stackrel{\star}{\rightarrow} e'$ *e* e' *iredutível, então* e' *é um valor.*

a prova deste teorema apoia-se sobre dois lemas

Lema (progressão)

Se $\emptyset \vdash e : \tau$, *então ou* e *é um valor ou existe um* e' *tal que* $e \rightarrow e'$.

Lema (preservação)

Se $\emptyset \vdash e : \tau$ *e* $e \rightarrow e'$ *então* $\emptyset \vdash e' : \tau$.

Lema (progressão)

Se $\emptyset \vdash e : \tau$, então ou e é um valor ou existe um e' tal que $e \rightarrow e'$.

Demonstração : por recorrência sobre a derivação $\emptyset \vdash e : \tau$

supomos por exemplo $e = e_1\ e_2$; temos então

$$\frac{\emptyset \vdash e_1 : \tau' \rightarrow \tau \quad \emptyset \vdash e_2 : \tau'}{\emptyset \vdash e_1\ e_2 : \tau}$$

aplicamos a HI sobre e_1

- se $e_1 \rightarrow e_1'$, então $e_1\ e_2 \rightarrow e_1'\ e_2$ (ver lema de passagem para o contexto)
- se e_1 é um valor, supomos $e_1 = \texttt{fun}\ x \rightarrow e_3$ (temos também + etc.) aplicamos a HI sobre e_2
  - se $e_2 \rightarrow e_2'$, então $e_1\ e_2 \rightarrow e_1\ e_2'$ (mesmo lema)
  - se e_2 é um valor, então $e_1\ e_2 \rightarrow e_3[x \leftarrow e_2]$

(exercício : tratar dos outros casos) □

comecemos por pequenos lemas fáceis

Lema (permutação)

Se $\Gamma + x : \tau_1 + y : \tau_2 \vdash e : \tau$ *e* $x \neq y$ *então* $\Gamma + y : \tau_2 + x : \tau_1 \vdash e : \tau$
(e a derivação tem a mesma altura).

Demonstração: por indução, trivial

Lema (enfraquecimento)

Se $\Gamma \vdash e : \tau$ *e* $x \notin dom(\Gamma)$, *então* $\Gamma + x : \tau' \vdash e : \tau$
(e a derivação tem a mesma altura).

Demonstração: por indução, trivial

continuemos por um **lema chave**

Lema (preservação por substituição)

Se $\Gamma + x : \tau' \vdash e : \tau$ *e* $\Gamma \vdash e' : \tau'$ *então* $\Gamma \vdash e[x \leftarrow e'] : \tau$.

demonstração : por indução sobre a derivação $\Gamma + x : \tau' \vdash e : \tau$

- caso de uma variável $e = y$
  - se $x = y$ então $e[x \leftarrow e'] = e'$ e $\tau = \tau'$
  - se $x \neq y$ então $e[x \leftarrow e'] = e$ e $\tau = \Gamma(y)$
- caso de uma abstração $e = \texttt{fun } y \rightarrow e_1$
  podemos supor que $y \neq x$, $y \notin dom(\Gamma)$ e y não livre em e' (α-conversão)
  temos $\Gamma + x : \tau' + y : \tau_2 \vdash e_1 : \tau_1$ e logo $\Gamma + y : \tau_2 + x : \tau' \vdash e_1 : \tau_1$
  (permutação) ; por outro lado $\Gamma \vdash e' : \tau'$ e por consequência
  $\Gamma + y : \tau_2 \vdash e' : \tau'$ (enfraquecimento)
  por HI temos então $\Gamma + y : \tau_2 \vdash e_1[x \leftarrow e'] : \tau_1$
  e logo $\Gamma \vdash (\texttt{fun } y \rightarrow e_1)[x \leftarrow e'] : \tau_2 \rightarrow \tau_1$, *i.e.* $\Gamma \vdash e[x \leftarrow e'] : \tau$

(exercício : tratar dos outros casos) □

podemos finalmente mostrar

Lema (preservação)

Se $\emptyset \vdash e : \tau$ *e* $e \rightarrow e'$ *então* $\emptyset \vdash e' : \tau$*.*

demonstração : por indução sobre a derivação $\emptyset \vdash e : \tau$

- caso $e = \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$

$$\frac{\emptyset \vdash e_1 : \tau_1 \qquad x : \tau_1 \vdash e_2 : \tau_2}{\emptyset \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}$$

