Typage avancé des langages fonctionnels

Gabriel Scherer

Gallium – INRIA

Section 1

Lambda-calcul typé

Lambda-calcul

t, u	$::=$		terme
	$\mid$	x, y, z	variable
	$\mid$	$t\ u$	application
	$\mid$	$\lambda(x)\, t$	$\lambda -$ abstraction
	$\mid$	(t, u)	paire
	$\mid$	$\pi_i\ t$	projection $(i \in \{1, 2\})$

Lambda-calcul

$$
\begin{array}{lll}
t, u & ::= & \text{terme} \\
& \mid \quad x, y, z & \text{variable} \\
& \mid \quad t\ u & \text{application} \\
& \mid \quad \lambda(x)\, t & \lambda - \text{abstraction} \\
& \mid \quad (t, u) & \text{paire} \\
& \mid \quad \pi_i\ t & \text{projection } (i \in \{1, 2\})
\end{array}
$$

$$
\frac{}{(\lambda(x)\, t)\ u \longrightarrow_\beta t[u/x]} \qquad \frac{}{\pi_i\ (t_1, t_2) \longrightarrow_\beta t_i}
$$

Lambda-calcul

$$
\begin{array}{lll}
t, u & ::= & \text{terme} \\
& \mid \quad x, y, z & \text{variable} \\
& \mid \quad t\ u & \text{application} \\
& \mid \quad \lambda(x)\, t & \lambda - \text{abstraction} \\
& \mid \quad (t, u) & \text{paire} \\
& \mid \quad \pi_i\ t & \text{projection } (i \in \{1, 2\})
\end{array}
$$

$$
\frac{}{(\lambda(x)\, t)\ u \longrightarrow_\beta t[u/x]} \qquad \frac{}{\pi_i\ (t_1, t_2) \longrightarrow_\beta t_i}
$$

$$
\pi_1\ (\lambda(x)\, x, \lambda(x)\, x)\ t \quad ?
$$

Lambda-calcul

$$
\begin{array}{lll}
t, u & ::= & \text{terme} \\
& \mid \quad x, y, z & \text{variable} \\
& \mid \quad t\ u & \text{application} \\
& \mid \quad \lambda(x)\, t & \lambda - \text{abstraction} \\
& \mid \quad (t, u) & \text{paire} \\
& \mid \quad \pi_i\ t & \text{projection } (i \in \{1, 2\})
\end{array}
$$

$$
\frac{}{(\lambda(x)\, t)\ u \longrightarrow_\beta t[u/x]} \qquad \frac{}{\pi_i\ (t_1, t_2) \longrightarrow_\beta t_i}
$$

$$
\frac{t \longrightarrow_\beta t'}{t\ u \longrightarrow_\beta t'\ u} \qquad \frac{u \longrightarrow_\beta u'}{t\ u \longrightarrow_\beta t\ u'} \qquad \ldots
$$

Lambda-calcul

$$
\begin{array}{lll}
t, u & ::= & \text{terme} \\
& \mid \quad x, y, z & \text{variable} \\
& \mid \quad t\ u & \text{application} \\
& \mid \quad \lambda(x)\, t & \lambda - \text{abstraction} \\
& \mid \quad (t, u) & \text{paire} \\
& \mid \quad \pi_i\ t & \text{projection } (i \in \{1, 2\})
\end{array}
$$

$$\frac{}{(\lambda(x)\, t)\ u \longrightarrow_\beta t[u/x]} \qquad \frac{}{\pi_i\ (t_1, t_2) \longrightarrow_\beta t_i}$$

$$
\begin{array}{lll}
E & ::= & \text{contexte (d'évaluation)} \\
& \mid \quad t\ \Box \quad \mid \quad \Box\ u & \\
& \mid \quad \lambda(x)\, \Box & \\
& \mid \quad (t, \Box) \mid (\Box, u) & \\
& \mid \quad \pi_i\ \Box &
\end{array}
$$

$$E[t] := E[t/\Box] \qquad \frac{t \longrightarrow_\beta t'}{E[t] \longrightarrow_\beta E[t']}$$

Erreurs dynamiques

$$(t, t')\ u \quad ?$$

Erreurs dynamiques

$$(t, t')\ u \quad ?$$

$$\begin{array}{lrll} A, B, C, T, U, V & ::= & & \text{type} \\ & | & X, Y, Z & \text{type de base} \\ & | & A \to B & \text{type de fonctions} \\ & | & (A, B) & \text{type produit} \end{array}$$

$$\frac{}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A)\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A * B} \qquad \frac{\Gamma \vdash t : A_1 * A_2}{\Gamma \vdash \pi_i\ t : A_i}$$

Well-typed programs do not go wrong.

