Interprétation et compilation

Christian Rinderknecht

25 October 2008

Prérequis et sources

Ce cours suppose que vous ayez compris les cours

- algorithmique et programmation,
- programmation fonctionnelle en Objective Caml.

Ce cours s'inspire de ceux de

- Martin Odersky (École Polytechnique Fédérale de Lausanne),
- Luc Maranget, Didier Rémy (INRIA, École Polytechnique),
- François Pottier, Xavier Leroy et Michel Mauny (INRIA).

Pourquoi étudier les compilateurs ?

Très peu de professionnels écrivent des compilateurs.

Alors pourquoi apprendre à construire des compilateurs ?

- Un bon informaticien comprend les langages de haut niveau ainsi que le matériel.
- Un compilateur relie ces deux aspects.
- C'est pourquoi comprendre les techniques de compilation c'est comprendre l'interaction entre les langages de programmation et les ordinateurs.
- Beaucoup d'applications contiennent de petits langages pour leur configuration ou rendre souple leur contrôle (macros Word, scripts pour le graphisme et l'animation, les descriptions des structures de données etc.)

Pourquoi étudier les compilateurs ? (suite)

- Les techniques de compilation sont nécessaires pour l'implantation de tels langages.
- Les formats de données sont aussi des langages formels (langages de spécification de données), tels que HTML, XML, ASN.1 etc.
- Les techniques de compilation sont nécessaires pour lire, traiter et écrire des données, mais aussi pour migrer des applications (réingénierie).
- À part cela, les compilateurs sont d'excellents exemples de grands systèmes complexes
  - qui peuvent être spécifiés rigoureusement,
  - qui ne peuvent être réalisés qu'en combinant théorie et pratique.

Le rôle d'un compilateur

- Le rôle d'un compilateur est de traduire des textes d'un langage *source* en un langage *cible*.
- Souvent le langage source est plus abstrait (p.ex. langage de programmation) que le langage cible (p.ex. assembleur).
- Néanmoins, on nomme parfois les compilateurs des *traducteurs* lorsqu'ils traduisent des programmes entre langages de même niveau d'abstraction.
- Une partie du travail d'un compilateur est de vérifier la validité du programme source.
- La spécification d'un compilateur est constituée par
  - une spécification des langages source et cible,
  - une spécification de la traduction des programmes de l'un vers l'autre.

Langages

- Formellement, un langage est un ensemble de *phrases*. Une phrase est une suite de *mots*. Un mot est une suite de *caractères* appartenant à un *alphabet* (ensemble de symboles fini non vide).
- Chaque phrase possède une structure qui peut être décrite par un arbre.
- Les règles de construction d'une phrase s'expriment à l'aide d'une *grammaire*.

Ainsi,

- les phrases d'un langage de programmation sont des programmes ;
- les mots d'un programmes sont appelés *lexèmes* ;
- les lexèmes suivent aussi des règles qui peuvent être données par une grammaire.

Structure simplifiée d'un compilateur

1. Analyse lexicale : texte source $\mapsto$ suite de lexèmes ;
2. Analyse syntaxique : suite de lexèmes $\mapsto$ arbre de syntaxe abstraite ;
3. Analyses sémantiques sur l'arbre de syntaxe abstraite :
   - 3.1 Vérification de la portée des identificateurs (gestion des environnements) ;
   - 3.2 Vérification ou inférence des types (optionel) : arbre $\mapsto$ arbre décoré.
4. Production de code intermédiaire : arbre [décoré ?] $\mapsto$ code intermédiaire ;
5. Optimisations intrinsèques du code intermédiaire ;
6. Production de code cible (objet), p.ex. assembleur ;
7. Optimisations du code cible (dépendantes de la machine cible) ;
8. Édition de liens (statique) : code cible + bibliothèques $\mapsto$ code exécutable.

Remarques

- L'analyseur lexical (*lexer*) reconnaît les espaces, les caractères de contrôle et les commentaires mais n'en fait pas des lexèmes (*tokens*).
- L'arbre de syntaxe abstraite est appelé *Abstract Syntax Tree* (AST).
- Les trois premières étapes constituent la *phase d'analyse*. Les restantes constituent la *phase de synthèse* (de code). On parle aussi de *phase frontale* pour les étapes jusqu'à la production de code intermédiaire incluse, et de *phase finale* pour les suivantes.
- L'association d'un type aux constructions du langage s'appelle le *typage*. Il garanti que les programmes ne provoqueront pas d'erreurs à l'exécution pour cause d'incohérence sur leurs données (néanmoins, une division par zéro restera possible). Selon la finesse du typage, plus ou moins de programmes valides sont rejetés.

