Sémantique des programmes

La sémantique consiste à donner un sens aux programmes.

- Sémantique informelle Exemples :
  - *La conditionnelle est de la forme **if** expr **then** $instr_1$ **else** $instr_2$. Si l'expression expr est vraie, alors l'instruction $instr_1$ est exécutée ; sinon l'instruction $instr_2$ est exécutée.*
  - *L'itération non bornée est de la forme **while** (expression) instruction. La sous-instruction est exécutée de manière répétée tant que la valeur de l'expression reste non nulle. On teste l'expression avant d'exécuter la sous-instruction.*
- Sémantique formelle Elle consiste en l'emploi de théories mathématiques ou de logique formelle pour suppléer les imprécisions ou les ambiguïtés de la sémantique informelle (« Si », « est vraie », « reste non nulle » etc.).

Sémantique opérationnelle

- La sémantique opérationnelle est une sémantique formelle.
- Elle définit un ensemble de valeurs puis une relation d'*évaluation* entre les programmes et les valeurs (c.-à-d. les résultats).
- L'évaluation est définie inductivement sur la syntaxe abstraite du langage, c.-à-d. que la valeur d'une construction dépend de la forme de celle-ci de la valeur de ses parties. En d'autres termes encore, la valeur d'un AST dépend de la forme de sa racine et des valeurs de ses sous-arbres.
- Formellement, un interprète est l'implantation d'une sémantique opérationnelle (qui est alors, par construction, définie en termes de la sémantique du langage d'implantation).

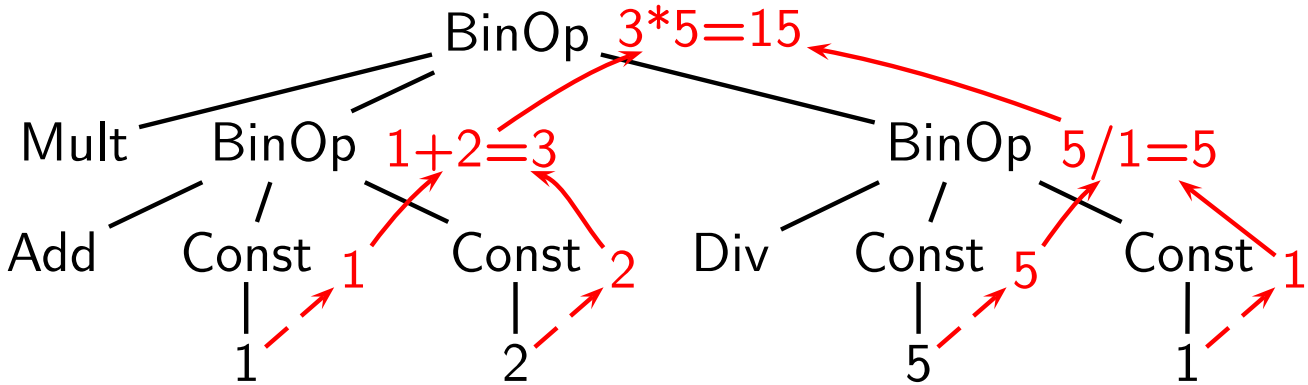
La calculette est un interprète

Les valeurs sont les entiers : **type** *value* = *int*;;
L'interprète, c.-à-d. la fonction d'évaluation, est :

```
let rec eval e = match e with
    Const n → n
|   BinOp (op,e1,e2) → (* L'ordre d'évaluation de e1 et e2 n'est pas spécifié : *)
      let v1 = eval e1 and v2 = eval e2
      in begin match op with
            Add → v1 + v2 | Sub → v1 - v2 | Mult → v1 * v2 | Div → v1 / v2
         end ;;
```

Une interprétation simple

Reprenons l'exemple `"(1+2)*(5/1)"` (cf. page 12). Soit e l'AST.
Son évaluation (cf. page 15) est interprétée par « eval e » et est représentée en couleur : les résultats (des appels récursifs) sont en rouge.



Le flot de contrôle est descendant (la racine est examinée avant les sous-arbres) et le flot de données est ascendant (eval ne prend aucun argument).

Les règles d'inférence

Une autre présentation, plus mathématique, consiste à définir une relation $e \twoheadrightarrow v$ (dite *jugement*), qui se lit « L'expression e s'évalue en la valeur v », par des *règles d'inférence*.
Ce sont des implications logiques $P_1 \wedge P_2 \wedge \ldots \wedge P_n \Rightarrow C$ présentées sous la forme :

$$\frac{P_1 \qquad P_2 \qquad \ldots \qquad P_n}{C}$$

Les propositions P_i sont alors nommées *prémisses*, et C la *conclusion*.
Lorsqu'il n'y a pas de prémisses, alors C est un *axiome* et on le note simplement C.
Lecture calculatoire : pour évaluer C, il faut d'abord évaluer les P_i (l'ordre n'étant pas spécifié).

Les règles d'inférence (suite)

Les règles et axiomes peuvent contenir des variables non quantifiées (par $\forall$ ou $\exists$) explicitement. Dans ce cas elles sont implicitement quantifiées universellement en tête de règle. Par exemple l'axiome $A(x)$ signifie $\forall x.A(x)$ et la règle

$$\frac{P_1(x) \qquad P_2(y)}{P(x,y)}$$

signifie $\forall x, y.(P_1(x) \wedge P_2(y) \Rightarrow P(x,y))$.
Étant donné un ensemble de règles d'inférence portant sur une ou plusieurs relations, on définit alors celles-ci comme étant les *plus petites relations* satisfaisant les règles.

Plus petites relations

Soient les règles suivantes, portant sur les prédicats Pair(n) et Impair(n) :

$$\text{Pair}(0) \qquad \frac{\text{Impair}(n)}{\text{Pair}(n+1)} \qquad \frac{\text{Pair}(n)}{\text{Impair}(n+1)}$$

Il faut les lire comme les conditions :

$$\text{Pair}(0)$$
$$\forall n.(\text{Impair}(n) \Rightarrow \text{Pair}(n+1))$$
$$\forall n.(\text{Pair}(n) \Rightarrow \text{Impair}(n+1))$$

De nombreux prédicats satisfont ces conditions, p.ex. Pair(n) et Impair(n) vrais pour tout n. Mais les plus petits prédicats (ceux vrais le moins souvent) satisfaisant sont $\text{Pair}(n) \triangleq (n \mod 2 = 0)$ et $\text{Impair}(n) \triangleq (n \mod 2 = 1)$.

Arbres de preuve

Un *arbre de preuve* est un arbre portant à chaque nœud la conclusion d'une règle d'inférence dont les prémisses correspondent aux fils de ce nœud. Les feuilles de l'arbre portent donc des axiomes. La conclusion de la preuve est la racine de l'arbre. Celle-ci est représentée en bas de la page. Par exemple, voici la preuve de Impair(3) avec le système page 19 :

$$\dfrac{\dfrac{\dfrac{\text{Pair}(0)}{\text{Impair}(1)}}{\text{Pair}(2)}}{\text{Impair}(3)}$$

Sémantique opérationnelle de la calculette

Une *méta-variable* est une variable du langage de description, ici la logique formelle, et non pas une variable du langage décrit (ici le langage d'expressions de la calculette). Ainsi

- les expressions, c.-à-d. les valeurs du type *expr*, sont notées e ;
- les valeurs sont notées v ($v \in \mathbb{Z}$) ;
- les entiers mathématiques associés à leur représentation OCaml n sont notés $\dot{n}$ ($\dot{n} \in \mathbb{Z}$).

Donc e, v, n et $\dot{n}$ sont des méta-variables.

- Les jugements sont de la forme $e \twoheadrightarrow v$

Sémantique opérationnelle de la calculette (suite et fin)

$$\text{Const } n \twoheadrightarrow \dot{n} \quad \text{const} \qquad \frac{e_1 \twoheadrightarrow v_1 \qquad e_2 \twoheadrightarrow v_2}{\text{BinOp}(\text{Add}, e_1, e_2) \twoheadrightarrow v_1 + v_2}\ \text{add}$$

$$\frac{e_1 \twoheadrightarrow v_1 \qquad e_2 \twoheadrightarrow v_2}{\text{BinOp}(\text{Sub}, e_1, e_2) \twoheadrightarrow v_1 - v_2}\ \text{sub}$$

$$\frac{e_1 \twoheadrightarrow v_1 \qquad e_2 \twoheadrightarrow v_2}{\text{BinOp}(\text{Mult}, e_1, e_2) \twoheadrightarrow v_1 \times v_2}\ \text{mult} \qquad \frac{e_1 \twoheadrightarrow v_1 \qquad e_2 \twoheadrightarrow v_2}{\text{BinOp}(\text{Div}, e_1, e_2) \twoheadrightarrow v_1 / v_2}\ \text{div}$$

Implantation des sémantiques opérationnelles

La façon systématique de programmer une sémantique opérationnelle en OCaml consiste à écrire un motif de filtre par règle d'inférence, comme le suggère

$$\frac{e_1 \twoheadrightarrow v_1 \qquad e_2 \twoheadrightarrow v_2}{\mathsf{BinOp}\,(\mathsf{Sub}, e_1, e_2) \twoheadrightarrow v_1 - v_2}\ \text{sub}$$

qui devient `| BinOp (Sub,e1,e2) -> let v1 = eval e1 and v2 = eval e2 in v1 - v2`

Note : page 15 nous avons regroupé toutes les règles avec BinOp en conclusion pour ne former qu'un seul motif.

Il faut se souvenir que l'évaluation des filtres OCaml est spécifiée (c'est l'ordre d'écriture des motifs) alors qu'il n'y a pas de telle notion dans la sémantique opérationnelle : *les règles ne sont pas ordonnées*.

Déterminisme

Il est important que les mêmes données conduisent toujours au même résultat. Cette propriété s'appelle le *déterminisme*. Formellement, une sémantique opérationnelle déterministe $e \twoheadrightarrow v$ vérifie la propriété

$$\textit{Si } e \twoheadrightarrow v \textit{ et } e \twoheadrightarrow v' \textit{ alors } v = v'.$$

En d'autres termes, l'évaluation est alors une *fonction partielle* (des expressions vers les valeurs). Autrement dit encore, les mêmes données conduisent toujours aux mêmes résultats.
Pour démontrer cette propriété, on raisonne par récurrence structurelle sur les arbres de preuve de $e \twoheadrightarrow v$ et $e \twoheadrightarrow v'$, ainsi que par cas sur la forme de e.

Ajout des variables et de la liaison locale

Dans le but de simplifier l'écriture des expressions, on souhaite nommer des sous-expressions, comme dans l'extrait de syntaxe concrète

```
let x = 1+2*7 in 9*x*x - x + 2
```

Pour cela il faut ajouter

- les *identificateurs* (`x`)
- et la *liaison locale* (`let ... in ...`)

à la syntaxe concrète des expressions.

Remarque On emploie le terme de *variable* pour qualifier les identificateurs au niveau de la syntaxe abstraite pour des raisons historiques (car rien de varie ici), mais *une variable est un nom*, pas un objet.