  - se $e_1 \rightarrow e_1'$, por HI temos $\emptyset \vdash e_1' : \tau_1$ e logo $\emptyset \vdash \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2 : \tau_2$
  - se e_1 é um valor e $e' = e_2[x \leftarrow e_1]$ então aplicamos o lema de preservação por substituição
- caso $e = e_1\ e_2$
  - se $e_1 \rightarrow e_1'$ ou se e_1 valor e $e_2 \rightarrow e_2'$ então utilizamos a HI
  - se $e_1 = \mathtt{fun}\ x \rightarrow e_3$ e e_2 valor então $e' = e_3[x \leftarrow e_2]$ e aplicamos também aqui o lema de substituição $\square$

Podemos então demonstrar facilmente o teorema

Theorema (segurança da tipagem)

Se $\emptyset \vdash e : \tau$ *e* $e \xrightarrow{\star} e'$ *e* e' *iredutível, então* e' *é um valor.*

demonstração : temos $e \rightarrow e_1 \rightarrow \cdots \rightarrow e'$ e por aplicações sucessivas do lema de preservação, temos então $\emptyset \vdash e' : \tau$
pelo lema de progressão, e' se reduz ou é um valor.
é por isso um valor □

é restritivo atribuir um tipo único a `fun` $x \rightarrow x$ em

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

de igual forma, gostaríamos de poder dar « vários tipos » às primitivas tais que `fst` ou `snd`

uma solução : o **polimorfismo paramétrico**

estendemos a álgebra dos tipos :

τ	$::=$	$\texttt{int} \mid \texttt{bool} \mid \ldots$	*tipos de base*
	$\mid$	$\tau \rightarrow \tau$	*tipo de uma função*
	$\mid$	$\tau \times \tau$	*tipo de um par*
	$\mid$	α	*variável de tipo*
	$\mid$	$\forall \alpha.\tau$	*tipo polimórfico*

denotamos $\mathcal{L}(\tau)$ o conjunto das variáveis de tipo **livres** em τ, definido por

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

para um ambiente Γ, definimos igualmente

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

notamos por $\tau[\alpha \leftarrow \tau']$ a substituição de α por τ' em τ, definido por

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

as regras são **exactamente** as mesmas, mais

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

e

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

o sistema obtido é conhecido como o **sistema F** (J.-Y. Girard / J. C. Reynolds)

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

podemos agora atribuir um tipo satisfatório às primitivas

$$
\begin{aligned}
\mathit{fst} &: \quad \forall\alpha.\forall\beta.\alpha \times \beta \to \alpha \\
\mathit{snd} &: \quad \forall\alpha.\forall\beta.\alpha \times \beta \to \beta \\
\mathit{opif} &: \quad \forall\alpha.\texttt{bool} \times \alpha \times \alpha \to \alpha \\
\mathit{opfix} &: \quad \forall\alpha.(\alpha \to \alpha) \to \alpha
\end{aligned}
$$

podemos construir uma derivação (de tipo) de

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

(exercício : fazê-lo)

a condição $\alpha \notin \mathcal{L}(\Gamma)$ na regra

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

é importante

sem ela, teríamos

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

e aceitaríamos assim o programa

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

para termos sem anotações, os dois problemas

- **inferência** : dado e, existirá um τ tal que $\vdash e : \tau$?
- **verificação** : dados e e τ, teremos $\vdash e : \tau$?

não são decidíveis!

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

para obter uma inferência de tipos decidível, vamos restringir a potência expressiva do sistema F
$\Rightarrow$ o sistema de **Hindley-Milner**, utilizado no OCaml, SML, Haskell, ...

limitamos a quantificação $\forall$ à cabeça dos tipos (*quantificação prenexa*)

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \dots & \textit{tipos de base} \\
 & \mid & \tau \to \tau & \textit{tipo de uma função} \\
 & \mid & \tau \times \tau & \textit{tipo de um par} \\
 & \mid & \alpha & \textit{variável de tipo} \\
 & & & \\
\sigma & ::= & \tau & \textit{esquemas} \\
 & \mid & \forall \alpha . \sigma &
\end{array}
$$

o ambiente Γ associa um esquema de tipo a cada identificador
e a relação de tipagem tem agora a forma $\Gamma \vdash e : \sigma$

(daí em diante, anotaremos por σ os esquemas de tipo e τ os tipos monomórficos com variáveis, designaremos por α, β, etc. as variáveis de tipos)

no sistema de Hindley-Milner, os tipos seguintes são sempre aceites

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

mas não mais os tipos tais que

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

nota : na sintaxe OCaml, a quantificação prenexa está implícita

```
# fst;;
```

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

$$\forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha$$

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

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

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

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}}\ldots \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}}\ldots$$