Polymorphisme

[Girard 72, Reynolds 74]

$$
\begin{array}{lcll}
A, B, C, T, U, V & ::= & \text{type} & \\
& | & \ldots & \\
& | & \alpha, \beta, \gamma & \text{variables de types} \\
& | & \forall(\alpha)\, A & \text{type polymorphe}
\end{array}
$$

Polymorphisme

[Girard 72, Reynolds 74]

$$
\begin{array}{rcll}
A, B, C, T, U, V & ::= & \text{type} & \\
& | & \ldots & \\
& | & \alpha, \beta, \gamma & \text{variables de types} \\
& | & \forall(\alpha)\, A & \text{type polymorphe}
\end{array}
$$

$$
\begin{array}{rcll}
t, u & ::= & & \\
& | & \ldots & \\
& | & \Lambda(\alpha)\, t & \text{généralisation} \\
& | & t\,[A] & \text{application de type}
\end{array}
$$

Polymorphisme

[Girard 72, Reynolds 74]

$$
\begin{array}{rcll}
A, B, C, T, U, V & ::= & \text{type} & \\
& | & \ldots & \\
& | & \alpha, \beta, \gamma & \text{variables de types} \\
& | & \forall(\alpha)\, A & \text{type polymorphe}
\end{array}
$$

$$
\begin{array}{rcll}
t, u & ::= & & \\
& | & \ldots & \\
& | & \Lambda(\alpha)\, t & \text{généralisation} \\
& | & t\,[A] & \text{application de type}
\end{array}
$$

$$\frac{\Gamma, \alpha \vdash t : T}{\Gamma \vdash \Lambda(\alpha)\, t : \forall(\alpha)\, T}$$

$$\frac{\Gamma \vdash t : \forall(\alpha)\, T}{\Gamma \vdash t\,[U] : T[U/\alpha]}$$

Pas de changement de la réduction : “type erasure”

On n'a en fait plus besoin des produits pour calculer : on peut les définir avec le polymorphisme.

$$\begin{array}{rcl} T * U & := & \forall(\alpha)\, T \to U \to \alpha \\ (t, u) & := & \Lambda(\alpha)\, \lambda(f)\, f\ t\ u \\ \pi_i\ (t : A_1 * A_2) & := & t\, [A_i]\, (\lambda(x_1 : A_1)\, \lambda(x_2 : A_2)\, x_i) \end{array}$$

Implicite (Curry) ou explicite (Church)

$$\dfrac{\dfrac{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\alpha)\,\alpha \to \alpha}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : (\forall(\beta)\,\beta \to \beta) \to (\forall(\beta)\,\beta \to \beta)} \quad x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\beta)\,\beta \to \beta}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\ x : \forall(\beta)\,\beta \to \beta}}{\vdash \lambda(x)\,x\ x : \forall(\alpha)\,\alpha \to \alpha \to \forall(\beta)\,\beta \to \beta}$$

Implicite (Curry) ou explicite (Church)

$$\dfrac{\dfrac{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\alpha)\,\alpha \to \alpha}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : (\forall(\beta)\,\beta \to \beta) \to (\forall(\beta)\,\beta \to \beta)} \quad x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\beta)\,\beta \to \beta}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,x : \forall(\beta)\,\beta \to \beta}}{\vdash \lambda(x)\,x\,x : \forall(\alpha)\,\alpha \to \alpha \to \forall(\beta)\,\beta \to \beta}$$

$$\dfrac{\dfrac{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\alpha)\,\alpha \to \alpha}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,[\forall(\beta)\,\beta \to \beta] : (\forall(\beta)\,\beta \to \beta) \to (\forall(\beta)\,\beta \to \beta)} \quad \ldots}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,[\forall(\beta)\,\beta \to \beta]\,x : \forall(\beta)\,\beta \to \beta}}{\vdash \lambda(x : \forall(\alpha)\,\alpha \to \alpha)\,x\,[\forall(\beta)\,\beta \to \beta]\,x : \forall(\alpha)\,\alpha \to \alpha \to \forall(\beta)\,\beta \to \beta}$$

Implicite (Curry) ou explicite (Church)

$$\dfrac{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\alpha)\,\alpha \to \alpha}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : (\forall(\beta)\,\beta \to \beta) \to (\forall(\beta)\,\beta \to \beta)} \quad x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\beta)\,\beta \to \beta}{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,x : \forall(\beta)\,\beta \to \beta}{\vdash \lambda(x)\,x\,x : \forall(\alpha)\,\alpha \to \alpha \to \forall(\beta)\,\beta \to \beta}}$$

$$\dfrac{\dfrac{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\alpha)\,\alpha \to \alpha}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,[\forall(\beta)\,\beta \to \beta] : (\forall(\beta)\,\beta \to \beta) \to (\forall(\beta)\,\beta \to \beta)} \quad \ldots}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,[\forall(\beta)\,\beta \to \beta]\,x : \forall(\beta)\,\beta \to \beta}}{\vdash \lambda(x : \forall(\alpha)\,\alpha \to \alpha)\,x\,[\forall(\beta)\,\beta \to \beta]\,x : \forall(\alpha)\,\alpha \to \alpha \to \forall(\beta)\,\beta \to \beta}$$

Bien comprendre de quel objet mathématique on parle :

- terme annoncé ou effacé ?
- terme et type (et environnement ?)
- dérivation ?