Ajout des variables et de la liaison locale (suite)

- Syntaxe concrète

```
Expression ::= ...  | ident                    /* identifier */
             | "let" ident "=" Expression "in" Expression
```

Il faudrait définir l'ensemble de lexèmes dénoté par `ident`.

- Syntaxe abstraite

type *expr* = ... | Var **of** *string* (* On parle plutôt de *variable* ici. *)
| Let **of** *string* * *expr* * *expr*;;

Les variables sont notées x. Il ne faut pas confondre x (méta-variable dénotant n'importe quelle variable), Var "x" (l'AST d'une variable particulière du langage décrit) et "x" ou `x` (code source de la variable précédente).

Quelle sémantique opérationnelle pour ces expressions avec variables ?

Environnements

Un *environnement* associe des variables à des valeurs (par construction, *leur* valeur). Une telle association est une *liaison*.

- Une liaison est une paire (x, v). Par abus de langage on la notera $x \mapsto v$, comme un environnement au domaine réduit à une seule variable.
- Un environnement est une fonction partielle des variables vers les valeurs. On peut implanter l'environnement vide en OCaml par :

  let empty_env = **fun** x → raise Not_found
- L'ajout d'une liaison $x \mapsto v$ à un environnement ρ se note $\rho \oplus (x \mapsto v)$. Si x était déjà liée dans ρ, c.-à-d. si $\rho(x)$ était définie, alors cette nouvelle liaison cache l'ancienne, c.-à-d. $(\rho \oplus x \mapsto v)(x) = v$ même si $\rho(x) \neq v$. Implantation :

  let extend env (x,v) = **fun** y → **if** x = y **then** v **else** env y
- Les jugements sont maintenant de la forme $\rho \vdash e \twoheadrightarrow v$

Sémantique opérationnelle étendue

$$\rho \vdash \mathsf{Const}\ n \twoheadrightarrow \dot{n} \quad \text{const} \qquad \frac{x \in \mathrm{dom}(\rho)}{\rho \vdash \mathsf{Var}\ x \twoheadrightarrow \rho(x)}\ \text{var}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \mathsf{BinOp}\,(\mathsf{Add}, e_1, e_2) \twoheadrightarrow v_1 + v_2}\ \text{add}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \mathsf{BinOp}\,(\mathsf{Sub}, e_1, e_2) \twoheadrightarrow v_1 - v_2}\ \text{sub}$$

Sémantique opérationnelle étendue

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \mathsf{BinOp}\,(\mathsf{Mult}, e_1, e_2) \twoheadrightarrow v_1 \times v_2}\ \mathsf{mult}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \mathsf{BinOp}\,(\mathsf{Div}, e_1, e_2) \twoheadrightarrow v_1 / v_2}\ \mathsf{div}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \oplus x \mapsto v_1 \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \mathsf{Let}\,(x, e_1, e_2) \twoheadrightarrow v_2}\ \mathsf{let}$$

Interprétation étendue

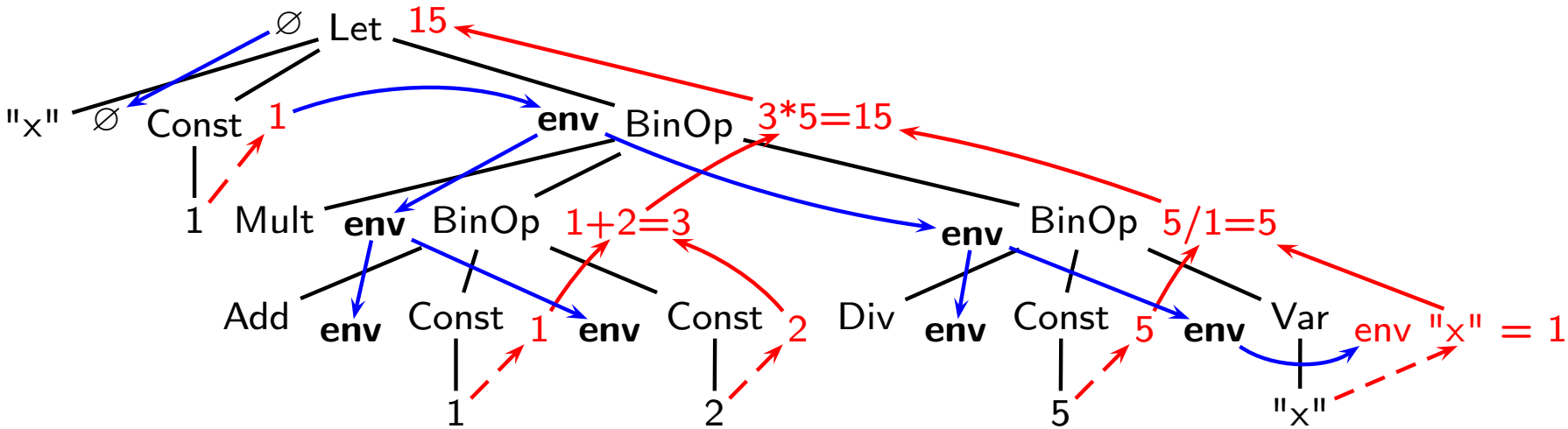
```
let rec eval env e = match e with
    Const n → n
|   BinOp (op,e1,e2) →
      let v1 = eval env e1 and v2 = eval env e2
      in begin match op with
            Add → v1 + v2 | Sub → v1 - v2 | Mult → v1 * v2 | Div → v1 / v2
         end
|   Var x → env x
|   Let (x,e1,e2) → let v1 = eval env e1 in eval (extend env (x,v1)) e2;;
```

Interprétation étendue (suite)

- Notez le codage de $\rho \oplus x \mapsto v_1$ par extend env (x,v_1).
- L'évaluation de l'expression initiale doit se faire dans un environnement vide.
- Les valeurs dans la sémantique sont dans $\mathbb{Z}$ alors que dans l'interprète elles sont de type *int*, *donc peuvent déborder*. On n'abordera pas ce problème ici.

Un exemple simple d'évaluation

Reprenons l'exemple page 12, en introduisant une variable qui lie l'expression : `"let x = 1 in (1+2)*(5/x)"`. Soit e l'arbre de syntaxe abstraite associé. Son évaluation est interprétée par « eval empty_env e ». Dénotons empty_env par $\varnothing$ et extend empty_env ("x",1) par env. Les résultats sont en rouge.



Une notation plus lisible

Afin de simplifier la présentation des évaluations nous pouvons utiliser une fonction auxiliaire, des chaînes de caractères vers les expressions, qui correspond à la composition de l'analyse lexicale et syntaxique. Nous la noterons $\langle\!\langle \_ \rangle\!\rangle$: *string* $\rightarrow$ *expr*, mais nous omettrons les guillemets de la chaîne :

$\langle\!\langle$`let x = 1 in let y = 2 in x + y`$\rangle\!\rangle$

$= \langle\!\langle$**let** x = 1 **in let** y = 2 **in** x + y$\rangle\!\rangle$

$=$ Let ("x", Const 1, Let ("y", Const 2, BinOp (Add, Var "x", Var "y")))

Une notation plus lisible (suite)

Des méta-variables $\overline{e}$ peuvent apparaître dans la syntaxe concrète pour désigner des chaînes de caractères produites par la règle de grammaire `Expression` :

$$\langle\!\langle \textbf{let}\ \mathsf{x} = 2\ \textbf{in}\ \overline{e} \rangle\!\rangle = \mathsf{Let}\ (\text{"x"}, \mathsf{Const}\, 2, \langle\!\langle \overline{e} \rangle\!\rangle)$$

Pour simplifier encore, on écrira e au lieu de $\langle\!\langle \overline{e} \rangle\!\rangle$.

Une notation plus lisible (suite)

Nous pouvons en fait aisément donner une définition formelle de l'analyse lexico-syntaxique sur les programmes... dont la syntaxe est déjà correcte.

$$
\begin{aligned}
\langle\!\langle \overline{n} \rangle\!\rangle &= \mathsf{Const}\,(\mathsf{int\_of\_string}\,(\overline{n})) = \mathsf{Const}\,(n) \\
\langle\!\langle \overline{e}_1 + \overline{e}_2 \rangle\!\rangle &= \mathsf{BinOp}\,(\mathsf{Add}, \langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle) \\
\langle\!\langle \overline{e}_1 - \overline{e}_2 \rangle\!\rangle &= \mathsf{BinOp}\,(\mathsf{Sub}, \langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle) \\
\langle\!\langle \overline{e}_1 * \overline{e}_2 \rangle\!\rangle &= \mathsf{BinOp}\,(\mathsf{Mult}, \langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle) \\
\langle\!\langle \overline{e}_1 / \overline{e}_2 \rangle\!\rangle &= \mathsf{BinOp}\,(\mathsf{Div}, \langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle) \\
\langle\!\langle x \rangle\!\rangle &= \mathsf{Var}\,(x) \\
\langle\!\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle &= \mathsf{Let}\,(x, \langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle) \\
\langle\!\langle (\overline{e}) \rangle\!\rangle &= \langle\!\langle \overline{e} \rangle\!\rangle
\end{aligned}
$$

Des règles plus lisibles

$$\rho \vdash \langle\!\langle \overline{n} \rangle\!\rangle \twoheadrightarrow \dot{n} \quad \text{const} \qquad \frac{x \in \mathrm{dom}(\rho)}{\rho \vdash \langle\!\langle x \rangle\!\rangle \twoheadrightarrow \rho(x)}\ \text{var}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \overline{e}_1 + \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_1 + v_2}\ \text{add} \qquad \frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \overline{e}_1 - \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_1 - v_2}\ \text{sub}$$

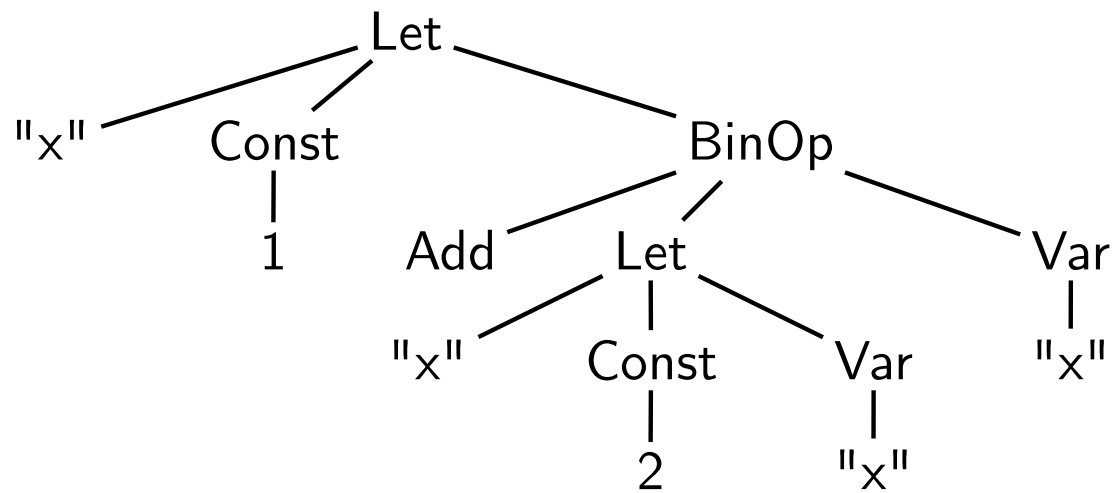
$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \overline{e}_1 * \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_1 \times v_2}\ \text{mult} \qquad \frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \overline{e}_1 \,/\, \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_1 / v_2}\ \text{div}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \oplus x \mapsto v_1 \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_2}\ \text{let}$$

Un exemple complexe d'évaluation

Soit l'extrait de syntaxe concrète
`"let x = 1 in ((let x = 2 in x) + x)"`.

