Introduction à la Compilation
Cours 9 : Langage du premier ordre

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

Le flot de contrôle

Comment programmer des fonctions non constantes ?

- Le langage des expressions arithmétiques caractérise des entiers naturels.
- Or, pour être utile, un programme produit généralement un résultat **en fonction** de ses entrées.

Les booléens et le branchement

- On étend le langage des expressions arithmétiques :

$$
\begin{array}{rcll}
e & ::= & n & \\
 & | & x & \\
 & | & e + e & \\
 & | & e * e & \\
 & | & \textbf{let } x := e \textbf{ in } e & \\
 & | & \textbf{true} \mid \textbf{false} & (0) \\
 & | & \textbf{if } e \textbf{ then } e \textbf{ else } e & (1) \\
 & | & x \, \mathcal{R}^? y & (2)
\end{array}
$$

- (0) est un booléen.
- (1) est une expression conditionnelle.
- (2) est la décision d'une propriété : ici, nous choisirons $\mathcal{R} \in \{<, >, \leq, \geq, =\}$.
- Le résultat v de l'évaluation d'une expression peut être n, **true** ou **false**.

Sémantique opérationnelle à grands pas

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_2 \Downarrow n_2}{\eta \vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{\eta \vdash n \Downarrow n} \qquad \frac{}{\eta \vdash \textbf{false} \Downarrow \textbf{false}}$$

$$\frac{}{\eta \vdash \textbf{true} \Downarrow \textbf{true}} \qquad \frac{\eta(x) = v}{\eta \vdash x \Downarrow v}$$

$$\frac{\eta \vdash e_1 \Downarrow v_1 \qquad \eta; (x \mapsto v_1) \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{let } x := e_1 \textbf{ in } e_2 \Downarrow v_2} \qquad \frac{\eta \vdash e_c \Downarrow \textbf{true} \qquad \eta \vdash e_1 \Downarrow v_1}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_1}$$

$$\frac{\eta \vdash e_c \Downarrow \textbf{false} \qquad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_2}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{true}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est vrai}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{false}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est faux}$$

L'interprète en OCaml : les valeurs

```
type value =
  | VInt of int
  | VBool of bool

exception NotInt

let as_int = function
  | VInt x → x
  | _ → raise NotInt

exception NotBool

let as_bool = function
  | VBool b → b
  | _ → raise NotBool
```

L'interprète en OCaml : l'AST

```
type binop = Add | Mul | Div | Sub

type comparison = Le | Ge | Lt | Gt | Eq

type variable = string

type t =
  | Int of int
  | Bool of bool
  | Binop of binop × t × t
  | Let of variable × t × t
  | Var of variable
  | If of t × t × t
  | Dec of comparison × t × t
```

L'interprète en OCaml : les comparaisons

```
let comparison = function
  | Le -> ( <= )
  | Ge -> ( >= )
  | Lt -> ( < )
  | Gt -> ( > )
  | Eq -> ( = )
```

L'interprète en OCaml

```
let rec eval env = function
| Int i →
   VInt i
| Bool b →
   VBool b
| Binop (op, e1, e2) →
   VInt ((binop op) (as_int (eval env e1)) (as_int (eval env e2)))
| Let (x, e1, e2) →
   eval (Env.bind env x (eval env e1)) e2
| Var x →
   Env.lookup env x
| Dec (c, e1, e2) →
   VBool ((comparison c) (as_int (eval env e1)) (as_int (eval env e2)))
| If (c, e1, e2) →
   if as_bool (eval env c) then eval env e1 else eval env e2
```

Bonne formation des expressions

Exercice

Quelles sont les expressions bloquées vis-à-vis de cette sémantique ?