Interprétation

Un *interprète* transforme un fichier source en une donnée, par exemple un arbre de syntaxe abstraite ou du code intermédiaire, que l'on passe ensuite à un programme, dit *machine virtuelle*, qui l'exécutera en mimant (donc abstraitement) une machine réelle (physique).

Remarques

- Non à l'horrible anglicisme « interpréteur » !
- Le code intermédiaire est parfois appelé *byte-code*.
- L'arbre de syntaxe abstraite n'est pas toujours construit, ou pas complètement, selon les langages ou les stratégies d'implantation.
- Un interprète contient donc les premières phases d'un compilateur.

Lexique et syntaxe

- L'ensemble des lexèmes d'un langage est appelé *lexique*.
- La *syntaxe concrète* décrit comment assembler les lexèmes en phrases pour constituer des programmes. En particulier,
  - elle ne donne pas de sens aux phrases ;
  - plusieurs notations sont possibles pour signifier la même chose, p.ex. en OCaml : 'a' et '\097', ou (...) et **begin** ... **end**.
  - ce qu'elle décrit est **linéaire** (le code source est du texte) et utilise généralement des parenthèses.
- La *syntaxe abstraite* décrit des **arbres** qui capturent la structure des programmes (p.ex. les imbrications correspondent à des sous-arbres).

Étude d'une calculette

- **Syntaxe concrète** (dans le style *Backus-Naur Form* (BNF))

```
Expression ::= integer
             | Expression BinOp Expression
             | "(" Expression ")"
BinOp ::= "+" | "-" | "*" | "/"
```

  Pour l'analyse syntaxique, les priorités des opérateurs est celle habituelle.

- **Syntaxe abstraite** (en OCaml)

```
type expr = Const of int
          | BinOp of bin_op * expr * expr
and bin_op = Add | Sub | Mult | Div;;
```

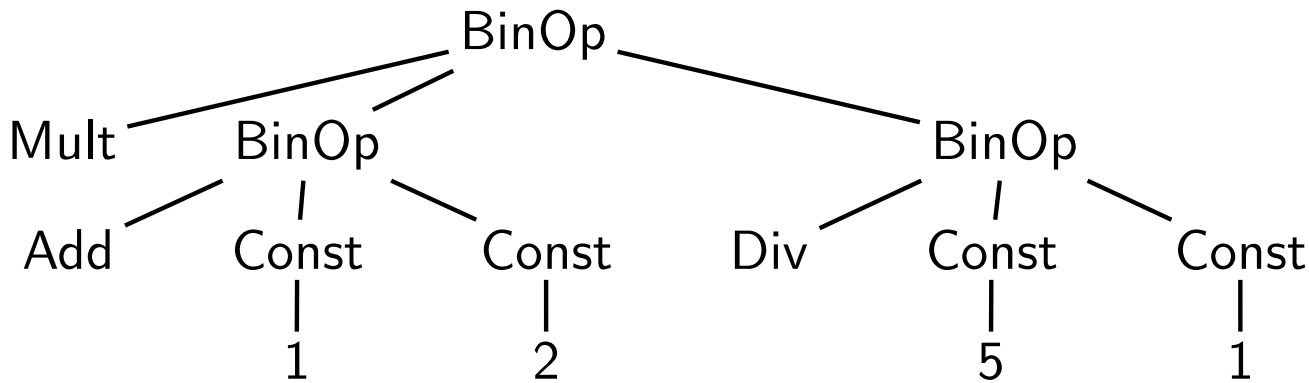
Remarque On utilisera parfois la police des arbres de syntaxe abstraite pour le code source. Par exemple (1+7)*9 (polices mêlées) au lieu de `(1+7)*9`.

Un exemple d'expression arithmétique

- Il faudrait définir l'ensemble de lexèmes dénoté par `integer` dans la grammaire.
- L'analyse lexico-syntaxique transforme l'extrait `"(1+2)*(5/1)"` en syntaxe concrète ou `"(1 +2)*(5 / 1)"` en le *terme* (c.-à-d. la valeur OCaml)

  BinOp (Mult, BinOp (Add, Const 1, Const 2), BinOp (Div, Const 5, Const 1))

  qui est le parcours préfixe gauche de l'arbre de syntaxe abstraite



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).