L'arbre de syntaxe abstraite produit par l'analyseur lexico-syntaxique est :



Arbre de preuve de l'évaluation

On combine les règles d'inférence pour évaluer l'expression : on obtient un *arbre de preuve* (de l'évaluation de l'expression en la valeur 3), ou *dérivation*.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash \langle\!\langle 1 \rangle\!\rangle \twoheadrightarrow 1} \qquad \dfrac{}{\text{"x"} \mapsto 1 \vdash \langle\!\langle \mathbf{2} \rangle\!\rangle \twoheadrightarrow 2} \qquad \dfrac{}{\text{"x"} \mapsto 1 \oplus \text{"x"} \mapsto 2 \vdash \langle\!\langle \mathsf{x} \rangle\!\rangle \twoheadrightarrow 2}}{\text{"x"} \mapsto 1 \vdash \langle\!\langle \mathbf{let}\ \ \mathbf{x}\ \ =\ \ \mathbf{2}\ \ \mathbf{in}\ \ \mathsf{x} \rangle\!\rangle \twoheadrightarrow 2} \qquad \dfrac{}{\text{"x"} \mapsto 1 \vdash \langle\!\langle \mathsf{x} \rangle\!\rangle \twoheadrightarrow 1}}{\text{"x"} \mapsto 1 \vdash \langle\!\langle (\mathbf{let}\ \ \mathbf{x}\ \ =\ \ \mathbf{2}\ \ \mathbf{in}\ \mathsf{x}) + \mathsf{x} \rangle\!\rangle \twoheadrightarrow 2 + 1}}{\varnothing \vdash \langle\!\langle \mathbf{let}\ \mathsf{x} = 1\ \mathbf{in}\ ((\mathbf{let}\ \ \mathbf{x}\ \ =\ \ \mathbf{2}\ \ \mathbf{in}\ \mathsf{x}) + \mathsf{x}) \rangle\!\rangle \twoheadrightarrow 3}$$

Construction de l'arbre de preuve

L'arbre de preuve se construit de bas en haut, c.-à-d. de la racine vers les feuilles, en fonction de la forme des conclusions et on en déduit à chaque étape des équations sur les méta-variables dénotant les valeurs.

Puis ces équations sont résolues et donnent la valeur recherchée, c.-à-d. le résultat de l'évaluation.

Soit v la valeur du terme initial $\langle\!\langle \textbf{let}\ \mathsf{x} = 1\ \textbf{in}\ ((\textbf{let}\ \mathsf{x} = 2\ \textbf{in}\ \mathsf{x}) + \mathsf{x}) \rangle\!\rangle$

Construction de l'arbre de preuve (suite)

La seule règle qui peut avoir une conclusion de cette forme est let (p. 36). Nous appliquons donc une instance de celle-ci (avec un environnement vide) :

$$\frac{\varnothing \vdash \langle\!\langle 1 \rangle\!\rangle \twoheadrightarrow 1 \qquad \text{"x"} \mapsto 1 \vdash \langle\!\langle (\textbf{let}\ \ \textbf{x}\ \ =\ \ \textbf{2}\ \ \textbf{in}\ \ \textbf{x}) + \text{x} \rangle\!\rangle \twoheadrightarrow v}{\varnothing \vdash \langle\!\langle \textbf{let}\ \text{x} = 1\ \textbf{in}\ ((\textbf{let}\ \ \textbf{x}\ \ =\ \ \textbf{2}\ \ \textbf{in}\ \ \textbf{x}) + \text{x}) \rangle\!\rangle \twoheadrightarrow v}$$

Construction de l'arbre de preuve (suite et fin)

La seconde prémisse ne peut être qu'une conclusion de la règle add (p. 36) :

$$\frac{\text{"x"} \mapsto 1 \vdash \langle\!\langle \textbf{let} \quad \textbf{x} \quad = \quad \textbf{2} \quad \textbf{in} \;\; \text{x} \rangle\!\rangle \twoheadrightarrow v_1 \qquad \text{"x"} \mapsto 1 \vdash \langle\!\langle \text{x} \rangle\!\rangle \twoheadrightarrow 1}{\text{"x"} \mapsto 1 \vdash \langle\!\langle (\textbf{let} \quad \textbf{x} \quad = \quad \textbf{2} \quad \textbf{in}\, \text{x}) + \text{x} \rangle\!\rangle \twoheadrightarrow v_1 + 1}$$

et l'équation que l'on déduit est alors simplement $v = v_1 + 1$.
La première prémisse ne peut être que la conclusion d'une règle let (p. 36) :

$$\frac{\text{"x"} \mapsto 1 \vdash \langle\!\langle \textbf{2} \rangle\!\rangle \twoheadrightarrow 2 \qquad \text{"x"} \mapsto 1 \oplus \text{"}\textbf{x}\text{"} \mapsto 2 \vdash \langle\!\langle \text{x} \rangle\!\rangle \twoheadrightarrow 2}{\text{"x"} \mapsto 1 \vdash \langle\!\langle \textbf{let} \quad \textbf{x} \quad = \quad \textbf{2} \quad \textbf{in} \;\; \text{x} \rangle\!\rangle \twoheadrightarrow 2}$$

d'où $v_1 = 2$. En substituant v_1 par sa valeur, il vient $v = 2 + 1 = 3$.
QED

Formalisation des erreurs

Lors de l'évaluation présentée à la page 32, plusieurs problèmes auraient pu se présenter : x aurait pu valoir 0 (division par zéro) ou "x" $\notin \mathrm{dom}(\rho)$. Dans le premier cas, la règle est :

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \overline{e}_1 \,/\, \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_1 / v_2}\ \text{div}$$

On peut formaliser les cas corrects, et éventuellement les erreurs :

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2 \qquad v_2 \neq 0}{\rho \vdash \langle\!\langle \overline{e}_1 \,/\, \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_1 / v_2} \qquad \frac{\rho \vdash e_2 \twoheadrightarrow 0}{\rho \vdash \langle\!\langle \overline{e}_1 \,/\, \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathit{erreur}}$$

Pour formaliser *erreur* on remplace la relation $\rho \vdash e \twoheadrightarrow v$ par $\rho \vdash e \twoheadrightarrow r$, où r est une *réponse*. Les réponses sont désormais des valeurs ou des *erreurs*.

Formalisation des erreurs (production des erreurs)

Nous allons considérer à partir de maintenant que les valeurs dans la sémantique sont de type *int* au lieu de l'ensemble mathématique $\mathbb{Z}$, car nous n'avons pas besoin de tant d'abstraction et cela nous rapprochera de l'implantation, c.-à-d. de l'interprète.

type *value* = *int*;;
type *error* = DivByZero | FreeVar **of** *string*;;
type *result* = Val **of** *value* | Err **of** *error*;;

Les règles qui peuvent produire des erreurs sont

$$\frac{\rho \vdash e_2 \twoheadrightarrow \mathsf{Val}\,(0)}{\rho \vdash \langle\!\langle \overline{e}_1 \,/\, \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Err}\,(\mathsf{DivByZero})}\ \text{div-zero}$$

$$\frac{x \notin \mathrm{dom}(\rho)}{\rho \vdash \langle\!\langle x \rangle\!\rangle \twoheadrightarrow \mathsf{Err}\,(\mathsf{FreeVar}\,x)}\ \text{free-var}$$

Formalisation des erreurs (production des valeurs)

$$\frac{\rho \vdash e_1 \twoheadrightarrow \mathsf{Val}\,(v_1) \qquad \rho \vdash e_2 \twoheadrightarrow \mathsf{Val}\,(v_2)}{\rho \vdash \langle\!\langle \overline{e}_1 \,/\, \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Val}\,(v_1 \,/\, v_2)}\ \mathsf{div}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow \mathsf{Val}\,(v_1) \qquad \rho \vdash e_2 \twoheadrightarrow \mathsf{Val}\,(v_2)}{\rho \vdash \langle\!\langle \overline{e}_1 * \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Val}\,(v_1 * v_2)}\ \mathsf{mult}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow \mathsf{Val}\,(v_1) \qquad \rho \vdash e_2 \twoheadrightarrow \mathsf{Val}\,(v_2)}{\rho \vdash \langle\!\langle \overline{e}_1 + \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Val}\,(v_1 + v_2)}\ \mathsf{add}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow \mathsf{Val}\,(v_1) \qquad \rho \vdash e_2 \twoheadrightarrow \mathsf{Val}\,(v_2)}{\rho \vdash \langle\!\langle \overline{e}_1 - \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Val}\,(v_1 - v_2)}\ \mathsf{sub}$$

Formalisation des erreurs (production des valeurs — suite et fin)

$$\rho \vdash \langle\!\langle \overline{n} \rangle\!\rangle \twoheadrightarrow \mathsf{Val}\,(n) \quad \text{const} \qquad \frac{x \in \mathrm{dom}(\rho)}{\rho \vdash \langle\!\langle x \rangle\!\rangle \twoheadrightarrow \mathsf{Val}\,(\rho(x))}\ \text{var}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow \mathsf{Val}\,(v_1) \qquad \rho \oplus x \mapsto v_1 \vdash e_2 \twoheadrightarrow \mathsf{Val}\,(v_2)}{\rho \vdash \langle\!\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Val}\,(v_2)}\ \text{let}$$

Formalisation des erreurs (propagation des erreurs)

$$\frac{\rho \vdash e_1 \twoheadrightarrow \mathsf{Err}\,(z) \qquad \rho \vdash e_2 \twoheadrightarrow r}{\rho \vdash \langle\!\langle \overline{e}_1 * \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Err}\,(z)}\ \text{add-err}_1$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow r \qquad \rho \vdash e_2 \twoheadrightarrow \mathsf{Err}\,(z)}{\rho \vdash \langle\!\langle \overline{e}_1 + \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Err}\,(z)}\ \text{add-err}_2$$

Remarques

- Il faut deux règles car deux prémisses peuvent s'évaluer en des erreurs et la sémantique n'exprime pas la commutativité de l'addition. Les autres cas sont similaires.
- En cas d'erreurs multiples, seule une sera propagée. L'ordre d'évaluation n'étant volontairement pas complètement spécifié, on ne peut dire *a priori* quelle erreur sera propagée.