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

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

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

notemos que somente a construção `let` permite a introdução de um tipo polimórfico no ambiente

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

em particular, podemos sempre tipar

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

com $f : \forall\alpha.\alpha \to \alpha$ no contexto para tipar $(f\ 1, f\ \texttt{true})$

no entanto, a regra de tipagem

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

não introduz um tipo polimórfico (senão $\tau_1 \to \tau_2$ não teria boa formação)

em particular, já não podemos tipar mais

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

para programar uma verificação ou uma inferência de tipo, procedemos por recorrência sobre a estrutura do programa por tipar

ora, para uma dada expressão, três regras podem ser aplicadas: a regra do sistema monomórfico, a regra

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

ou ainda a regra

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

como escolher ? deveremos proceder por *tentativa/erro* ?

vamos modificar a apresentação do sistema de Hindley-Milner para que esse fique **dirigido pela sintaxe** (*syntax-directed*), *i.e.*, para que a toda a expressão, no máximo uma regra se aplique

as regras tem a mesma capacidade de expressão : todo o termo tipável num sistema o é se e só se é tipável no outro.

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

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

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

surgem duas operações

- a **instanciação**, na regra

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

a relação $\tau \leq \sigma$ lê-se « τ é uma instância de σ » e é definida por

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

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

- e a **generalização**, na regra

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

onde

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

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

com $\Gamma' = \Gamma + f : Gen(\alpha \rightarrow \alpha, \Gamma) = \Gamma + f : \forall\alpha.\alpha \rightarrow \alpha$
se α é (devidamente) escolhida não livre em Γ

para inferir o tipo de uma expressão, resta-nos algumas situações

- em `fun` $x \rightarrow e$, que tipo dar a x ?
- para uma variável x, que instância de $\Gamma(x)$ escolher ?

existe uma solução : **o algoritmo W** (Damas, Milner, Tofte)

duas ideias

- utilizamos **novas variáveis de tipo** para representar tipos desconhecidos
  - para o tipo de x em `fun` $x \rightarrow e$
  - para instanciar as variáveis do esquema $\Gamma(x)$

- determina-se o valor desta variáveis mais tarde por **unificação entre tipos** no momento da tipagem da aplicação

corresponde em juntar conhecimento sobre as restrições que cada variável de tipo considerado tem de satisfazer
a unificação, tendo em conta o sistema de restrições (equações) em causa, propõe uma solução

sejam dois tipos τ_1 e τ_2 contendo as variáveis de tipo $\alpha_1, \dots, \alpha_n$

existirá uma instanciação f das variáveis α_i tais que $f(\tau_1) = f(\tau_2)$?

é o que chamamos a **unificação**

exemplo 1 :

$$\begin{array}{l} \tau_1 = \alpha \times \beta \rightarrow \texttt{int} \\ \tau_2 = \texttt{int} \times \texttt{bool} \rightarrow \gamma \\ \text{solução :} \quad \alpha = \texttt{int} \ \wedge \ \beta = \texttt{bool} \ \wedge \ \gamma = \texttt{int} \end{array}$$

exemplo 2 :

$$\begin{array}{l} \tau_1 = \alpha \times \texttt{int} \rightarrow \alpha \times \texttt{int} \\ \tau_2 = \gamma \rightarrow \gamma \\ \text{solução :} \quad \gamma = \alpha \times \texttt{int} \end{array}$$

exemplo 3 :

$$\tau_1 = \alpha \to \texttt{int}$$
$$\tau_2 = \beta \times \gamma$$

não há solução

exemplo 4 :

$$\tau_1 = \alpha \to \texttt{int}$$
$$\tau_2 = \alpha$$

não há solução

$\mathit{unifier}(\tau_1, \tau_2)$ determina se existe uma instância das variáveis de tipo contidas em τ_1 e τ_2 tal que $\tau_1 = \tau_2$

$$
\begin{aligned}
\mathit{unifier}(\tau, \tau) &= \text{successo} \\
\mathit{unifier}(\tau_1 \to \tau'_1, \tau_2 \to \tau'_2) &= \mathit{unifier}(\tau_1, \tau_2) \ ; \ \mathit{unifier}(\tau'_1, \tau'_2) \\
\mathit{unifier}(\tau_1 \times \tau'_1, \tau_2 \times \tau'_2) &= \mathit{unifier}(\tau_1, \tau_2) \ ; \ \mathit{unifier}(\tau'_1, \tau'_2) \\
\mathit{unifier}(\alpha, \tau) &= \begin{array}{l} \text{se } \alpha \notin \mathcal{L}(\tau), \text{ substituir } \alpha \text{ por } \tau \text{ em todo o lado} \\ \text{senão, falha} \end{array} \\
\mathit{unifier}(\tau, \alpha) &= \mathit{unifier}(\alpha, \tau) \\
\mathit{unifier}(\tau_1, \tau_2) &= \text{falha em todos os casos}
\end{aligned}
$$