Implicite (Curry) ou explicite (Church)

$$\dfrac{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\alpha)\,\alpha \to \alpha}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : (\forall(\beta)\,\beta \to \beta) \to (\forall(\beta)\,\beta \to \beta)} \quad x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\beta)\,\beta \to \beta}{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,x : \forall(\beta)\,\beta \to \beta}{\vdash \lambda(x)\,x\,x : \forall(\alpha)\,\alpha \to \alpha \to \forall(\beta)\,\beta \to \beta}}$$

$$\dfrac{\dfrac{\dfrac{x : \forall(\alpha)\,\alpha \to \alpha \vdash x : \forall(\alpha)\,\alpha \to \alpha}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,[\forall(\beta)\,\beta \to \beta] : (\forall(\beta)\,\beta \to \beta) \to (\forall(\beta)\,\beta \to \beta)} \quad \ldots}{x : \forall(\alpha)\,\alpha \to \alpha \vdash x\,[\forall(\beta)\,\beta \to \beta]\,x : \forall(\beta)\,\beta \to \beta}}{\vdash \lambda(x : \forall(\alpha)\,\alpha \to \alpha)\,x\,[\forall(\beta)\,\beta \to \beta]\,x : \forall(\alpha)\,\alpha \to \alpha \to \forall(\beta)\,\beta \to \beta}$$

Bien comprendre de quel objet mathématique on parle :

- terme annoncé ou effacé ?
- terme et type (et environnement ?)
- dérivation ?

Vérification immédiate, reconstruction indécidable [Joe B. Wells, 1994]

Section 2

Inférence de types

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x : ?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$
$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$
$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$
$?a =?a_1 \to ?a_2$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$
$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$
$?a =?a_1 \to ?a_2 \qquad ?b =?a_1$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$
$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$
$?a =?a_1 \to ?a_2 \qquad ?b =?a_1 \qquad ?c =?a_2$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$
$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$
$?a =?a_1 \to ?a_2 \qquad ?b =?a_1 \qquad ?c =?a_2$
$?a =?b \to ?c$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$
$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$
$?a =?a_1 \to ?a_2 \qquad ?b =?a_1 \qquad ?c =?a_2$
$?a =?b \to ?c$
$\lambda(f :?b \to ?c)\,\lambda(x :?b)\,(f\ x) :?c$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :
$\lambda(f)\,\lambda(x)\,f\ x$
$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$
$?a =?a_1 \to ?a_2 \qquad ?b =?a_1 \qquad ?c =?a_2$
$?a =?b \to ?c$
$\lambda(f :?b \to ?c)\,\lambda(x :?b)\,(f\ x) :?c$
$\Lambda(\beta)\,\Lambda(\gamma)\,\lambda(f : \beta \to \gamma)\,\lambda(x : \beta)\,f\ x :$

Inférence à la ML (1)

Idée : la difficulté vient de deviner un type polymorphe à une variable inconnue : $\lambda(x :?) \ldots$. Si on n'essaie jamais de "deviner" des quantificateurs, typer une expression t connue est assez facile :

- le terme donne lieu à un système d'équations
- on résoud le système
- les inconnues restantes peuvent être généralisées

Exemple :

$\lambda(f)\,\lambda(x)\,f\ x$

$\lambda(f :?a)\,\lambda(x :?b)\,((f\ x) :?c)$

$?a =?a_1 \to ?a_2 \qquad ?b =?a_1 \qquad ?c =?a_2$

$?a =?b \to ?c$

$\lambda(f :?b \to ?c)\,\lambda(x :?b)\,(f\ x) :?c$

$\Lambda(\beta)\,\Lambda(\gamma)\,\lambda(f : \beta \to \gamma)\,\lambda(x : \beta)\,f\ x : \forall(\beta)\,\forall(\gamma)\,(\beta \to \gamma) \to \beta \to \gamma$

Inférence à la ML (2)

Avec cette technique d'inférence, on ne devinera jamais un type polymorphe pour un paramètre de fonction.