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_2 \Downarrow n_2}{\eta \vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{\eta \vdash n \Downarrow n} \qquad \frac{}{\eta \vdash \textbf{false} \Downarrow \textbf{false}}$$

$$\frac{}{\eta \vdash \textbf{true} \Downarrow \textbf{true}} \qquad \frac{\eta(x) = v}{\eta \vdash x \Downarrow v} \qquad \frac{\eta \vdash e_1 \Downarrow v_1 \qquad \eta; (x \mapsto v_1) \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{let } x := e_1 \textbf{ in } e_2 \Downarrow v_2}$$

$$\frac{\eta \vdash e_c \Downarrow \textbf{true} \qquad \eta \vdash e_1 \Downarrow v_1}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_1} \qquad \frac{\eta \vdash e_c \Downarrow \textbf{false} \qquad \eta \vdash e_2 \Downarrow v_2}{\eta \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 \Downarrow v_2}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{true}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est vrai}$$

$$\frac{\eta \vdash e_1 \Downarrow n_1 \qquad \eta \vdash e_1 \Downarrow n_2}{\eta \vdash e_1 \mathcal{R}^? e_2 \Downarrow \textbf{false}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est faux}$$

Caractériser la forme de la valeur d'une expression

1. Certaines expressions sont "clairement" à valeur dans les entiers naturels tandis que d'autres sont à valeur dans l'ensemble $\{\textbf{true}, \textbf{false}\}$ des booléens :
   - 0, $1+2$, $\textbf{let}\ x := 42\ \textbf{in}\ x+1$, ... ;
   - $\textbf{true}$, $0 <^? 1$, $\textbf{let}\ b := 0 <^? 1\ \textbf{in}\ b$, ...
2. Certaines expressions sont "clairement" bloquées :
   - $0 + \textbf{true}$, $\textbf{if}\ 42\ \textbf{then}\ 1\ \textbf{else}\ 0$, ...
3. Certaines ne sont pas nécessairement bloquées mais il faudrait faire un calcul pour s'en assurer et pour connaître la forme de la valeur calculée :
   - $\textbf{if}\ (\textbf{if}\ 0 <^? 1\ \textbf{then}\ \textbf{true}\ \textbf{else}\ 21)\ \textbf{then}\ \textbf{false}\ \textbf{else}\ 42$

Exercice

Existe-t-il une analyse **statique** permettant :

- d'accepter uniquement les expressions de type 1 ?
- d'accepter aussi les programmes de type 2 ?

Le typage

- Le typage permet de répondre à la première question par l'affirmative.
- Deux types caractérisent les deux **formes** de valeur :

$$\tau ::= \mathbf{bool} \mid \mathbf{nat}$$

- On étend la syntaxe des environnements de nommage :

$$\boxed{\begin{array}{rcl} \Gamma & ::= & \bullet \\ & \mid & \Gamma; x : \tau \end{array}}$$

$\Rightarrow$ Ces objets sont des environnements de typage.

- Le jugement de typage « $\Gamma \vdash e : \tau$ » où τ est un type, se lit alors :

« Sous l'environnement de typage Γ, l'expression e a le type τ. »

Règles de typage

$$\Gamma \vdash n : \textbf{nat} \qquad \Gamma \vdash \textbf{true} : \textbf{bool} \qquad \Gamma \vdash \textbf{false} : \textbf{bool} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \textbf{nat} \qquad \Gamma \vdash e_2 : \textbf{nat}}{\Gamma \vdash e_1 \oplus e_2 : \textbf{nat}} \qquad \frac{\Gamma \vdash e_1 : \textbf{nat} \qquad \Gamma \vdash e_2 : \textbf{nat}}{\Gamma \vdash e_1 \mathcal{R}^? e_2 : \textbf{bool}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{let } x := e_1 \textbf{ in } e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_c : \textbf{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2 : \tau}$$

- Notez que le type de x dans la règle **let** peut être **déduit** de celui de e_1.

$\Rightarrow$ C'est une forme très restreinte d'inférence de type.