Implantation de la gestion d'erreur dans l'interprète

Pour simplifier, nous utilisons le système d'exception du langage d'implantation, s'il existe (et nous nous passons donc du type *result*). C'est le cas en OCaml :

exception Err **of** *error*;;
let rec eval env e = **match** e **with** ...
| Var x → **begin try** env x **with** Not_found → raise (Err(FreeVar x)) **end**
| BinOp (**Div**,e_1,e_2) → **let** v_1 = eval env e_1 **and** v_2 = eval env e_2
in if v_2 = 0 **then** raise (Err(DivByZero)) **else** v_1/v_2
| ...

Remarque Nous pourrions accélérer le traitement d'erreur en évaluant d'abord e_2 puis e_1 seulement si v_2 <> 0. L'interprète fixe alors l'ordre d'évaluation des arguments de la division, mais officiellement l'usager de l'interprète ne doit s'en tenir qu'à la sémantique opérationnelle (qui ne fixe pas l'ordre).

L'ordre d'évaluation des arguments

- Si l'ordre d'évaluation est spécifié par la sémantique (p.ex. Java), alors celle-ci est dite non ambiguë.
- Si non (p.ex. C et Caml), l'auteur du compilateur peut optimiser.

La sémantique opérationnelle semble parfois ne rien dire, mais elle peut exprimer

- les dépendances entre les évaluations (cf. règle let p. 36),
- les évaluations en présence d'erreurs : comparez avec page 43 la règle

$$\frac{\rho \vdash e_1 \twoheadrightarrow r_1 \qquad \rho \vdash e_2 \twoheadrightarrow \mathsf{Val}\,(0)}{\rho \vdash \langle\!\langle \overline{e}_1 \,/\, \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathsf{Err}\,(\mathsf{DivByZero})}\ \text{div-zero}$$

Variables libres

Il est possible de déterminer si des variables sont libres dans une expression *avant d'évaluer celle-ci*, et donc d'éviter l'erreur FreeVar à l'exécution. Ce type d'analyse est dite *statique* car elle a lieu à la compilation.
Soit $\mathcal{L}$ la fonction qui associe une expression à ses variables libres. On peut noter $\mathcal{L}\langle\langle \_ \rangle\rangle$ au lieu de $\mathcal{L}(\langle\langle \_ \rangle\rangle)$. Elle est définie par les équations (la priorité de $\setminus$ est plus grande que celle de $\cup$) :

$$\begin{aligned}
\mathcal{L}\langle\langle \overline{n} \rangle\rangle &= \varnothing \\
\mathcal{L}\langle\langle x \rangle\rangle &= \{x\} \\
\mathcal{L}\langle\langle \overline{e}_1 \ \overline{o} \ \overline{e}_2 \rangle\rangle &= \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \\
\mathcal{L}\langle\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\rangle &= \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \setminus \{x\}
\end{aligned}$$

où $\overline{o}$ désigne une chaîne de caractère produite par la règle grammaticale `BinOp`.

Expressions closes

Reprenons l'exemple page 37 :
`"let x = 1 in ((let x = 2 in x) + x)"`.

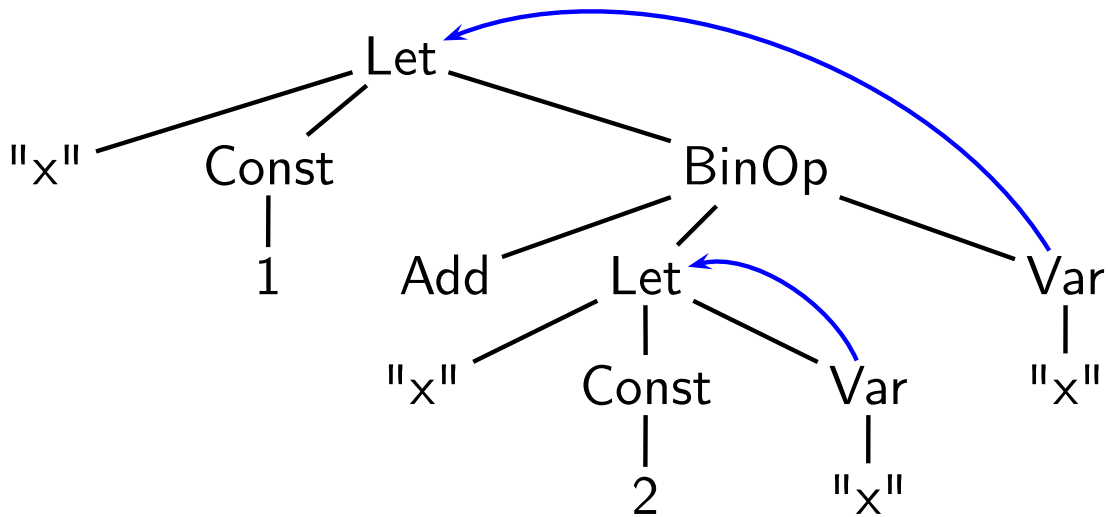
$$\mathcal{L}\langle\!\langle \textbf{let x} = 1 \textbf{ in } ((\textbf{let x} = 2 \textbf{ in x}) + \text{x})\rangle\!\rangle$$

$$\begin{aligned}
&= \mathcal{L}\langle\!\langle 1\rangle\!\rangle \cup \mathcal{L}\langle\!\langle (\textbf{let x} = 2 \textbf{ in x}) + \text{x}\rangle\!\rangle \backslash \{\text{"x"}\} \\
&= \varnothing \cup (\mathcal{L}\langle\!\langle \textbf{let x} = 2 \textbf{ in x}\rangle\!\rangle \cup \mathcal{L}\langle\!\langle \text{x}\rangle\!\rangle) \backslash \{\text{"x"}\} \\
&= (\mathcal{L}\langle\!\langle 2\rangle\!\rangle \cup \mathcal{L}\langle\!\langle \text{x}\rangle\!\rangle \backslash \{\text{"x"}\} \cup \{\text{"x"}\}) \backslash \{\text{"x"}\} \\
&= (\varnothing \cup \{\text{"x"}\} \backslash \{\text{"x"}\} \cup \{\text{"x"}\}) \backslash \{\text{"x"}\} \\
&= \varnothing
\end{aligned}$$

L'expression ne contient donc aucune variable libre. Par définition, une telle expression est dite *close*. De plus, l'expression ne contenant aucune division, nous avons prouvé qu'il n'y aura pas d'erreurs lors de l'évaluation.

Représentation graphique des liaisons dans une expression

Pour le moment, le constructeur Let est le seul à ajouter des liaisons dans l'environnement (p. 36) : on le dit « liant ». Reprenons l'AST de l'exemple page 37. À partir de chaque occurrence de variable (Var), remontons vers la racine. Si nous trouvons un Let liant cette variable, créons un arc entre celle-ci et le Let liant. Si, à la racine, aucun Let n'a été trouvé, la variable est libre dans l'expression.



Une conditionnelle avec test à zéro

Revenons au système page 36 et une conditionnelle avec test à zéro.

- Syntaxe concrète

```
Expression ::=
  ... | "ifz" Expression "then" Expression
                         "else" Expression
```

- Syntaxe abstraite

  type *expr* = ... | Ifz **of** *expr* * *expr* * *expr*;;

- Analyse syntaxique

  $\langle\!\langle \textbf{ifz}\ \overline{e}_1\ \textbf{then}\ \overline{e}_2\ \textbf{else}\ \overline{e}_3 \rangle\!\rangle = \mathsf{Ifz}\,(\langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle, \langle\!\langle \overline{e}_3 \rangle\!\rangle)$

- Variables libres

  $\mathcal{L}\langle\!\langle \textbf{ifz}\ \overline{e}_1\ \textbf{then}\ \overline{e}_2\ \textbf{else}\ \overline{e}_3 \rangle\!\rangle = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \cup \mathcal{L}(e_3)$

Une conditionnelle avec test à zéro (suite)

- Sémantique opérationnelle

$$\frac{\rho \vdash e_1 \twoheadrightarrow 0 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \textbf{ifz } \overline{e}_1 \textbf{ then } \overline{e}_2 \textbf{ else } \overline{e}_3 \rangle\!\rangle \twoheadrightarrow v_2} \text{ if-then}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow \dot{n} \qquad \dot{n} \neq 0 \qquad \rho \vdash e_3 \twoheadrightarrow v_3}{\rho \vdash \langle\!\langle \textbf{ifz } \overline{e}_1 \textbf{ then } \overline{e}_2 \textbf{ else } \overline{e}_3 \rangle\!\rangle \twoheadrightarrow v_3} \text{ if-else}$$

Calculette + fonctions = mini-ML

Ajoutons à la calculette des fonctions (*abstractions*) et leur appel (*application*).

- Syntaxe concrète

```
Expression ::= ...
             | "fun" ident "->" Expression /* abstraction */
             | Expression Expression       /* application */
```

- Syntaxe abstraite

  type *expr* = ... | Fun **of** *string* * *expr* | App **of** *expr* * *expr*;;

- Analyse syntaxique

  La priorité de l'abstraction (resp. l'application) est inférieure (resp. supérieure) à celle des opérateurs.

$$\langle\langle \textbf{fun}\ x \to \overline{e} \rangle\rangle = \text{Fun}\,(x, \langle\langle \overline{e} \rangle\rangle)$$
$$\langle\langle \overline{e}_1\ \overline{e}_2 \rangle\rangle = \text{App}\,(\langle\langle \overline{e}_1 \rangle\rangle, \langle\langle \overline{e}_2 \rangle\rangle)$$

Mini-ML et le *bootstrap*

- Fun (x, e) désigne une fonction qui à la variable x (le *paramètre*) associe l'expression e (le *corps*).
- App (e_1, e_2) désigne l'application d'une expression e_1 (dont on attend qu'elle s'évalue en une abstraction) à une expression e_2 (l'*argument*).

Nous souhaitons en fait que le langage de notre calculette fonctionnelle aie la même sémantique opérationnelle que le sous-ensemble de OCaml, nommé mini-ML, avec lequel sa syntaxe se confond.

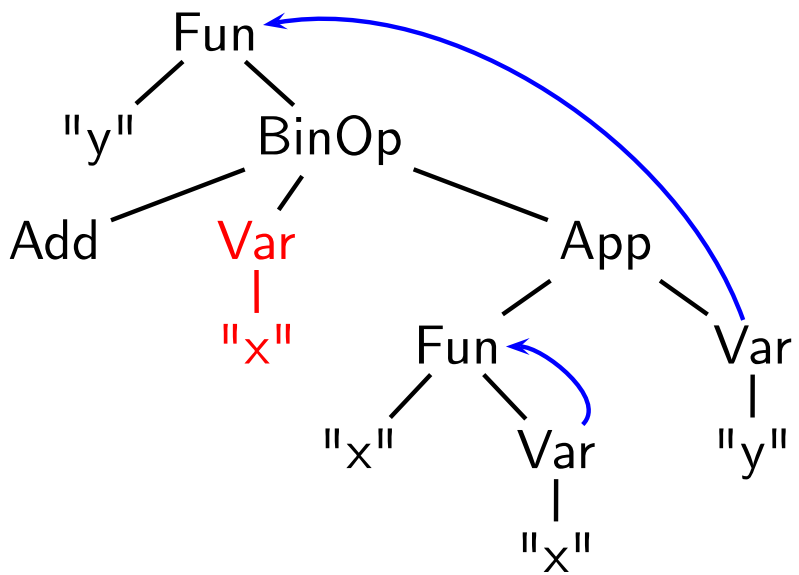
L'implantation d'un interprète ou d'un compilateur dans le même langage qu'il interprète ou compile se nomme un *bootstrap* (une auto-génération). Par exemple, le compilateur OCaml est lui-même auto-généré, un premier compilateur étant écrit en langage C.
Quelle sémantique opérationnelle pour l'abstraction et l'application ?