Deux classes : types **monomorphes** notés τ, types **polymorphes** notés σ ("schéma de type").

$$
\begin{array}{lcl}
\tau & ::= & \text{monomorphic type} \\
 & | & \alpha, \beta, \gamma \\
 & | & \tau_1 \to \tau_2 \\
 & | & (\tau_1, \tau_2)
\end{array}
$$

$$
\frac{}{\Gamma, x : A \vdash x : A}
$$

$$
\begin{array}{lcl}
\sigma & ::= & \text{type schemes} \\
 & | & \tau \\
 & | & \forall(\alpha)\, \sigma
\end{array}
$$

$$
\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A)\, t : A \to B}
\qquad
\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}
$$

Inférence à la ML (3)

$$\begin{array}{lcll} \tau & ::= & \alpha, \beta, \gamma & \mid \tau_1 \to \tau_2 \quad \mid (\tau_1, \tau_2) \\ \sigma & ::= & \tau & \mid \forall(\alpha)\, \sigma \end{array}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda(x)\, t : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$$

Inférence à la ML (3)

$$\begin{array}{lcll} \tau & ::= & \alpha, \beta, \gamma & \mid \tau_1 \to \tau_2 \quad \mid (\tau_1, \tau_2) \\ \sigma & ::= & \tau & \mid \forall(\alpha)\, \sigma \end{array}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda(x)\, t : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash x : \sigma \vdash u : \tau}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : \tau}$$

Inférence à la ML (3)

$$\begin{array}{lcll} \tau & ::= & \alpha, \beta, \gamma & \mid \tau_1 \to \tau_2 \quad \mid (\tau_1, \tau_2) \\ \sigma & ::= & \tau & \mid \forall(\alpha)\, \sigma \end{array}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda(x)\, t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash x : \sigma \vdash u : \tau}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : \tau}$$

$$\frac{\Gamma, \alpha \vdash t : \sigma}{\Gamma \vdash t : \forall(\alpha)\, \sigma} \qquad \frac{\Gamma \vdash t : \forall(\alpha)\, \sigma}{\Gamma \vdash t : \sigma[U/\tau]}$$

Inférence à la ML (3)

$$\begin{array}{lcll} \tau & ::= & \alpha, \beta, \gamma & \mid \tau_1 \to \tau_2 \quad \mid (\tau_1, \tau_2) \\ \sigma & ::= & \tau & \mid \forall(\alpha)\, \sigma \end{array}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda(x)\, t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ u : \tau_2}$$

$$\frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash x : \sigma \vdash u : \tau}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : \tau}$$

$$\frac{\Gamma, \alpha \vdash t : \sigma}{\Gamma \vdash t : \forall(\alpha)\, \sigma} \qquad \frac{\Gamma \vdash t : \forall(\alpha)\, \sigma}{\Gamma \vdash t : \sigma[U/\tau]}$$

Remarque : let $x = t$ in u était auparavant encodable par $(\lambda(x)\, u)\ t$. Plus dans ce système restreint.

Principalité

Un terme peut avoir beaucoup de types ML différents. Pour $\lambda(x)\,x$:

- $\texttt{int} \to \texttt{int}$
- $\texttt{bool} \to \texttt{bool}$
- $\alpha \to \alpha$
- $\beta * \gamma \to \beta * \gamma$

Principalité

Un terme peut avoir beaucoup de types ML différents. Pour $\lambda(x)\, x$:

- $\texttt{int} \to \texttt{int}$
- $\texttt{bool} \to \texttt{bool}$
- $\alpha \to \alpha$
- $\beta * \gamma \to \beta * \gamma$

Théorème de principalité : pour tout terme dans un environnement fixé, il existe un (schéma de) type qui est plus général que tous les autres.

Ici $\forall(\alpha)\, \alpha \to \alpha$

C'est cette propriété qui permet d'inférer "le bon" type.

Polymorphisme de première classe

Par construction, les types de ML n'ont que du polymorphisme "prénexe" – quantificateurs toujours en tête.

Problématique pour la programmation orientée objet :

```
type 'a list = {
  head : 'a option;
  tail : 'a list option;
  length : int;
}
```

Polymorphisme de première classe

Par construction, les types de ML n'ont que du polymorphisme "prénexe" – quantificateurs toujours en tête.

Problématique pour la programmation orientée objet :

```
type 'a list = {
  head : 'a option;
  tail : 'a list option;
  length : int;
}

let nil = { head = None; tail = None; length = 0; }
let cons x xs = { head = Some x; tail = Some xs;
                  length = 1 + xs.length; }
```

Polymorphisme de première classe

Par construction, les types de ML n'ont que du polymorphisme "prénexe" – quantificateurs toujours en tête.