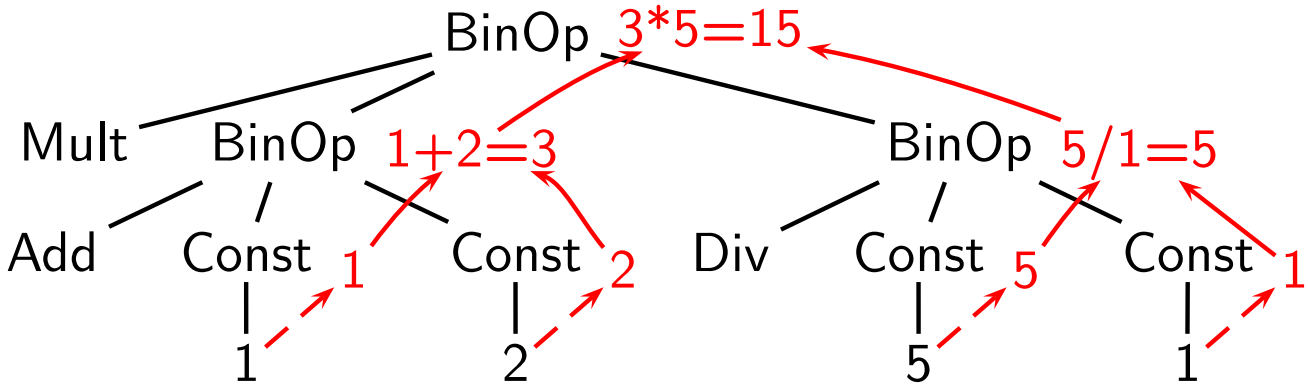
La calculette est un interprète

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 :