Variables libres d'une abstraction et d'une application

On peut étendre la définition p. 49 ainsi :

$$\begin{cases} \mathcal{L}\langle\langle \textbf{fun}\ x \to \overline{e} \rangle\rangle = \mathcal{L}(e) \backslash \{x\} \\ \mathcal{L}\langle\langle \overline{e}_1\ \overline{e}_2 \rangle\rangle = \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \end{cases}$$

Exemple $\mathcal{L}\langle\langle \textbf{fun}\ \mathsf{y} \to \mathsf{x} + (\textbf{fun}\ \mathsf{x} \to \mathsf{x})\ \mathsf{y} \rangle\rangle = \{\mathsf{x}\}$. Graphiquement :



Sémantique opérationnelle de l'abstraction — premier jet

Essayons

$$\rho \vdash \langle\!\langle \textbf{fun } x \to \overline{e} \rangle\!\rangle \twoheadrightarrow \langle\!\langle \textbf{fun } x \to \overline{e} \rangle\!\rangle \quad \text{abs-dyn}$$

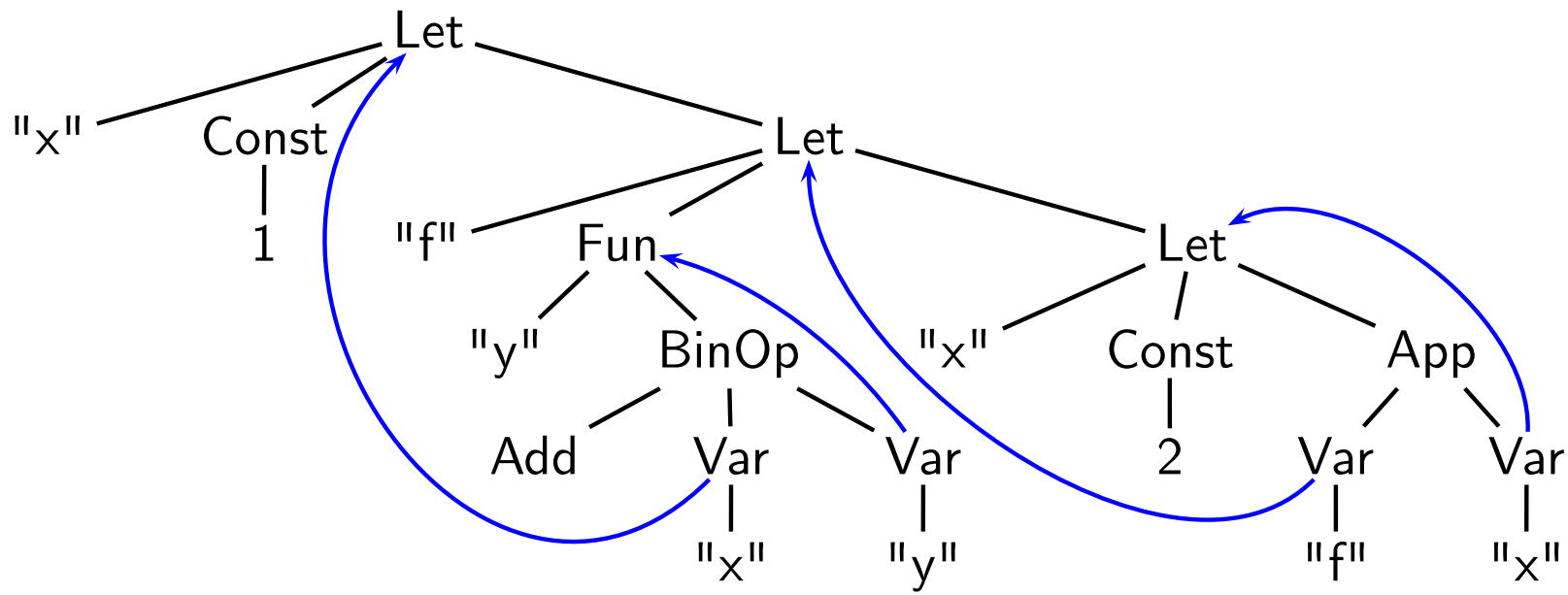
Par ailleurs remplacer une variable x par sa valeur laisse invariante la valeur de l'expression contenant x (on parle de *transparence référentielle*). C'est une propriété très désirable.

En conjonction avec abs-dyn, cela implique alors que les deux programmes suivants sont équivalents :

```
let x = 1 in
   let f = fun y → x + y in
   let x = 2
in f x
```

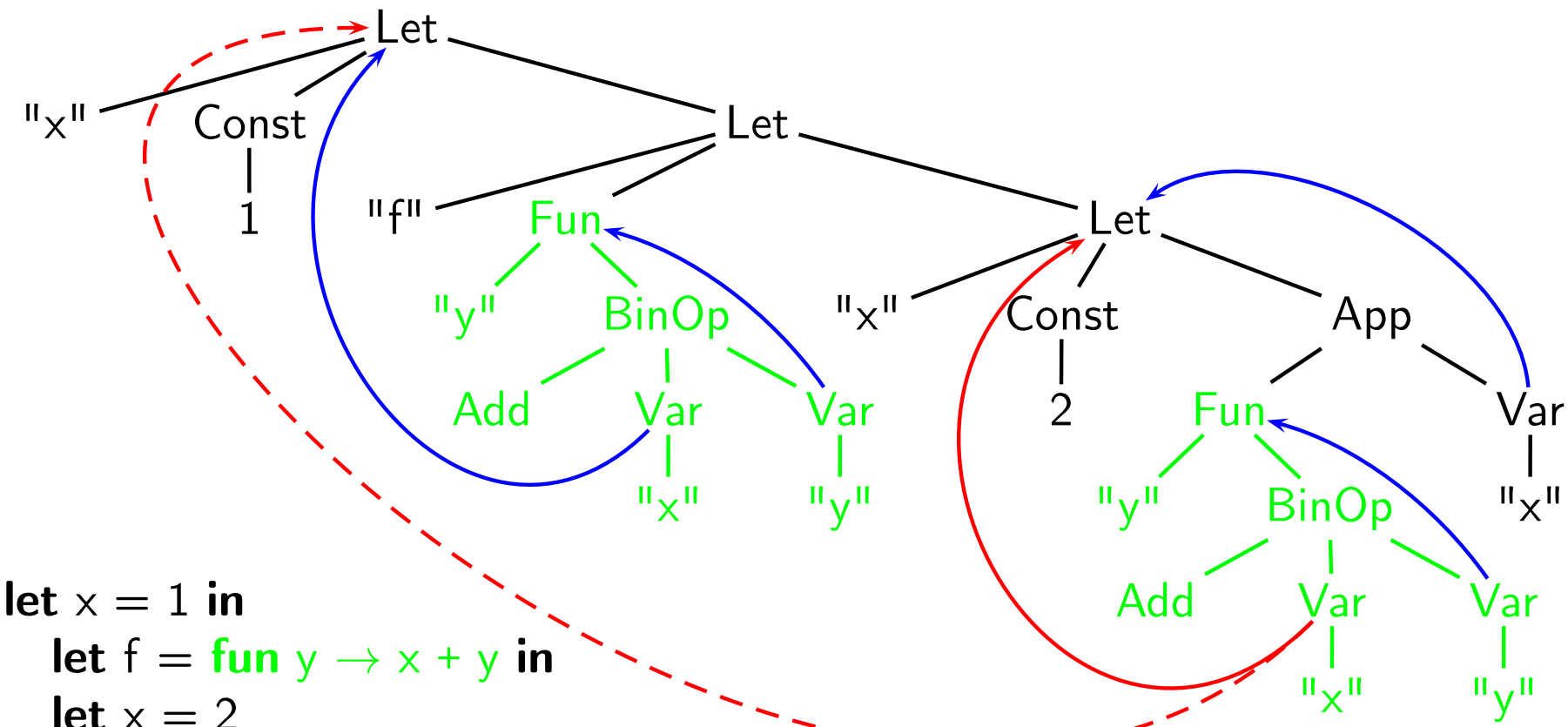
```
let x = 1 in
   let f = fun y → x + y in
   let x = 2
in (fun y → x + y) x
```

Arbre de syntaxe abstraite du premier programme




```
let x = 1 in
  let f = fun y → x + y in    (* Ce x est lié à 1. *)
  let x = 2
in f x    (* Ce x est lié à 2. *)
```

Arbre de syntaxe du second programme




```
let x = 1 in
   let f = fun y → x + y in
   let x = 2
in (fun y → x + y) x    (* Les deux x sont liés à 2.  *)
```

Capture de variable, liaison statique et dynamique

On dit que la variable x sous le **fun** a été *capturée* (par le troisième **let**). En d'autre termes, avec notre sémantique abs-dyn, la valeur d'une variable peut changer au cours de l'évaluation (selon l'environnement courant) : c'est la *liaison dynamique*.

Peu de langages l'emploient (Lisp, macros C), car les programmes sont alors plus difficiles à comprendre et à maintenir.

En général on préfère la *liaison statique* (dite aussi *lexicale*) : la valeur des variables libres dans le corps des fonctions est figée au moment de la définition. Le premier programme s'évaluerait alors en 3 et le second en 4.

Nous devons donc trouver une sémantique de l'abstraction qui respecte la transparence référentielle et la liaison statique.

Sémantique opérationnelle de l'application

Si les fonctions sont des valeurs, elles peuvent être retournées en résultat (c.-à-d. être la valeur d'une application). Par exemple :

```
let add = fun x → fun y → x + y in
  let incr = add (1)   (* incr est une fonction. *)
in incr (5)
```

On parle d'application *partielle* (par opposition à application *complète*, qui ne retourne pas de fonction, comme add 1 5).

Nous devons donc trouver une sémantique de l'application qui permette l'application partielle.

Fermetures, erreurs de typage

La solution générale aux contraintes posées par l'abstraction et l'application consiste à évaluer les fonctions en des *fermetures*.

Une fermeture $\mathrm{Clos}\,(x, e, \rho)$ est formée à partir d'une expression fonctionnelle $\mathrm{Fun}\,(x, e)$ et d'un environnement ρ. Il faut donc redéfinir le type des valeurs

type *value* = Int **of** *int* | Clos **of** *string* $*$ *expr* $*$ (*string* $\rightarrow$ *value*);;

et aussi l'implantation de la fonction OCaml eval.