Problématique pour la programmation orientée objet :

```
type 'a list = {
  head : 'a option;
  tail : 'a list option;
  length : int;
}

let nil = { head = None; tail = None; length = 0; }
let cons x xs = { head = Some x; tail = Some xs;
                  length = 1 + xs.length; }

type 'a list = {
  ...
  map : 'b . ('a -> 'b) -> 'b list;
}
```

Polymorphisme de première classe (2)

```
auto (x : forall a . a -> a) = x x
```

Toute la puissance de Système F ?
Problème de décidabilité : on sait qu'il faudra annoter.

Problème de principalité. Si on a une fonction `choose` au type $\forall(\beta)\,\beta \to \beta \to \beta$, alors `choose` $(\lambda(x)\,x)$ peut avoir deux types incomparables :

1. $\forall(\alpha)\,((\alpha \to \alpha) \to (\alpha \to \alpha))$
2. $(\forall(\alpha)\,\alpha \to \alpha) \to (\forall(\alpha)\,\alpha \to \alpha)$

(1) can be instantiated in $(\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$, (2) cannot.
Conversely, (2) is more polymorphic than (1).

Polymorphisme de première classe (3)

Quand le polymorphisme est attaché aux champs de records (exemple POO), il est marqué explicitement dans les déclarations des types. Pas besoin de “deviner” le polymorphisme.

Dans le cas général, on fait “au mieux” selon les annotations de l'utilisateur.

Pour aller plus loin : MLF (un système de type **plus riche** que Système F, à nouveau principal). $\forall(\beta \geq \forall(\alpha)\, \alpha \to \alpha)\, \beta \to \beta$

Section 3

Types existentiels

$$\texttt{thunk}(\alpha) := \exists(\gamma)\, \gamma * (\gamma \to \alpha)$$

$$\texttt{pack}\,(1, ((+)1)) : \texttt{thunk}(\texttt{int})$$

$$\texttt{pack}\,(\texttt{true}, \lambda(b)\,\texttt{if}\ b\ \texttt{then}\ 10\ \texttt{else}\ 20) : \texttt{thunk}(\texttt{int})$$

$$\texttt{thunk}(\alpha) := \exists(\gamma)\, \gamma * (\gamma \to \alpha)$$

$$\texttt{pack}\,(1, ((+)1)) : \texttt{thunk}(\texttt{int})$$

$$\texttt{pack}\,(\texttt{true}, \lambda(b)\,\texttt{if}\ b\ \texttt{then}\ 10\ \texttt{else}\ 20) : \texttt{thunk}(\texttt{int})$$

$$\frac{\Gamma \vdash t : T[U/\alpha]}{\Gamma \vdash \mathsf{pack}\, t : \exists(\alpha)\, T} \qquad \frac{\Gamma \vdash t : \exists(\alpha)\, T \qquad \Gamma, \alpha, x : T \vdash u : U}{\Gamma \vdash \mathsf{unpack}\ \alpha, x = t\ \mathsf{in}\ u : U}$$

$$\texttt{thunk}(\alpha) := \exists(\gamma)\, \gamma * (\gamma \to \alpha)$$

$$\texttt{pack}\,(1, ((+)1)) : \texttt{thunk}(\texttt{int})$$

$$\texttt{pack}\,(\texttt{true}, \lambda(b)\,\texttt{if}\ b\ \texttt{then}\ 10\ \texttt{else}\ 20) : \texttt{thunk}(\texttt{int})$$

$$\frac{\Gamma \vdash t : T[U/\alpha]}{\Gamma \vdash \mathsf{pack}\, t : \exists(\alpha)\, T} \qquad \frac{\Gamma \vdash t : \exists(\alpha)\, T \qquad \Gamma, \alpha, x : T \vdash u : U}{\Gamma \vdash \mathsf{unpack}\ \alpha, x = t\ \mathsf{in}\ u : U}$$

Restriction de portée pour l'ouverture : pas de α dans U pour la bonne formation.

$$\mathsf{unpack}\ \alpha, (x, f) = t : \texttt{thunk}(\alpha)\ \mathsf{in}\ f\ x \qquad \text{(ok)}$$

$$\mathsf{unpack}\ \alpha, (x, f) = t : \texttt{thunk}(\alpha)\ \mathsf{in}\ x \qquad \text{(wrong)}$$

Programming language design : hide `pack`/`unpack` in a variant (sum type) constructor [Laüfer and Odersky, 1992].

```
type 'a thunk = Frozen of exists 'b . 'b * ('b -> 'a)

type ('a, 'b) func =
  | Fun of ('a -> 'b)
  | Compose of
      'exists 'e . ('e, 'b) func * ('a, 'e) func

let rec apply : type a b . (a, b) func -> a -> b =
  function
  | Fun f -> f
  | Compose (f, g) -> (fun x -> apply f (apply g x))
```

(This is imaginary syntax. Remark on polymorphic recursion.)