Fonction de typage en OCaml

```
let rec type_check env = function
  | Var x → TyEnv.lookup env x
  | Int _ → TInt
  | Bool _ → TBool
  | If (cond, e1, e2) →
      let tycond = type_check env cond in
      let ty1 = type_check env e1 and ty2 = type_check env e2 in
      if tycond ≠ TBool ∨ ty1 ≠ ty2 then raise IllTyped ;
      ty1
  | Let (x, e1, e2) →
      let ty1 = type_check env e1 in
      type_check (TyEnv.bind env x ty1) e2
  | Binop (_, e1, e2) →
      let ty1 = type_check env e1 and ty2 = type_check env e2 in
      if (ty1 ≠ TInt ∨ ty2 ≠ TInt) then raise IllTyped ;
      TInt
  | Dec (_, e1, e2) →
      let ty1 = type_check env e1 and ty2 = type_check env e2 in
      if (ty1 ≠ TInt ∨ ty2 ≠ TInt) then raise IllTyped ;
      TBool
```

Théorème de correction du typage

- On peut montrer que :

$$\forall\ e\ \tau, \bullet \vdash e : \tau \Rightarrow \exists\ v : \tau, \bullet \vdash e \Downarrow v$$

Compilation vers une machine virtuelle

- Les programmes de la machine virtuelle définie lors du dernier cours sont des séquences d'instruction évaluées de la première à la dernière.
- En présence de branchement, on ne veut exécuter qu'un **sous-ensemble** des instructions, suivant le chemin d'exécution caractérisé par l'issue des tests.
- En d'autres termes, il s'agit de **contrôler quelles opérations sont évaluées**.
- C'est le rôle des opérateurs de flot de contrôle.

Nouveau langage pour la machine virtuelle

$$
\begin{array}{rcll}
\textit{programme} & ::= & (\ell : \textit{instruction})^{+} & \\
\\
\textit{instruction} & ::= & \textbf{remember}\ n & n \in \mathbb{N} \\
 & \mid & \textbf{add} \mid \textbf{mul} \mid \textbf{div} \mid \textbf{sub} & \\
 & \mid & \textbf{cmple} \mid \textbf{cmpge} \mid \textbf{cmpeq} & \\
 & \mid & \textbf{cmplt} \mid \textbf{cmpgt} & \\
 & \mid & \textbf{getvar}\ i & i \in \mathbb{N} \\
 & \mid & \textbf{define} & \\
 & \mid & \textbf{undefine} & \\
 & \mid & \textbf{branch}\ \ell \mid \textbf{branchif}\ \ell, \ell &
\end{array}
$$

- Les étiquettes ℓ sont des noms symboliques pour les emplacements des instructions dans le programme.
- Les deux dernières instructions sont des opérateurs de contrôle.

Une machine virtuelle à pile

- Les opérateurs de flot influencent la fonction de transition du contrôle.
- On doit maintenant expliciter le flot du contrôle dans la sémantique :

$$\mathit{instruction} : (pc, \zeta_v, \zeta_r) \rightarrow (pc', \zeta'_v, \zeta'_r)$$

  et qui se lit :

  > « *L'instruction à la position* pc *du programme transforme les piles* (ζ_v, ζ_r) *en* (ζ'_v, ζ'_r) *et transporte l'exécution à la position* pc'. »

Sémantique à petits pas des instructions

$$
\begin{array}{rcl}
\textbf{op} & : & \text{(Pour toutes les opérations définies dans le dernier cours.)} \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (pc + 1, \ldots, \ldots) \\
 & & \\
\textbf{branch } \ell & : & \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (\ell, \zeta_v, \zeta_r) \\
 & & \\
\textbf{branchif } \ell_1, \ell_2 & : & // \text{ Si } \mathit{Observe}(\zeta_r, 0) \equiv \textbf{true} \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (\ell_1, \zeta_v, \mathit{Depile}(\zeta_r)) \\
 & & \\
\textbf{branchif } \ell_1, \ell_2 & : & // \text{ Si } \mathit{Observe}(\zeta_r, 0) \equiv \textbf{false} \\
(pc, \zeta_v, \zeta_r) & \rightarrow & (\ell_2, \zeta_v, \mathit{Depile}(\zeta_r))
\end{array}
$$