Remarque Désormais, les expressions peuvent être incohérentes (on parle d'*erreurs de typage*) : il faut s'assurer qu'on n'opère que sur des entiers, et qu'on n'appelle que des fonctions.

Sémantique opérationnelle de l'abstraction et de l'application

$$\rho \vdash \langle\langle \textbf{fun}\ x \to \overline{e} \rangle\rangle \twoheadrightarrow \mathsf{Clos}\,(x, e, \rho) \quad \text{abs}$$

$$\frac{\rho \vdash e_1 \twoheadrightarrow \mathsf{Clos}\,(x_0, e_0, \rho_0) \qquad \rho \vdash e_2 \twoheadrightarrow v_2 \qquad \rho_0 \oplus x_0 \mapsto v_2 \vdash e_0 \twoheadrightarrow v_0}{\rho \vdash \langle\langle \overline{e}_1\ \overline{e}_2 \rangle\rangle \twoheadrightarrow v_0}\ \text{app}$$

Sémantique opérationnelle de l'abstraction et de l'application (suite)

Remarques

- Dans une fermeture $\mathrm{Clos}\,(x, e, \rho)$ on peut restreindre l'environnement ρ aux variables libres de la fonction $\mathrm{Fun}\,(x, e)$:

$$\rho \vdash \langle\langle \textbf{fun}\ x \to \overline{e} \rangle\rangle\ \textbf{as}\ f \twoheadrightarrow \mathrm{Clos}\,(x, e, \rho|\mathcal{L}(f)) \quad \text{abs-opt}$$

- L'implantation devrait évaluer e_1 avant e_2 afin de vérifier d'abord que e_1 s'évalue bien en une fermeture (si ce n'est pas le cas, on gagne du temps en signalant l'erreur au plus tôt).

Stratégies d'appel

Dans la règle de l'application (c.-à-d. appel de fonction), l'argument e_2 est d'abord évalué en v_2, et cette valeur est passée ensuite à la fermeture résultant de l'évaluation de e_1. C'est la stratégie d'*appel par valeur*, employée dans des langages comme OCaml et Java. On parle aussi de *stratégie stricte*.

D'autres langages, comme Haskell, emploient une stratégie dite d'*appel par nom*, qui consiste à passer l'argument *non évalué* à la fonction : il ne sera évalué que s'il est nécessaire au calcul du résultat de la fonction. On parle aussi de *stratégie paresseuse*.

Une optimisation de l'appel par nom est l'*appel par nécessité* qui ne recalcule pas deux fois le même appel (les résultats sont mémorisés). On parle aussi de *stratégie pleinement paresseuse*.

Non-terminaison

En théorie, nous pouvons d'ores et déjà calculer avec notre calculette tout ce qui est calculable avec l'ordinateur sous-jacent. Par exemple, nous avons déjà la récurrence, comme le montre le programme suivant qui ne termine pas :

$$\textbf{let}\ \text{omega} = \textbf{fun}\ \text{f} \to \text{f}\ \text{f}\ \textbf{in}\ \text{omega}\ (\text{omega})$$

Notre style de sémantique qui évalue directement une expression en sa valeur n'est pas pratique pour étudier la non-terminaison. Pour le programme précédent, cela se manifesterait par l'occurrence dans la dérivation de

$$\frac{\rho \vdash \langle\!\langle \text{f} \rangle\!\rangle \twoheadrightarrow v_1 \qquad \rho \oplus \text{"f"} \mapsto v_1 \vdash \langle\!\langle \text{f f} \rangle\!\rangle \twoheadrightarrow v}{\rho \vdash \langle\!\langle \text{f f} \rangle\!\rangle \twoheadrightarrow v}$$

Le première prémisse dit que $\rho(\text{"f"}) = v_1$, donc $\rho = \rho \oplus \text{"f"} \mapsto v_1$, donc la conclusion et la seconde prémisse sont identiques, donc le calcul de v boucle.

Turing-complétude

Un langage muni de la conditionnelle et de la récurrence (ou seulement du branchement conditionnel) permet de spécifier tous les calculs qui sont possibles par l'ordinateur sous-jacent : on le dit *Turing-complet*.

Cette propriété est très utile mais elle implique nécessairement l'existence de programmes qui ne terminent pas (pour certaines données) et l'inexistence de programmes permettant de les reconnaître tous (*incomplétude de Gödel*). L'idée est la suivante.

Turing-complétude (suite)

Raisonnons par l'absurde.

Supposons donc l'existence d'un prédicat de terminaison. Soit f la fonction telle que pour toute fonction g, si g termine toujours (c.-à-d. $\forall x.g(x)$ est défini) alors $f(g)$ ne termine pas, sinon $f(g)$ termine. Dans ce cas, $f(f)$ ne termine pas si f termine, et $f(f)$ termine si f ne termine pas, ce qui est contradictoire. Donc un tel prédicat n'existe pas.

Problèmes indécidables

On dit que le problème de la terminaison (ou de l'*arrêt de la machine de Turing*) est *indécidable*.

La négation de tout problème indécidable est indécidable aussi.

Il est important de connaître quelques-uns de ces problèmes car ils n'ont *théoriquement* pas de solution en général.

Problèmes indécidables (suite)

Pour les programmes des langages Turing-complets, sont indécidables

- la terminaison ;
- la valeur d'une variable à un moment donné de l'exécution (en particulier savoir si elle est initialisée ou non, si le problème se pose dans le langage considéré — comparez OCaml, C et Java) ;
- l'appel d'une fonction (en particulier le problème du *code mort*).

Remarque On peut parfois résoudre ces problèmes sur des cas particuliers.

Les termes non-clos revus

Nous avons présenté une analyse statique (p. 49) qui nous donne les variables libres d'une expression. Nous avons vu qu'une expression close ne peut échouer par absence de liaison. Tous les compilateurs (comme OCaml) rejettent les programmes non-clos, mais, du coup, rejettent d'innocents programmes, comme **if true then** 1 **else** x.

Pour accepter ce type d'exemple (non-clos), il faudrait pouvoir prédire le flot de contrôle (ici, quelle branche de la conditionnelle est empruntée pour toutes les exécutions). Dans le cas ci-dessus cela est trivial, mais en général le problème est indécidable, et ce ne peut donc être une analyse statique (car la compilation doit toujours terminer).

Fonctions récursives

Pour mettre en évidence la puissance de mini-ML, voyons comment définir des fonctions récursives à l'aide de la fonction auto-applicative omega.

Définissons d'abord une fonction fix, traditionnellement appelée le *combinateur Y de point fixe de Curry* :

```
let omega = fun f → f f in
  let fix = fun g → omega (fun h → fun x → g (h h) x) in
  ...
```

Point fixe d'une fonction

On démontre (péniblement) que l'évaluation de (fix f x) a la forme :

$$\dfrac{\dots \qquad \rho \vdash \langle\!\langle \mathsf{f}\ (\mathsf{fix}\ \mathsf{f})\ \mathsf{x} \rangle\!\rangle \twoheadrightarrow v}{\dfrac{\dots \qquad \vdots}{\rho \vdash \langle\!\langle (\mathsf{fix}\ \mathsf{f})\ \mathsf{x} \rangle\!\rangle \twoheadrightarrow v}}$$

En d'autre termes, pour tout x on a (fix f) x = f (fix f) x, soit (fix f) = f (fix f). D'autre part, par définition, le point fixe p d'une fonction f vérifie $p = f(p)$.

Donc le point fixe d'une fonction f, *s'il existe*, est (fix f).

Factorielle et cas général

Posons

let pre_fact = **fun** f $\rightarrow$ **fun** n $\rightarrow$ **ifz** n **then** 1 **else** n $*$ f (n-1) **in**
let fact = fix (pre_fact) **in** ...

Donc fact est le point fixe de pre_fact, s'il existe, c'est-à-dire

fact = pre_fact (fact) = **fun** n $\rightarrow$ **ifz** n **then** 1 **else** n $*$ fact (n-1)

Donc fact est la fonction factorielle (équation de récurrence).

On peut donc prédéfinir un opérateur de point fixe fix (qui n'est pas forcément celui de Curry) et permettre au programmeur de s'en servir directement.

Liaison locale récursive

Pour plus de commodité, étendons la syntaxe avec une liaison locale récursive.

- Syntaxe concrète

```
Expression ::= ... |
  "let" "rec" ident "=" Expression "in" Expression
```

- Syntaxe abstraite

  type *expr* = ... | LetRec **of** *string* * *expr* * *expr*;;

- Analyse syntaxique

  $\langle\!\langle \textbf{let rec } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle = \text{LetRec}\,(x, \langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle)$

- Variables libres

  $\mathcal{L}\langle\!\langle \textbf{let rec } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle = (\mathcal{L}(e_1) \cup \mathcal{L}(e_2)) \backslash \{x\}$

Sémantique opérationnelle de la liaison locale récursive

On peut définir la sémantique de cette construction de deux façons. La première consiste à ne pas la considérer comme élémentaire (ou *native*) et exprimer sa sémantique en fonction de celle d'une autre construction, en l'occurrence (en supposant que l'opérateur fix est prédéfini dans l'interprète) :

$$\frac{\rho \vdash \langle\!\langle \textbf{let } x = \text{fix } (\textbf{fun } x \to \overline{e}_1) \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v}{\rho \vdash \langle\!\langle \textbf{let rec } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v} \text{ let-rec}$$

La seconde consiste à considérer cette construction comme différente des autres :

$$\frac{\rho \oplus x \mapsto v_1 \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \oplus x \mapsto v_1 \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \textbf{let rec } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_2} \text{ let-rec}$$

Interprétation de la liaison locale récursive native

L'implantation immédiate des expressions récursives natives est

```
let rec eval env e = match e with ...
|   LetRec (x,e1,e2) →
      let rec env' = extend env (x,v1)      (* ρ ⊕ x ↦ v1 *)
      and v1 = eval env' e1                 (* ρ ⊕ x ↦ v1 ⊢ e1 ↠ v1 *)
      in eval env' e2                       (* ρ ⊕ x ↦ v1 ⊢ e2 ↠ v2 *)
```

Pour des raisons techniques liées au typage de OCaml, il faut écrire en fait

```
let rec eval env e = match e with ...
|   LetRec (x,e1,e2) →
      let rec env' = fun x → extend env (x, v1()) x
      and v1 = fun () → eval env' e1
      in eval env' e2
```

Les expressions mutuellement récursives

Le **let rec** multiple (avec **and**) peut toujours se ramener à un **let rec** simple (avec **in**) en paramétrant l'une des définitions par rapport à l'autre. Posons

$$\textbf{let rec } x = \overline{e}_1 \textbf{ and } y = \overline{e}_2 \textbf{ in } \overline{e}$$

où $x \neq y$, est équivalent (par définition) à

$$\begin{array}{l}\textbf{let rec } x = \textbf{fun } y \rightarrow \overline{e}_1 \textbf{ in} \\ \quad \textbf{let rec } y = \textbf{let } x = x\ (y) \textbf{ in } \overline{e}_2 \textbf{ in} \\ \quad \textbf{let } x = x\ (y) \\ \textbf{in } \overline{e}\end{array}$$

On peut ensuite coder les **let rec** simples avec fix ou les considérer natifs dans le langage interprété. Dans les deux cas, il n'y a pas besoin d'étendre la sémantique opérationnelle. Il faut néanmoins penser à généraliser notre équivalence syntaxique à n variables : **let rec** $x_1 = \overline{e}_1$ **and** $x_2 = \overline{e}_2$ **and** $\ldots$ **and** $x_n = \overline{e}_n$ **in** $\overline{e}$

Les expressions parallèles

Nous pouvons aussi ajouter à notre langage la construction

$$\textbf{let } x = \overline{e}_1 \textbf{ and } y = \overline{e}_2 \textbf{ in } \overline{e} \quad \text{où } x \neq y$$

Si $x \in \mathcal{L}(e_2)$, nous la définissons comme étant équivalente à

$$\begin{array}{l} \textbf{let } z = x \textbf{ in} \\ \quad \textbf{let } x = \overline{e}_1 \textbf{ in} \\ \quad \textbf{let } y = \textbf{let } x = z \textbf{ in } \overline{e}_2 \\ \textbf{in } \overline{e} \end{array}$$

où $z \notin \mathcal{L}(e_1) \cup \mathcal{L}(e_2) \cup \mathcal{L}(e)$, pour n'être capturé ni par e_1, ni par e_2, ni par e.