In practice, programming languages use a different syntax, which can be explained by the following technique :

$$(\exists(\alpha)\, T) \to U \quad \simeq \quad \forall(\alpha)\, (T \to U)$$

The constructor `K of foo` of type `'a bar` can be understood as a function `K : foo -> 'a bar`. If an existential appear in `foo`, it becomes universal :

```
type 'a thunk =
  | Frozen : 'b * ('b -> 'a) -> 'a thunk

data Thunk a =
  | forall b . K (b, b -> a)

data Thunk a =
  | K :: forall b . (b, b -> a) -> Thunk a
```

Avancé : conversion des clôtures

```
linear f = (f 2) == 2 * (f 1)

test = all linear (multiples 10) where
  multiples 1 = [(\n -> n)]
  multiples i = (\n -> n*i) : multiples (i - 1)
```

Avancé : conversion des clôtures

```
linear f = (f 2) == 2 * (f 1)

test = all linear (multiples 10) where
  multiples 1 = [(\n -> n)]
  multiples i = (\n -> n*i) : multiples (i - 1)
```

La fermeture `\n -> n + i` capture la variable `i` de son environnement. À la compilation, les fonctions sont transformées en des paires (environnement, fonction close) :

```
(\n -> t) => (Env(t), \e n -> t)
t u => (snd t) (fst t) u
```

Avancé : conversion des clôtures

```
linear (e,code) = code e 2 == 2 * code e 1

test = all linear (multiples 10) where
  multiples 1 = [((), \() n -> n)]
  multiples i = (i, \i n -> n*i) : multiples (i - 1)
```

Avancé : conversion des clôtures

```
linear (e,code) = code e 2 == 2 * code e 1

test = all linear (multiples 10) where
  multiples 1 = [((), \() n -> n)]
  multiples i = (i, \i n -> n*i) : multiples (i - 1)
```

Problème de typage.

Avancé : conversion des clôtures

```
linear (e,code) = code e 2 == 2 * code e 1

test = all linear (multiples 10) where
  multiples 1 = [((), \() n -> n)]
  multiples i = (i, \i n -> n*i) : multiples (i - 1)
```

Problème de typage.

$$
\begin{array}{rcl}
T \to U & \Rightarrow & \exists(\Delta)\,(\Delta * (\Delta \to T \to U)) \\
\lambda(x)\,t & \Rightarrow & \mathtt{pack}(\mathtt{Env}(t), \lambda(\mathtt{Env}(t), x)\,t) \\
t\ u & \Rightarrow & \text{unpack } \Delta, f = t \text{ in } (\pi_2\ f)\ (\pi_1\ f)\ u
\end{array}
$$

Avancé : OO contre ADT

```
type 'a list = {
  head : unit -> 'a option;
  tail : unit -> 'a list option;
}
```

Avancé : OO contre ADT

```
type 'a list = {
  head : unit -> 'a option;
  tail : unit -> 'a list option;
}

{
  head : ∃(Δ) Δ * (Δ -> 'a option);
  tail : ∃(Δ) Δ * (Δ -> 'a list option);
}
```

Avancé : OO contre ADT

```
type 'a list = {
  head : unit -> 'a option;
  tail : unit -> 'a list option;
}

{
  head : ∃(Δ) Δ * (Δ -> 'a option);
  tail : ∃(Δ) Δ * (Δ -> 'a list option);
}

∃(Δ) Δ * {
  head : Δ * unit -> 'a option;
  tail : Δ * unit -> 'a list option;
}
```

```
struct
  type 'a list = ∃(Δ) Δ * {
    head : Δ * unit -> 'a option;
    tail : Δ * unit -> 'a list option;
  }
  val nil : 'a list
  val cons : 'a -> 'a list -> 'a list
end

∃(Δ) struct
  type 'a list = Δ * {
    head : Δ * unit -> 'a option;
    tail : Δ * unit -> 'a list option;
  }
  val nil : 'a list
  val cons : 'a -> 'a list -> 'a list
  val length : 'a list -> int
end
```

Section 4

GADTs

Types fantômes

```
module type EXPR = sig
  type 'a expr
  val int : int -> int expr
  val bool : bool -> bool expr
  val eq : 'a expr -> 'a expr -> bool expr
end

module Expr : EXPR = struct
  type 'a expr = I of int | B of bool
  let int n = I n
  let bool b = B b
  let eq a b = match a, b with
    | I m, I n -> B (m = n)
    | B b, B c -> B (b = c)
    | I _, B _ | B _, I _ -> assert false
end
```

```
module type EXPR = sig
  type 'a expr
  val int : int -> int expr
  val bool : bool -> bool expr
  val eq : 'a expr -> 'a expr -> bool expr

  val eval : 'a expr -> 'a
end

module Expr : EXPR = struct
  type 'a expr = I of int | B of bool
  ...
  let rec eval = function
    | I n -> n
    | B b -> b
end
```

Idée : dans le cas *I*, on veut rajouter la contrainte `'a = int`.
Égalités de types.