Interprète en OCaml

```
let rec eval_instr labels vm = function
  | Branch l ->
      goto labels vm l

  | BranchIf (l1, l2) ->
      let x = Stack.inspect 0 vm.val_stack in
      let vm =
        { vm with val_stack = Stack.pop vm.val_stack }
      in
      (match x with
         | VBool true -> goto labels vm l1
         | VBool false -> goto labels vm l2
         | _ -> raise NotBool)
  | ...
and goto labels vm l =
  eval_instr labels vm (snd (vm.prog.(pos_of_label labels l)))
```

La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{if } 0 \leq 1 \textbf{ then } 0 \textbf{ else } 1) =$

La fonction de traduction sur quelques exemples

- $\mathcal{C}(\textbf{if } 0 \leq 1 \textbf{ then } 0 \textbf{ else } 1) =$

	remember 1
	remember 0
	cmple
	branchif ℓ_1, ℓ_2
ℓ_1 :	**remember** 0
	branch ℓ_3
ℓ_2 :	**remember** 1
	branch ℓ_3
ℓ_3 :	

Traduction

$$
\boxed{
\begin{array}{rcrl}
\mathcal{C}(\textbf{if } e_c \textbf{ then } e_1 \textbf{ else } e_2) & = & & \mathcal{C}(e_c) \\
 & & & \textbf{branchif } \ell_1, \ell_2 \\
 & & \ell_1: & \mathcal{C}(e_1) \\
 & & & \textbf{branch } \ell_3 \\
 & & \ell_2: & \mathcal{C}(e_2) \\
 & & & \textbf{branch } \ell_3 \\
 & & \ell_3: &
\end{array}
}
$$

Une (mauvaise) idée

- Dans le langage source, l'opérateur de flot de contrôle « **if** ... **then** ... **else** ... » est très peu **expressif** : il ne permet pas de coder une boucle par exemple.
- Au contraire, dans le langage de destination, on a **toute liberté** pour coder des itérations :

	remember 1
ℓ :	**remember** 1
	add
	branch ℓ

- Pourquoi ne pas rajouter une construction de branchement arbitraire (**goto**) directement dans le langage source ?

Un langage d'expressions avec **goto**

$$
\begin{array}{rcl}
e & ::= & n \\
  & | & x \\
  & | & e + e \\
  & | & e * e \\
  & | & \textbf{let } x := e \textbf{ in } e \\
  & | & \textbf{true} \mid \textbf{false} \\
  & | & \textbf{if } e \textbf{ then } e \textbf{ else } e \\
  & | & x \, \mathcal{R}^? y \\
  & | & \ell : e \\
  & | & \textbf{goto } \ell
\end{array}
$$

Une sémantique pour le **goto**

- De façon à pouvoir évaluer l'expression ciblée par un **goto**, une structure associative doit associer une expression à toute étiquette ℓ.
- On se donne une syntaxe pour représenter cette association :

$$\boxed{\begin{array}{rcl} \xi & ::= & \bullet \\ & | & \xi; \ell \mapsto e \end{array}}$$

- On suppose qu'une première analyse de l'expression e a conduit à la construction du dictionnaire associant une expression à chaque étiquette.
- On étend ensuite la syntaxe des jugements d'évaluation :

$$\eta, \xi \vdash e \Downarrow v$$

qui se lit :

« Sous l'environnement η et considérant le dictionnaire ξ, l'évaluation de l'expression e mène à la valeur v. »

Règles de la sémantique à grands pas

$$\frac{\eta, \xi; \ell \mapsto e \vdash e \Downarrow v}{\eta, \xi \vdash \ell : e \Downarrow v} \qquad \frac{\xi(\ell) = e \qquad \eta, \xi \vdash e \Downarrow v}{\eta, \xi \vdash \textbf{goto } \ell \Downarrow v}$$

Exemple

$$\begin{array}{l}
\textbf{let } n := 5 \textbf{ in} \\
\ell : \textbf{if } n = 0 \textbf{ then } 1 \\
\quad \textbf{else } n \times (\textbf{let } n := n - 1 \textbf{ in goto } \ell)
\end{array}$$

Difficulté du raisonnement global sur les programmes

- En présence de **goto**, pour savoir si une expression est bien formée, il ne suffit plus de regarder « au dessus » et de vérifier la bonne liaison des variables.
- En effet, si cette expression est étiquetée par ℓ, il faut maintenant vérifier **dans l'ensemble du programme** qu'à toute occurence de l'expression **goto** ℓ l'environnement d'évaluation définit toutes les variables nécessaires à l'évaluation de la fonction.