Il n'y a donc pas besoin d'étendre la sémantique opérationnelle pour traiter cette construction : une équivalence entre arbres de syntaxe abstraite suffit pour donner le sens. Il faut néanmoins penser à généraliser l'équivalence :

$\textbf{let } x_1 = \overline{e}_1 \textbf{ and } x_1 = \overline{e}_2 \textbf{ and } \ldots \textbf{ and } x_n = \overline{e}_n \textbf{ in } \overline{e}$

Plus loin avec le rasoir d'Occam

En observant les règles let, d'une part, et abs et app d'autre part, on peut se rendre compte que la règle let peut être supprimée sans conséquence sur l'expressivité du langage. Plus précisément, nous allons prouver que les constructions **let** $x = \overline{e}_1$ **in** $\overline{e}_2$ et (**fun** $x \to \overline{e}_2$) $\overline{e}_1$ sont équivalentes du point de vue de l'évaluation — c'est-à-dire que l'une produit une valeur v si et seulement si l'autre produit également v.
La règle app (p. 63) peut se réécrire en intervertissant e_1 et e_2 :

$$\frac{\rho \vdash e_2 \twoheadrightarrow \mathsf{Clos}\,(x_0, e_0, \rho_0) \qquad \rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho_0 \oplus x_0 \mapsto v_1 \vdash e_0 \twoheadrightarrow v_0}{\rho \vdash \langle\!\langle \overline{e}_2\ \overline{e}_1 \rangle\!\rangle \twoheadrightarrow v_0}$$

Une preuve d'équivalence (suite)

En substituant $\mathbf{fun}\ x \to \overline{e}_2$ à $\overline{e}_2$ il vient

$$\frac{\rho \vdash \langle\!\langle \mathbf{fun}\ x \to \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathrm{Clos}\,(x_0, e_0, \rho_0) \qquad \rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho_0 \oplus x_0 \mapsto v_1 \vdash e_0 \twoheadrightarrow v_0}{\rho \vdash \langle\!\langle (\mathbf{fun}\ x \to \overline{e}_2)\ \overline{e}_1 \rangle\!\rangle \twoheadrightarrow v_0}$$

Or l'axiome abs dit $\rho \vdash \langle\!\langle \mathbf{fun}\ x \to \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathrm{Clos}\,(x, e_2, \rho)$ donc $x = x_0$, $e_2 = e_0$ et $\rho = \rho_0$, d'où, en remplaçant dans la pénultième règle et en renommant v_0 en v_2 :

$$\frac{\rho \vdash \langle\!\langle \mathbf{fun}\ x \to \overline{e}_2 \rangle\!\rangle \twoheadrightarrow \mathrm{Clos}\,(x, e_2, \rho) \qquad \rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \oplus x \mapsto v_1 \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle (\mathbf{fun}\ x \to \overline{e}_2)\ \overline{e}_1 \rangle\!\rangle \twoheadrightarrow v_2}$$

Une preuve d'équivalence (suite et fin)

Un axiome étant par définition vrai, on peut le supprimer d'une prémisse :

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \oplus x \mapsto v_1 \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle (\mathbf{fun}\ x \to \overline{e}_2)\ \overline{e}_1 \rangle\!\rangle \twoheadrightarrow v_2}$$

Or la règle let (p. 36) est :

$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \oplus x \mapsto v_1 \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \mathbf{let}\ x = \overline{e}_1\ \mathbf{in}\ \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_2}\ \text{let}$$

Les prémisses sont les mêmes que dans la règle précédente, donc les conclusions sont identiquement vraies [QED]. Il est donc en théorie possible de se passer de la liaison locale dans notre langage, que ce soit au niveau de la sémantique, de la syntaxe abstraite ou concrète.

Discussion sur les constructions non-élémentaires

De façon générale, lorsqu'on découvre qu'une construction a la même sémantique qu'une combinaison d'autres constructions (pp. 78,79,80), il vaut mieux conserver cette construction dans la syntaxe abstraite car cela permet d'y adjoindre les positions des lexèmes correspondants dans le code source, pour l'éventualité d'un message d'erreur. En effet, l'autre solution, qui consiste à produire l'AST de la combinaison lors de l'analyse syntaxique, perd cette information.

Il est utile parfois de simplifier la sémantique. Par exemple, ici, définir :

$$\frac{\rho \vdash \langle\!\langle (\textbf{fun}\ x \to\ \overline{e}_2)\ \overline{e}_1 \rangle\!\rangle \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \textbf{let}\ x = \overline{e}_1\ \textbf{in}\ \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_2}\ \text{let}$$

Programmation impérative

Jusqu'à présent les variables méritaient mal leur nom pour des raisons historiques, car elles dénotaient des constantes ou des fonctions. Nous allons rendre les variables vraiment variables, c'est-à-dire *mutables* dans le jargon de OCaml.

Le style de programmation qui fait usage de telles variables se nomme *impératif*.

Bien que nous souhaitons rendre toutes les variables mutables (pour des raisons d'uniformité de présentation ici), nous souhaitons néanmoins distinguer par la syntaxe concrète leur modification impérative par le truchement d'*affectations*.
Exemple **let** x = 1 **in let** y = (x := x + 2) **in** x s'évalue en Int (3).
L'affectation est ici x := x + 2

Un modèle fonctionnel pour les variables mutables

Pour modéliser de façon fonctionnelle ce nouveau paradigme de programmation, nous devons introduire les notions d'*adresse* et de *mémoire*. Une adresse est un élément d'un ensemble infini dénombrable. Une mémoire σ lie les adresses a aux valeurs v. Un environnement ρ lie maintenant une variable x à son adresse a, *et non plus à sa valeur directement*.

C'est ainsi que l'on peut cacher une liaison par une autre dans la mémoire sans changer l'environnement, ce qui modélise l'affectation de façon fonctionnelle. Nous n'avons donc pas besoin d'une notion native pour l'affectation dans la sémantique. Quant à l'interprète, s'il est écrit dans un langage fonctionnel, ce langage n'est pas contraint de posséder non plus d'une notion d'affectation native.

Instructions *versus* expressions

On peut ensuite soit introduire la notion d'*instruction* (C, Java), soit rester avec celle d'expression (OCaml). Dans ce dernier cas, il nous faut ajouter une valeur spéciale qui est le résultat de l'évaluation d'une *affectation* : la valeur Unit.

Exemple $\mathsf{x} := 2$ s'évalue en Unit.

Par souci de généralité et de commodité, il est bon d'ajouter aussi une expression qui s'évalue immédiatement en Unit. Nous suivrons la syntaxe de OCaml pour la noter ().

Exemple $\textbf{let}\ \mathsf{f} = \textbf{fun}\ \mathsf{x} \to 1\ \textbf{in}\ \mathsf{f}\ ()$

Attention, il faut distinguer la *valeur* Unit et l'*expression* correspondante, que nous noterons U dans la syntaxe abstraite.

Affectation et sémantique avec mutables

- Syntaxe concrète

  `Expression ::= ... | ident ":=" Expression | ()`
- Syntaxe abstraite

  type *expr* = ... | Assign **of** *string* * *expr* | U;;
- Analyse syntaxique

  $\langle\!\langle x := \overline{e} \rangle\!\rangle = \mathsf{Assign}\,(x, \langle\!\langle \overline{e} \rangle\!\rangle)$ et $\langle\!\langle () \rangle\!\rangle = \mathsf{U}$
- Variables libres

  $\mathcal{L}\langle\!\langle x := \overline{e} \rangle\!\rangle = \{x\} \cup \mathcal{L}(e)$ et $\mathcal{L}\langle\!\langle () \rangle\!\rangle = \varnothing$
- Sémantique opérationnelle

  Il nous faut revoir notre jugement et nos règles d'inférence. Soit maintenant $\rho/\sigma \vdash e \twoheadrightarrow v/\sigma'$, se lisant « Dans l'environnement ρ et la mémoire σ, l'évaluation de e produit une valeur v et une nouvelle mémoire σ'. »

Sémantique avec mutables (unité et variables)

Voici d'abord les règles d'inférence les plus simples (comparez var avec p. 36) :

$$\rho/\sigma \vdash \langle\!\langle () \rangle\!\rangle \twoheadrightarrow \mathsf{Unit}/\sigma \quad \text{unit} \qquad \frac{x \in \mathrm{dom}(\rho) \qquad \rho(x) \in \mathrm{dom}(\sigma)}{\rho/\sigma \vdash \langle\!\langle x \rangle\!\rangle \twoheadrightarrow \sigma \circ \rho(x)/\sigma} \text{ var}$$

Pour accéder au contenu d'une variable il faut donc passer via l'environnement puis la mémoire, c'est pour cela qu'il faut s'assurer que $x \in \mathrm{dom}(\rho)$ et $\rho(x) \in \mathrm{dom}(\sigma)$. Généralement, notre nouveau jugement $\rho/\sigma \vdash e \twoheadrightarrow v/\sigma'$ doit vérifier la propriété : *si* $\mathrm{codom}(\rho) \subseteq \mathrm{dom}(\sigma)$ *alors* $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$, c.-à-d. que l'évaluation peut occulter une liaison par une autre en mémoire, ou en ajouter de nouvelles, mais pas en retirer. Autrement dit encore, lorsque l'évaluation termine, les variables ont toujours une valeur mais qui a pu changer.

Preuve par récurrence sur la longueur des dérivations

La sémantique opérationnelle permet un type de preuve dite par récurrence généralisée sur la longueur des dérivations (ou arbres de preuves).