Idée : dans le cas *I*, on veut rajouter la contrainte `'a = int`.

Égalités de types.

Syntaxe théorique :

```
type 'a expr =
  | I of ['a = int] int
  | B of ['a = bool] bool
  | Prod of exists 'b 'c ['a = 'b*'c] 'b expr * 'c expr
```

Idée : dans le cas *I*, on veut rajouter la contrainte `'a = int`.
Égalités de types.
Syntaxe théorique :

```
type 'a expr =
  | I of ['a = int] int
  | B of ['a = bool] bool
  | Prod of exists 'b 'c ['a = 'b*'c] 'b expr * 'c expr
```

Syntaxe utilisée en pratique :

```
type 'a expr =
  | I : int -> int expr
  | B : bool -> bool expr
  | Prod : 'b expr * 'c expr -> ('b * 'c) expr
```

```
data Expr a =
  | I :: Int -> Expr Int
  | B :: Bool -> Expr Bool
  | Prod :: Expr b -> Expr c -> Expr (b, c)
```

Cas général.

$$\texttt{type ty}(\overline{\alpha}) = \quad \texttt{|} \ \ldots \quad \texttt{|}\ K_i\ \texttt{of}\ \exists(\overline{\beta})\ [\overline{\alpha} = \overline{T[\overline{\beta}]}]\ \ \overline{U[\overline{\alpha}, \overline{\beta}]}$$

$$\frac{\forall j,\ \Gamma \vdash t_j : U_j[\overline{A}, \overline{B}] \qquad \forall k,\ \Gamma \vdash A_k = T_k[\overline{B}]}{\Gamma \vdash K_i\ \overline{t} : \texttt{ty}(\overline{A})}$$

$$\frac{\Gamma \vdash t : \texttt{ty}(\overline{A}) \qquad \Gamma, \overline{\beta}, [\overline{A} = \overline{T[\overline{\beta}]}] \vdash u : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ \cdots \mid K_i\ \overline{x} \mapsto u : C}$$

```
{-# OPTIONS -XGADTs #-}

data Expr a where
  I :: Int -> Expr Int
  B :: Bool -> Expr Bool
  App :: (a -> b) -> Expr a -> Expr b

func (App f t) = f

.../escape.hs:8:18:
  Couldn't match expected type `t' with actual type `a -> t1'
    `t' is a rigid type variable bound by
        the inferred type of func :: Expr t1 -> t
        at .../escape.hs:8:1
  In the expression: f
  In an equation for `func': func (App f t) = f
```

```
type _ expr =
  | I : int -> int expr
  | B : bool -> bool expr
  | App : ('a -> 'b) * 'a expr -> 'b expr

let func (App (f, u)) = f
```

```
File "escape.ml", line 6, characters 24-25:
Error: This expression has type ex#0 -> 'a
       but an expression was expected of type ex#0 -> 'a
       The type constructor ex#0 would escape its scope
```

Avancé : formats

```
Printf.printf "foo %d: bar %s"
  : int -> string -> unit
```

Avancé : formats

```
Printf.printf "foo %d: bar %s"
  : int -> string -> unit

Printf.printf
  (Lit ("foo ", Int (Lit (": bar ", String End))))

type _ fmt =
   | End : unit fmt
   | Lit : string * 'a fmt -> 'a fmt
   | Int : 'a fmt -> (int -> 'a) fmt
   | String : 'a fmt -> (String -> 'a) fmt

val printf : 't fmt -> 't
```

Section 5

Typer la mémoire modifiable

Références modifiables

```
type 'a ref
val ref : 'a -> 'a ref
val get : 'a ref -> 'a
val set : 'a ref -> 'a -> unit
```

On ne calcule pas seulement sur un terme t, mais aussi sur une **mémoire** : (t, M).

$$\frac{l \notin M}{(\mathsf{ref}\, v, M) \longrightarrow_\beta (l, M + (l \mapsto v))}$$

$$\frac{l \in M}{(\mathsf{get}\, l, M) \longrightarrow_\beta (M(l), M)} \qquad \frac{l \in M}{(\mathsf{set}\, l\, v, M) \longrightarrow_\beta ((), M[l \mapsto v])}$$

$$\frac{t \longrightarrow_\beta t'}{(t, M) \longrightarrow_\beta (t', M)} \qquad \frac{(t, M) \longrightarrow_\beta (t', M')}{(E[t], M) \longrightarrow_\beta (E[t'], M')}$$