$\Rightarrow$ C'est une forme de portée dynamique.

Exemple

$$
\begin{array}{l}
\textbf{let } \mathit{fact} := \\
\quad \textbf{let } n := 5 \textbf{ in} \\
\quad\quad \ell : \textbf{if } n = 0 \textbf{ then } 1 \\
\quad\quad\quad \textbf{else } n \times (\textbf{let } n := n - 1 \textbf{ in goto } \ell) \\
\textbf{in} \\
\textbf{let } y := \textbf{goto } \ell \ \textbf{ in } y
\end{array}
$$

- Cette expression est bloquée car l'évaluation de **goto** ℓ nécessite une variable n qui n'est plus définie.

Difficulté du raisonnement global sur les programmes

- En présence de **goto**, même si une expression est bien formée, il est difficile de s'assurer qu'elle fait bien « ce que l'on veut ».

Exemple

$$
\begin{array}{l}
\textbf{let } \mathit{fact5} := \\
\quad \textbf{let } n := 5 \textbf{ in} \\
\qquad \ell : \textbf{if } n = 0 \textbf{ then } 1 \\
\qquad\quad \textbf{else } n \times (\textbf{let } n := n - 1 \textbf{ in goto } \ell) \\
\textbf{in} \\
\textbf{let } y := \textbf{let } n := -5 \textbf{ in goto } \ell \ \textbf{ in } y
\end{array}
$$

- Quand un programme a **plusieurs points d'entrée** , il est difficile de raisonner sur la cohérence des entrées.
- Imaginez une expression de plusieurs milliers de lignes de code : pour s'assurer que `fact5` se comporte bien, il faut vérifier la totalité du programme.

$\Rightarrow$ Ceci est caractéristique d'un logiciel non modulaire.