On vérifie la propriété à démontrer sur les axiomes (dérivations de longueur 1), puis on suppose que la propriété est vraie pour toutes les dérivations de longueur $n-1$, puis on prouve finalement qu'elle est vraie pour toutes les dérivations de longueur n en examinant toutes les règles et en imaginant qu'elles sont la racine d'une preuve de longueur n. Ainsi, les prémisses vérifient la propriété à démontrer par hypothèse de récurrence, car leur dérivation est de taille strictement inférieure à n.

Considérons ici la propriété suivante. *Soit* $\rho/\sigma \vdash e \twoheadrightarrow v/\sigma'$. *Si* $\mathrm{codom}(\rho) \subseteq \mathrm{dom}(\sigma)$ *alors* $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$. Elle est vraie sur tous les axiomes.

Sémantique avec mutables (affectation)

L'affectation masque la liaison de la variable concernée en mémoire :

$$\frac{\rho/\sigma \vdash e \twoheadrightarrow v/\sigma' \qquad x \in \mathrm{dom}(\rho) \qquad \rho(x) \in \mathrm{dom}(\sigma')}{\rho/\sigma \vdash \langle\!\langle x := \overline{e} \rangle\!\rangle \twoheadrightarrow \mathsf{Unit}/(\sigma' \oplus \rho(x) \mapsto v)}\ \text{assign}$$

Cette règle vérifie la propriété de la page précédente. En effet, par hypothèse de récurrence (sur la taille de la dérivation), la première prémisse implique que si $\mathrm{codom}(\rho) \subseteq \mathrm{dom}(\sigma)$ alors $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma')$. La troisième prémisse implique alors $\mathrm{dom}(\sigma' \oplus \rho(x) \mapsto v) = \mathrm{dom}(\sigma')$, donc $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma' \oplus \rho(x) \mapsto v)$.
Nous avons besoin de σ' car il se peut que l'évaluation de e masque des liaisons de σ autres que $\rho(x) \mapsto v$ et il faut conserver ces occultations.
Exemple **let** x = 1 **in** **let** y = 2 **in** **let** z = (x := (y := 3)) **in** y s'évalue en Int (3).

Sémantique avec mutables (liaison locale)

La règle let ajoute une nouvelle liaison dans l'environnement et la mémoire en respectant l'invariant cité p. 88 :

$$\frac{\rho/\sigma \vdash e_1 \twoheadrightarrow v_1/\sigma_1 \qquad a \notin \mathrm{dom}(\sigma_1) \qquad (\rho \oplus x \mapsto a)/(\sigma_1 \oplus a \mapsto v_1) \vdash e_2 \twoheadrightarrow v_2/\sigma_2}{\rho/\sigma \vdash \langle\!\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_2/\sigma_2}\ \text{let}$$

En effet, si $\mathrm{codom}(\rho) \subseteq \mathrm{dom}(\sigma)$ alors, par hypothèse de récurrence (sur la longueur de la dérivation de la première prémisse), $\mathrm{dom}(\sigma) \subseteq \mathrm{dom}(\sigma_1)$, donc, transitivement, $\mathrm{codom}(\rho) \subseteq \mathrm{dom}(\sigma_1)$, d'où $\mathrm{codom}(\rho \oplus x \mapsto a) \subseteq \mathrm{dom}(\sigma_1 \oplus a \mapsto v_1)$. Par hypothèse de récurrence (troisième prémisse), cela implique alors que $\mathrm{dom}(\sigma_1 \oplus a \mapsto v_1) \subseteq \mathrm{dom}(\sigma_2)$. Or $a \notin \mathrm{dom}(\sigma_1)$ implique $\mathrm{dom}(\sigma_1) \subset \mathrm{dom}(\sigma_1 \oplus a \mapsto v_1)$, donc finalement $\mathrm{dom}(\sigma) \subset \mathrm{dom}(\sigma_2)$. *En particulier, il n'y a pas égalité.*

Glânage de cellules (*Garbage Collection*)

Lorsque nous avons introduit les environnements, nous avons vu que la liaison locale pouvait masquer des liaisons. L'introduction des mémoires nous montre que la liaison locale peut masquer et ajouter des liaisons en mémoire qui ne sont pas accessibles à partir de l'environnement. Dans les deux cas, des liaisons dans l'environnement ou la mémoire deviennent définitivement inaccessibles, donc l'espace mémoire qu'occupent les valeurs correspondantes est gâché pour le reste de l'exécution (ou interprétation).

C'est pourquoi les compilateurs de certains langages (OCaml, Java, Ada) produisent un code effectuant dynamiquement (selon diverses stratégies) une analyse d'accessibilité des données et restituant au processus sous-jacent la mémoire virtuelle correspondant aux cellules inatteignables : c'est le *glâneur de cellules*, ou *garbage collector*.

Sémantique avec mutables (expressions arithmétiques)

On doit fixer l'ordre d'évaluation des arguments des opérateurs arithmétiques dans la sémantique pour conserver en mémoire les *effets* des affectations qui ont éventuellement eu lieu lors de l'évaluation des arguments.

$$\rho/\sigma \vdash \langle\langle \overline{n} \rangle\rangle \twoheadrightarrow \dot{n}/\sigma \quad \text{const}$$

$$\frac{\rho/\sigma \vdash e_1 \twoheadrightarrow v_1/\sigma_1 \qquad \rho/\sigma_1 \vdash e_2 \twoheadrightarrow v_2/\sigma_2}{\rho/\sigma \vdash \langle\langle \overline{e}_1 + \overline{e}_2 \rangle\rangle \twoheadrightarrow v_1 + v_2/\sigma_2} \quad \text{add}$$

Il est néanmoins toujours possible de nier officiellement, c'est-à-dire dans la documentation livrée avec l'interprète, que l'ordre n'est pas fixé, laissant ainsi le loisir de modifier l'ordre à des fins d'optimisation de l'interprète.

Les autres expressions arithmétiques suivent le même schéma.

Sémantique avec mutables (abstraction et application)

$$\rho/\sigma \vdash \langle\!\langle \textbf{fun}\ x \to \overline{e} \rangle\!\rangle \twoheadrightarrow \mathsf{Clos}\,(x, e, \rho)/\sigma \quad \text{abs}$$

$$\frac{\rho/\sigma \vdash e_1 \twoheadrightarrow \mathsf{Clos}\,(x_0, e_0, \rho_0)/\sigma_1 \qquad \rho/\sigma_1 \vdash e_2 \twoheadrightarrow v_2/\sigma_2 \qquad a \notin \mathrm{dom}(\sigma_2) \qquad (\rho_0 \oplus x_0 \mapsto a)/(\sigma_2 \oplus a \mapsto v_2) \vdash e_0 \twoheadrightarrow v_0/\sigma_3}{\rho/\sigma \vdash \langle\!\langle \overline{e}_1\ \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_0/\sigma_3}\ \text{app}$$

Remarques

- Par rapport à la page 63, l'ordre d'évaluation est ici fixé dans la sémantique : e_1 avant e_2, afin de vérifier d'abord que e_1 s'évalue bien en une fermeture (si ce n'est pas le cas, on gagne du temps en signalant l'erreur au plus tôt).
- Le fait que $a \notin \mathrm{codom}(\rho)$ permet d'oublier les affectations sur le paramètre, p.ex. **let** f = **fun** x $\to$ x := x + 1 **in** f 3.

Sémantique avec mutables (expressions récursives natives)

$$\frac{\begin{array}{c} a \notin \mathrm{dom}(\sigma) \\ (\rho \oplus x \mapsto a)/(\sigma \oplus a \mapsto v_1) \vdash e_1 \twoheadrightarrow v_1/\sigma_1 \\ (\rho \oplus x \mapsto a)/\sigma_1 \vdash e_2 \twoheadrightarrow v_2/\sigma_2 \end{array}}{\rho/\sigma \vdash \langle\!\langle \textbf{let rec } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_2/\sigma_2} \text{ let-rec}$$

La séquence et l'itération générale

On peut maintenant trouver avantage à ajouter à notre langage la séquence et l'itération générale :

- Syntaxe concrète

```
Expression ::= ...
             | Expression ";" Expression
             | "while" Expression "do" Expression "done"
```

- Syntaxe abstraite

  type *expr* = ... | Seq **of** *expr* * *expr* | While **of** *expr* * *expr*;;

- Analyse syntaxique

  $\langle\!\langle \overline{e}_1; \overline{e}_2 \rangle\!\rangle = \mathsf{Seq}\,(\langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e}_2 \rangle\!\rangle)$ et
  $\langle\!\langle \textbf{while}\ \overline{e}_1\ \textbf{do}\ \overline{e}_2\ \textbf{done} \rangle\!\rangle = \mathsf{While}\,(\langle\!\langle \overline{e}_1 \rangle\!\rangle, \langle\!\langle \overline{e_2} \rangle\!\rangle)$

- Variables libres

  $\mathcal{L}\langle\!\langle \overline{e}_1; \overline{e}_2 \rangle\!\rangle = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$ et
  $\mathcal{L}\langle\!\langle \textbf{while}\ \overline{e}_1\ \textbf{do}\ \overline{e}_2\ \textbf{done} \rangle\!\rangle = \mathcal{L}(e_1) \cup \mathcal{L}(e_2)$

Sémantique de la séquence et de l'itération générale

$$\frac{x \notin \mathcal{L}(e_2) \qquad \rho \vdash \langle\!\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v}{\rho \vdash \langle\!\langle \overline{e}_1 ; \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v} \text{ seq}$$

while

$$\frac{f, p \notin \mathcal{L}(e_2) \qquad x \notin \mathcal{L}(e_1) \qquad \rho \vdash \langle\!\langle \textbf{let rec } f = \textbf{fun } p \to \textbf{if } p() \textbf{ then } \overline{e}_2 ; f\ p \textbf{ else } () \textbf{ in } f(\textbf{fun } x \to \overline{e}_1) \rangle\!\rangle \twoheadrightarrow v}{\rho \vdash \langle\!\langle \textbf{while } \overline{e}_1 \textbf{ do } \overline{e}_2 \textbf{ done} \rangle\!\rangle \twoheadrightarrow v}$$

Remarque La valeur v est en fait toujours () dans la règle while.

Si on ajoutait les opérateurs booléens de comparaison sur les entiers, on pourrait enfin écrire **let** x = 0 **in** (**while** x < 10 **do** x := x + 1 **done**; x)

Implantation de la sémantique avec mutables

Le premier choix à faire est celui du type de donnée OCaml qui convient pour les adresses. Bien que les entiers OCaml peuvent déborder, on les utilisera ici comme implantation des adresses.
Un autre point concerne les règles où il faut choisir une adresse qui est libre dans la mémoire, formellement $a \notin \mathrm{dom}(\sigma)$. Une solution consiste à produire une adresse à chaque fois qui soit absolument unique à partir d'un incrément entier (les adresses étant implantées par des entiers). Il faudrait donc passer un argument supplémentaire à notre fonction d'évaluation eval, qui est le compteur, et à chaque fois qu'on a besoin d'une nouvelle adresse on lui ajoute 1.
Les fonctions polymorphes (extend pour l'ajout d'une liaison et lookup pour la recherche d'une liaison) qui opérent sur les environnements servent aussi pour les mémoires.