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

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

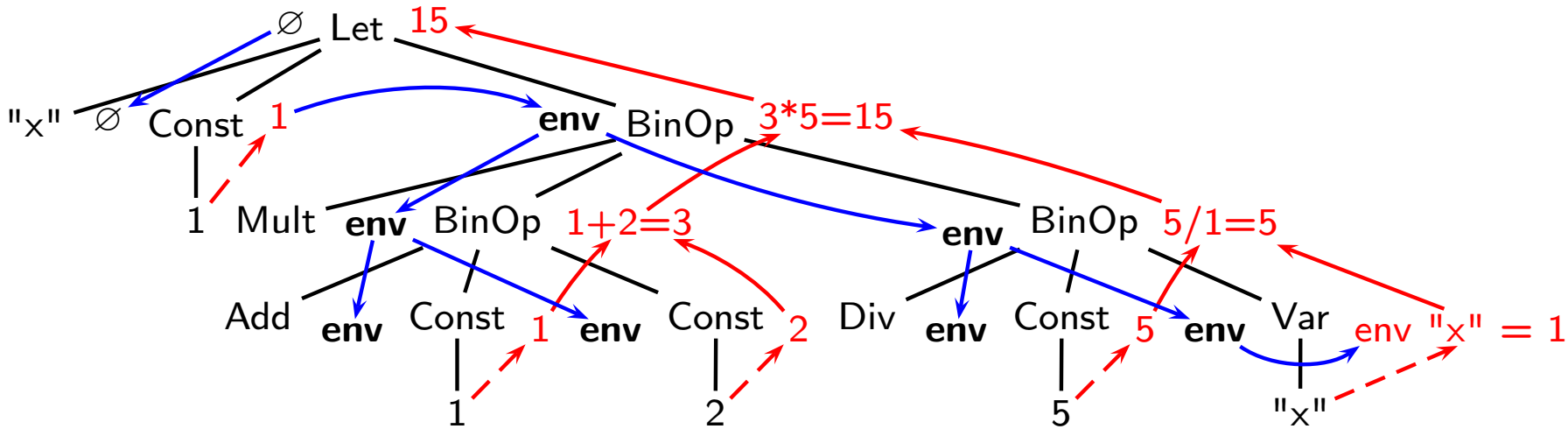
```
let rec eval env e = match e with
    Const n → n
|   BinOp (op,e1,e2) →
      let v1 = eval env e1 and v2 = eval env e2
      in begin match op with
            Add → v1 + v2 | Sub → v1 - v2 | Mult → v1 * v2 | Div → v1 / v2
         end
|   Var x → env x
|   Let (x,e1,e2) → let v1 = eval env e1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

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

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

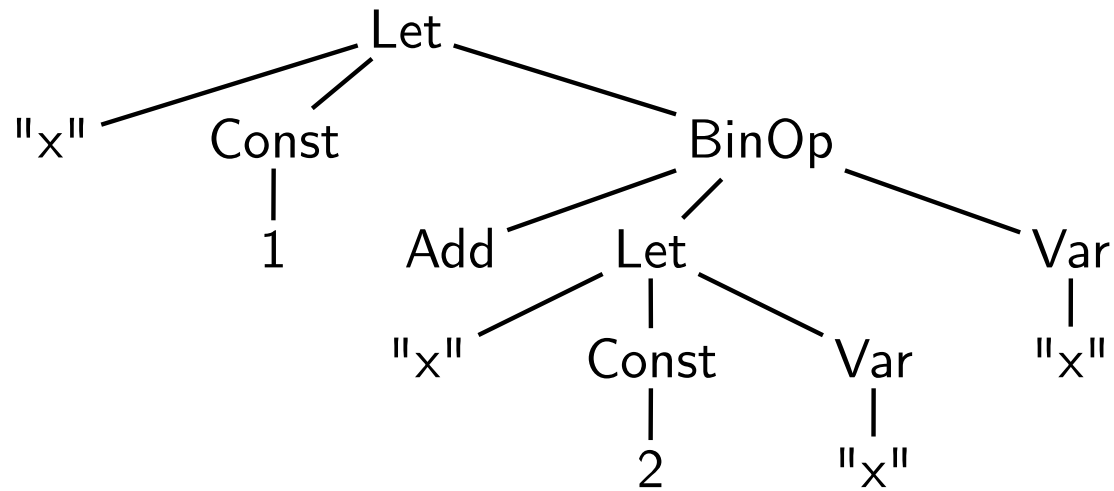
$$\frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow v_2}{\rho \vdash \langle\!\langle \overline{e}_1 * \overline{e}_2 \rangle\!\rangle \twoheadrightarrow v_1 \times v_2} \text{ mult} \qquad \frac{\rho \vdash e_1 \twoheadrightarrow v_1 \qquad \rho \vdash e_2 \twoheadrightarrow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

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

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*.

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

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) :

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

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) :