Un autre exemple : un programme BASIC

```
10 INPUT "What is your name: ", U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: ", N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? ", A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

La crise du logiciel

- Cette incapacité à raisonner de façon modulaire sur les programmes a provoqué une crise du logiciel dans les années 60.
- C'est à cette époque que le coût de la construction du logiciel a dépassé le coût de celle des machines.
- Les logiciels étaient alors de faible qualité, inefficaces, incorrects, impossibles à faire évoluer et à corriger, . . .

$\Rightarrow$ Pour remédier à ces problèmes, a été introduit un ensemble de méthodes et d'outils : la programmation structurée.

Les mécanismes de la programmation structurée

Contrôle structuré

- La programmation structurée interdit une utilisation arbitraire du **goto**.
- Il a été prouvé que l'on peut se limiter à un ensemble restreint d'opérateur de flot de contrôle :

Théorème (Complétude de la programmation structurée)

Toute fonction calculable peut être réalisée dans un langage de programmation qui combine les sous-programmes de seulement trois façons :

1. *Séquencement : exécuter un sous-programme puis un autre.*
2. *Sélection : exécuter un sous-programme en fonction d'un booléen.*
3. *Répétition : exécuter un sous-programme tant qu'une condition est vraie.*

- Ainsi, on peut interdire le **goto**.

Comment réaliser ces opérateurs de flot de contrôle ?

- Nous allons voir deux façons de réaliser ces opérateurs de contrôle, l'une basée uniquement sur les appels de fonctions (cours d'aujourd'hui) et l'autre sur les opérateurs de flot de contrôle de la programmation impérative (un cours prochain).

Les appels de fonctions

Le concept de fonction

- Du point de vue des langages de programmation, une fonction est un composant logiciel qui :
  - est un fragment de code (relativement) indépendant du reste du programme ;
  - possède un **unique point d'entrée** ;
  - est **parametré** par un certain nombre d'arguments formels ;
  - **redonne le contrôle** à son appelant lorsque son calcul est terminé.

Utilité des fonctions

- Factorisation du code.

⇒ À sémantique équivalente, un programme court est **préférable** à un long.

- Réutilisation du code.

⇒ Il vaut mieux raisonner **une fois** de façon générale que d'instancier un raisonnement particulier plusieurs fois.

- Respecte le principe de **décomposition des problèmes en sous-problèmes**.

⇒ C'est l'unique moyen d'aborder un problème **complexe**.

- Permet de travailler en équipe.

⇒ Il suffit de s'entendre sur l'interface et la spécification des fonctions pour se partager leur développement.

- Permet d'**abstraire**, c'est-à-dire se doter du **vocabulaire adéquat** pour résoudre le problème. On **cache ainsi les détails d'implémentation non pertinents**, que l'on pourra d'ailleurs changer indépendamment des utilisations de la fonction tant que l'on ne change pas la spécification.

⇒ Une fois un problème résolu par une fonction, on ne s'intéresse plus à la façon de résoudre ce problème mais à ce qu'apporte sa solution.

Les fonctions de seconde classe

- Aujourd'hui, nous allons nous intéresser à une implantation particulière des fonctions telle qu'on la trouve dans les langages de programmation du premier ordre (C, Pascal, ...).
- Dans ces langages du premier ordre (contrairement aux langages fonctionnels ou objets, dits d'ordre supérieur), les fonctions sont des composants logiciels que l'on peut pas manipuler comme des valeurs arbitraires du calcul.
- En d'autres termes, dans un langage du premier ordre, l'ensemble des fonctions est fixé une fois pour toute par le code source du programme et n'évolue pas au cours de l'évaluation du programme.

Un langage d'expressions et de fonctions

$$
\boxed{
\begin{array}{rcll}
e & ::= & \ldots & \\
  & \mid & f\ (e_1, \ldots, e_n) & (1) \\
 & & & \\
\mathit{declaration} & ::= & \mathbf{def}\ f(x_1, \ldots, x_n) := e & (2) \\
 & & & \\
\mathit{programme} & ::= & \mathit{declaration}^+\ \mathbf{in}\ \ e & (3)
\end{array}
}
$$

- (1) représente les appels de fonction.
- (2) définit une fonction f d'arguments formels $x_1, \ldots, x_n$ et de corps e.
- (3) est un programme composé de déclarations et d'un corps.

Exemple

$$
\begin{array}{l}
\textbf{def } \mathit{fact}(n) := \\
\quad \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \mathit{fact}\ (n-1) \\
\textbf{in} \\
\quad \mathit{fact}\ (5)