(nada de pánico.... é o objecto das práticas laboratoriais)

uma ideia do algoritmo W sobre um exemplo

consideremos a expressão $\boxed{\texttt{fun}\ x \rightarrow +(\mathit{fst}\ x, 1)}$

- à x damos o tipo α_1, uma nova variável de tipo
- a primitiva $+$ tem por tipo $\texttt{int} \times \texttt{int} \rightarrow \texttt{int}$
- tipamos a expressão $(\mathit{fst}\ x, 1)$
  - fst tem por tipo o esquema $\forall\alpha.\forall\beta.\alpha \times \beta \rightarrow \alpha$
  - atribuímo-lhe assim o tipo $\alpha_2 \times \beta_1 \rightarrow \alpha_2$ para duas novas variáveis
  - a aplicação $\mathit{fst}\ x$ obriga à unificação de α_1 e de $\alpha_2 \times \beta_1$, $\Rightarrow \alpha_1 = \alpha_2 \times \beta_1$
- $(\mathit{fst}\ x, 1)$ tem assim por tipo $\alpha_2 \times \texttt{int}$
- a aplicação $+(\mathit{fst}\ x, 1)$ unifica $\texttt{int} \times \texttt{int}$ e $\alpha_2 \times \texttt{int}$, $\Rightarrow \alpha_2 = \texttt{int}$

no final obtemos o tipo $\texttt{int} \times \beta_1 \rightarrow \texttt{int}$,
generalizando obtemos finalmente $\boxed{\forall\beta.\texttt{int} \times \beta \rightarrow \texttt{int}}$

definimos a função W que toma como argumentos um ambiente Γ e uma expressão e, retorna o tipo inferido para e

- se e é uma variável x,
  retornar uma instância trivial de $\Gamma(x)$
- se e é uma constante c,
  retornar uma instância trivial do seu tipo (pense, e.g. ao caso $[] : \alpha\ \texttt{list}$)
- se e é uma primitiva *op*,
  retornar uma instância trivial do seu tipo
- se e é um par (e_1, e_2),
  calcular $\tau_1 = W(\Gamma, e_1)$
  calcular $\tau_2 = W(\Gamma, e_2)$
  retornar $\tau_1 \times \tau_2$

- se e é uma função `fun` $x \rightarrow e_1$,
  seja α uma nova variável
  calcular $\tau = W(\Gamma + x : \alpha, e_1)$
  retornar $\alpha \rightarrow \tau$

- se e é uma aplicação $e_1\ e_2$,
  calcular $\tau_1 = W(\Gamma, e_1)$
  calcular $\tau_2 = W(\Gamma, e_2)$
  seja α uma nova variável
  unificar(τ_1, $\tau_2 \rightarrow \alpha$)
  retornar α

- se e é `let` $x = e_1$ `in` e_2,
  calcular $\tau_1 = W(\Gamma, e_1)$
  retornar $W(\Gamma + x : Gen(\tau_1, \Gamma), e_2)$

Theorema (Damas, Milner, 1982)

O algoritmo W é correcto, completo e determina o tipo principal :

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

Theorema (segurança da tipagem)

O sistema de Hindley-Milner é seguro (safe).
(Se $\emptyset \vdash e : \tau$*, então a redução de e é infinita ou termina num valor.)*

existam várias formas de implementar a operação de unificação

- manipulação explícita de uma **substituição**

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

- utilizando variáveis de tipo **destrutíveis**

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

existam também várias formas de implementar o algoritmo W

- com **esquemas explícitos** e calculando $Gen(\tau, \Gamma)$

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

- com recurso a **níveis**

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

ver a Prática Laboratorial desta aula

podemos estender mini-ML de várias formas

- recursão
- tipos construídos (*n*-tuplos, listas, tipos soma e produtos)
- referências