Pour typer, on a besoin d'un typage de la mémoire : $\Gamma; \Sigma$.

$$\frac{}{\Gamma; \emptyset \vdash \emptyset} \qquad \frac{\Gamma; \Sigma \vdash M \qquad \Gamma; \Sigma \vdash v : A}{\Gamma; \Sigma + (l \mapsto A) \vdash M + (l \mapsto v)} \qquad \frac{}{\Gamma; \Sigma \vdash l : \mathsf{ref}\, \Sigma(l)}$$

$$\frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A}{\Gamma; \Sigma \vdash \mathsf{get}\, t : A} \qquad \frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A \qquad \Gamma; \Sigma \vdash u : A}{\Gamma; \Sigma \vdash \mathsf{set}\, t\, u : \mathtt{unit}}$$

(Remarque : value restriction)

Pour typer, on a besoin d'un typage de la mémoire : $\Gamma; \Sigma$.

$$\frac{}{\Gamma; \emptyset \vdash \emptyset} \qquad \frac{\Gamma; \Sigma \vdash M \qquad \Gamma; \Sigma \vdash v : A}{\Gamma; \Sigma + (l \mapsto A) \vdash M + (l \mapsto v)} \qquad \frac{}{\Gamma; \Sigma \vdash l : \mathsf{ref}\, \Sigma(l)}$$

$$\frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A}{\Gamma; \Sigma \vdash \mathsf{get}\, t : A} \qquad \frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A \qquad \Gamma; \Sigma \vdash u : A}{\Gamma; \Sigma \vdash \mathsf{set}\, t\, u : \mathtt{unit}}$$

(Remarque : value restriction)
Problème : $(\Gamma; \emptyset \vdash \mathsf{let}\, x = \mathsf{ref}\, 0\, \mathsf{in}\, \mathsf{get}\, x)$ ne préserve pas le typage.

Pour typer, on a besoin d'un typage de la mémoire : $\Gamma; \Sigma$.

$$\frac{}{\Gamma; \emptyset \vdash \emptyset} \qquad \frac{\Gamma; \Sigma \vdash M \qquad \Gamma; \Sigma \vdash v : A}{\Gamma; \Sigma + (l \mapsto A) \vdash M + (l \mapsto v)} \qquad \frac{}{\Gamma; \Sigma \vdash l : \mathsf{ref}\, \Sigma(l)}$$

$$\frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A}{\Gamma; \Sigma \vdash \mathsf{get}\, t : A} \qquad \frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A \qquad \Gamma; \Sigma \vdash u : A}{\Gamma; \Sigma \vdash \mathsf{set}\, t\, u : \mathtt{unit}}$$

(Remarque : value restriction)
Problème : $(\Gamma; \emptyset \vdash \mathsf{let}\, x = \mathsf{ref}\, 0 \,\mathsf{in}\, \mathsf{get}\, x)$ ne préserve pas le typage.

$$\frac{\Gamma; \Sigma, \Sigma' \vdash t : A \qquad t \cap \Sigma' = \emptyset}{\Gamma; \Sigma \vdash t : A}$$

Typer les effets

Quand on programme il est utile de savoir quelles fonctions modifient de l'état global. $A \xrightarrow{\rho} B$

$$\frac{}{\Gamma; \emptyset \vdash \emptyset} \qquad \frac{\Gamma; \Sigma \vdash M \qquad \Gamma; \Sigma \vdash v : A}{\Gamma; \Sigma + (l^\rho \mapsto A) \vdash M + (l^\rho \mapsto v)} \qquad \frac{}{\Gamma; \Sigma \vdash l^\rho : \mathsf{ref}\, \Sigma(l)^\rho}$$

$$\frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A^\rho}{\Gamma; \Sigma \vdash \mathsf{get}\, t : A} \qquad \frac{\Gamma; \Sigma \vdash t : \mathsf{ref}\, A \qquad \Gamma; \Sigma \vdash u : A}{\Gamma; \Sigma \vdash \mathsf{set}\, t\, u : \mathtt{unit}}$$

$$\rho(\emptyset) := 0 \qquad \rho(\Sigma, l^{\rho'} \mapsto A) := \rho(\Sigma) \cup \rho'$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x)\, t : A \to B} \qquad \frac{\Gamma, x : A; \Sigma \vdash t : B}{\Gamma; \Sigma \vdash \lambda(x)\, t : A \xrightarrow{\rho(\Sigma)} B}$$

Polymorphisme de région

Une fonction au type $\forall(\rho)\, A \xrightarrow{\rho} B$ est “observationellement” pure.