\end{array}
$$

Exemple

$$
\begin{array}{l}
\textbf{def } \mathit{even}(n) := \\
\quad \textbf{if } n = 0 \textbf{ then true else if } n = 1 \textbf{ then false else } \mathit{odd}\ (n-1) \\
\textbf{def } \mathit{odd}(n) := \\
\quad \textbf{if } n = 0 \textbf{ then false else if } n = 1 \textbf{ then true else } \mathit{even}\ (n-1) \\
\textbf{in} \\
\quad \mathit{even}\ (5)
\end{array}
$$

Sémantique opérationnelle

- De nouveau, il suffit d'une première analyse capable d'associer dans un dictionnaire ξ à chaque nom de fonction f, son corps et l'ensemble de ses arguments formels.

Règles de la sémantique opérationnelle

$$\frac{\xi(f) = (x_1, \ldots, x_n, e) \qquad \begin{array}{c}\forall i \in \{1..n\} \quad \eta, \xi \vdash e_i \Downarrow v_i \\ \bullet; (x_1 \mapsto v_1) \ldots; (x_n \mapsto v_n), \xi \vdash e \Downarrow v\end{array}}{\eta, \xi \vdash f(e_1, \ldots, e_n) \Downarrow v}$$

Exemple

$$
\begin{array}{l}
\textbf{def } \mathit{fact}(n) := \\
\quad \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \mathit{fact}\ (n-1) \\
\textbf{in} \\
\quad \mathit{fact}\ (3)
\end{array}
$$

Exercice

Écrivez l'arbre de dérivation correspondant à l'évaluation de ce programme.

Expressions bloquées

- Un appel de fonction est bloqué dans deux cas de figure :
  - le nom de la fonction est indéfini ;
  - l'arité de la fonction est incorrecte (trop ou trop peu d'arguments effectifs).
- Par ailleurs, l'évaluation du corps de la fonction appelée peut échouer si les types des arguments effectifs ne sont pas ceux attendus par la fonction.
- Pour le premier cas, une analyse statique de bonne liaison des noms de fonction dans le dictionnaire suffit.
- Pour le second cas, et aussi pour s'assurer que les types des arguments effectifs sont corrects, il faut associer à chaque fonction une signature.

Signature de fonction

- La syntaxe des signatures de fonction est définie par :

$$\boxed{\sigma \quad ::= \quad \tau \times \ldots \times \tau \to \tau}$$

  où τ est soit le type **int**, soit le type **bool**.
- On suppose, pour le moment, qu'il existe une fonction Σ qui associe une signature σ à chaque fonction f
- On peut alors étendre le jugement de typage :

$$\Sigma, \Gamma \vdash e : \tau$$

  qui se lit :

  « Sous le dictionnaire de signatures Σ et l'environnmment de typage Γ, l'expression e a le type τ. »

Règle de typage des appels de fonction

$$\frac{\Sigma(f) = \tau_1 \times \ldots \times \tau_n \to \tau \qquad \forall i \in \{1..n\} \quad \Sigma, \Gamma \vdash e_i : \tau_i}{\Sigma, \Gamma \vdash f(e_1, \ldots, e_n) : \tau}$$

Vérification du dictionnaire de signatures de fonction

- Il reste maintenant à s'assurer que la signature $\xi(f)$ est effectivement une signature correcte pour f.
- Pour cela, on définit le jugement :

$$\Sigma \vdash \textbf{def } f(x_1, \ldots, x_n) := e : \sigma$$

par l'unique règle :

$$\boxed{\dfrac{\Sigma, \bullet; x_1 : \tau_1; \ldots; x_n : \tau_n \vdash e : \tau}{\Sigma \vdash \textbf{def } f(x_1, \ldots, x_n) := e : \tau_1 \times \ldots \times \tau_n \rightarrow \tau}}$$

Comment obtenir le dictionnaire des signatures ?

- Il y a deux procédés pour obtenir le dictionnaire des signatures Σ :
  1. On étend la syntaxe du langage par des annotations de type, ainsi :

     $$\boxed{\quad \textit{declaration} \quad ::= \quad \textbf{def } f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau := e \quad}$$

     dont on peut extraire facilement Σ.
  2. On utilise un algorithme d'inférence de type dont nous parlerons dans un prochain cours.

Caractéristique du mécanisme d'appel de fonction

- Un appel de fonction garantit que les arguments formels de la fonction ne sont pas modifiables de l'extérieur.

$\Rightarrow$ C'est cette propriété qui permet de garantir la **modularité**.

Nous regarderons les choses d'un peu plus près . . .

- La sémantique opérationnelle à grands pas ne met pas en lumière le mécanisme de va-et-vient du contrôle entre la fonction appelée et le contexte appelant.
- Il faut pourtant le comprendre pour écrire un compilateur.

$\Rightarrow$ Ce sera le sujet du prochain cours.

Synthèse

Synthèse

- Nous avons introduit une analyse statique appelée typage.
- Tous les programmes bien typés s'évaluent sans bloquer.
- Nous avons une sémantique du branchement conditionnel et sa compilation.
- Un autre opérateur de flot de contrôle très riche a été présenté : le **goto**.
- Cependant, il s'avère **trop** riche.
- Nous avons abordé le mécanisme d'appel de fonction.