como já o abordamos, podemos definir

$$\begin{array}{l}\texttt{let rec } f\ x = e_1 \texttt{ in } e_2 \stackrel{\text{def}}{=} \\ \qquad \texttt{let } f = \mathit{opfix}\ (\texttt{fun } f \to \texttt{fun } x \to e_1) \texttt{ in } e_2\end{array}$$

onde

$$\mathit{opfix} : \forall\alpha.\forall\beta.((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

de forma equivalente podemos considerar a regra

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

já vimos os pares

as listas não apresentam dificuldade particular

$$\begin{array}{rl} \mathit{nil}: & \forall\alpha.\alpha\ \mathit{list} \\ \mathit{cons}: & \forall\alpha.\alpha \times \alpha\ \mathit{list} \rightarrow \alpha\ \mathit{list} \end{array}$$

$$\frac{\Gamma \vdash e_1 : \tau_1\ \mathit{list} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma + x : \tau_1 + y : \tau_1\ \mathit{list} \vdash e_3 : \tau}{\Gamma \vdash \mathtt{match}\ e_1\ \mathtt{with}\ \mathit{nil} \rightarrow e_2 \mid \mathit{cons}(x, y) \rightarrow e_3 : \tau}$$

este tipo de regra generaliza-se sem dificuldade aos tipos soma e tipos produto

para as referências, podemos ingenuamente pensar que basta juntar as primitivas

$$\begin{aligned} \mathit{ref} &: \forall\alpha.\alpha \rightarrow \alpha\ \mathit{ref} \\ ! &: \forall\alpha.\alpha\ \mathit{ref} \rightarrow \alpha \\ := &: \forall\alpha.\alpha\ \mathit{ref} \rightarrow \alpha \rightarrow \mathit{unit} \end{aligned}$$

infelizmente, esta solução não é correcta !

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

é o problema dito das **referências polimórficas**

para contornar este problema, existe uma solução extremamente simples, ou seja, uma restrição sintáctica na construção `let`

Definição (*value restriction*, Wright 1995)

*Um programa respeita o critério do **value restriction** se toda a sub-expressão* `let` *é da forma*

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

onde v_1 é um valor.

já não podemos escrever

$$\texttt{let}\ r = \mathit{ref}\ (\texttt{fun}\ x \rightarrow x)\ \texttt{in}\ \dots$$

mas podemos, em contrapartida, escrever

$$(\texttt{fun}\ r \rightarrow \dots)\ (\mathit{ref}\ (\texttt{fun}\ x \rightarrow x))$$

onde o tipo de r não é generalizado

na prática, continuamos com a escrita de `let` $r = ref$ `... in ...`
mas o tipo de r não é generalizado

em OCaml, uma variável não generalizada é da forma `'_a`

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

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

a *value restriction* é igualmente ligeiramente flexibilizada para autorizar as expressões seguras, tais como a aplicação de constructores

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

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

restam no entanto alguns desagrados

```
# 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>
```

a solução : expandir para exibir uma função, *i.e.*, um valor

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

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

(fala-se de η-expansão)

na presença de sistemas de módulos, a realidade é mais complexa ainda

dado um módulo `M`

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

terei o direito de generalizar o tipo de `M.create 17` ?

a resposta depende do tipo `'a t` : **não** para uma tabela de dispersão, **sim** para uma lista pura, etc.

em OCaml, uma indicação de **variância** permite distinguir os dois casos

```
type +'a t   (* podemos generalizar *)
type  'a u   (* não podemos generalizar *)
```

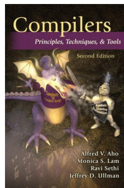
ler *Relaxing the value restriction*, J. Garrigue, 2004

duas referência bibliográfica fortemente ligadas à matéria dada nesta aula

- Benjamin C. Pierce, *Types and Programming Languages*
- Xavier Leroy e Pierre Weis, *Le langage Caml*
  (o último capítulo fala da inferência com níveis)

- aula prática
  - unificação e algoritmo W
- próxima teórica
  - análise léxica

estes acetatos resultam essencialmente de uma adaptação do material pedagógico gentilmente cedido pelo Jean-Christophe Filliâtre (link1, link2)



adicionalmente poderá consultar as obras seguintes

- **Modern Compilers: Principles, Techniques, and Tools**, Alfred V. Aho, Monica S. Lam, Ravi Sethi et Jeffrey D. Ullman
- **Types and Programming Languages**, Benjamin C. Pierce
- **Modern Compiler Implementation**, Andrew W. Appel (3 versões: ML, C, Java)