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

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)"`.

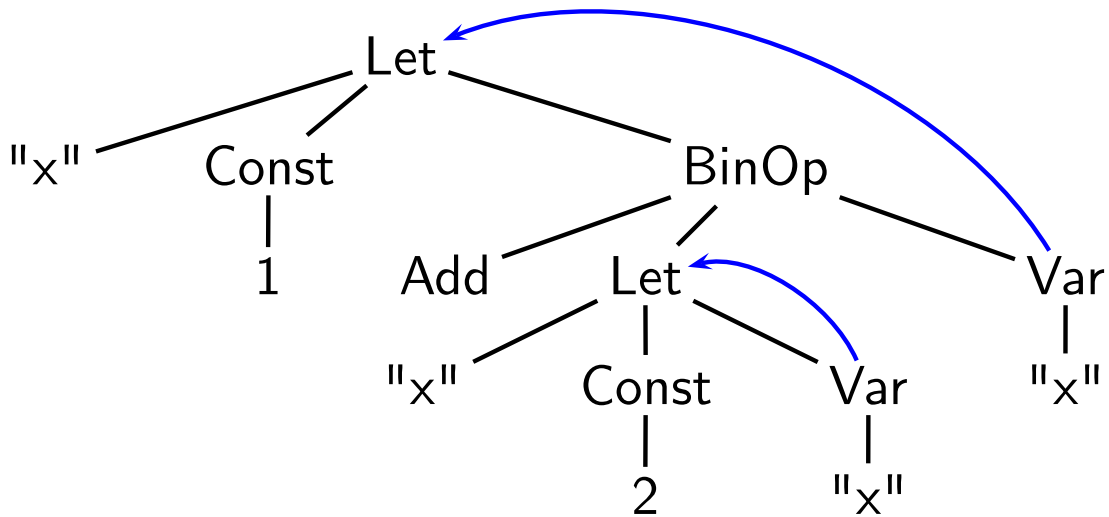
$$\mathcal{L}\langle\!\langle \textbf{let}\ \mathrm{x} = 1\ \textbf{in}\ ((\textbf{let}\ \mathrm{x} = 2\ \textbf{in}\ \mathrm{x}) + \mathrm{x})\rangle\!\rangle$$

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

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.

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

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.

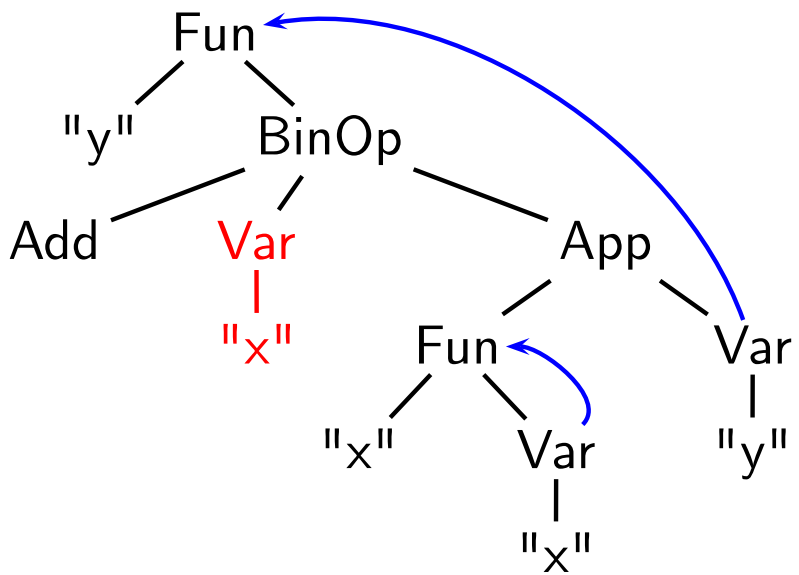
L'implantation d'un interprète ou d'un compilateur dans le même langage qu'il interprète ou compile se nomme un *bootstrap* (une auto-génération). Par exemple, le compilateur OCaml est lui-même auto-généré, un premier compilateur étant écrit en langage C.
Quelle sémantique opérationnelle pour l'abstraction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

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

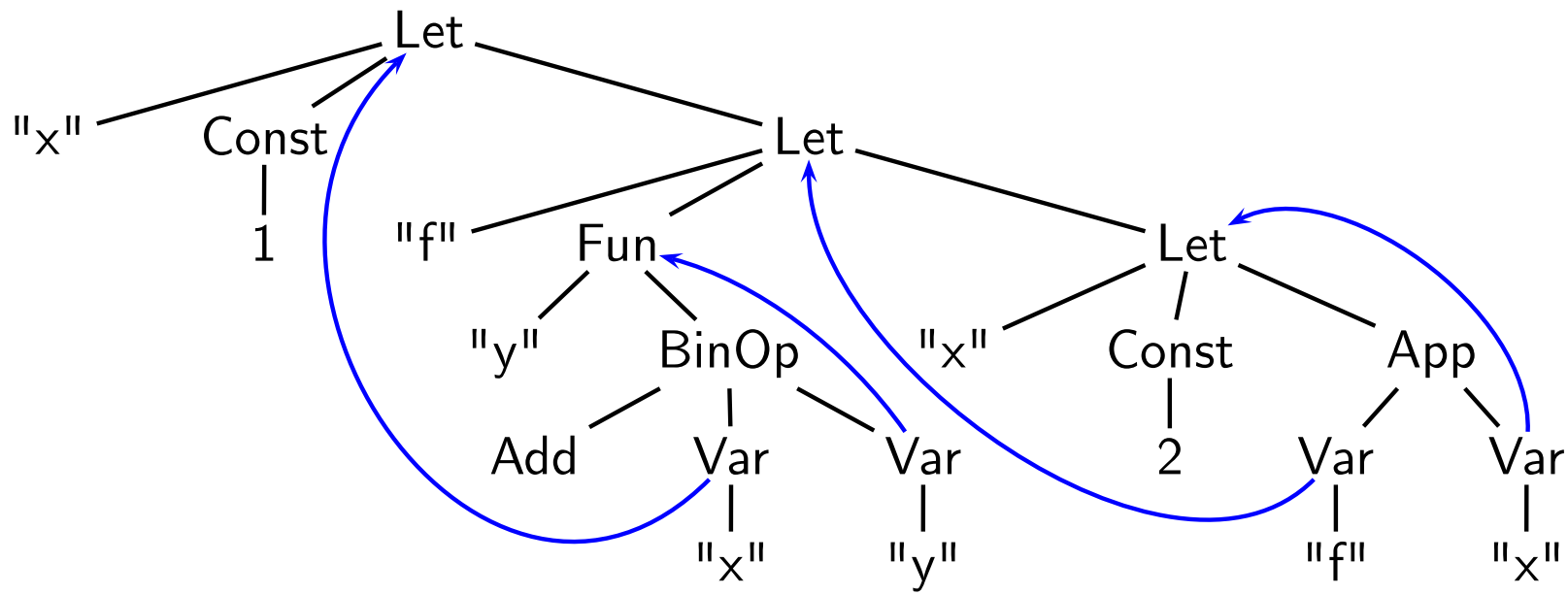
Par ailleurs remplacer une variable x par sa valeur laisse invariante la valeur de l'expression contenant x (on parle de *transparence référentielle*). C'est une propriété très désirable.

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
```

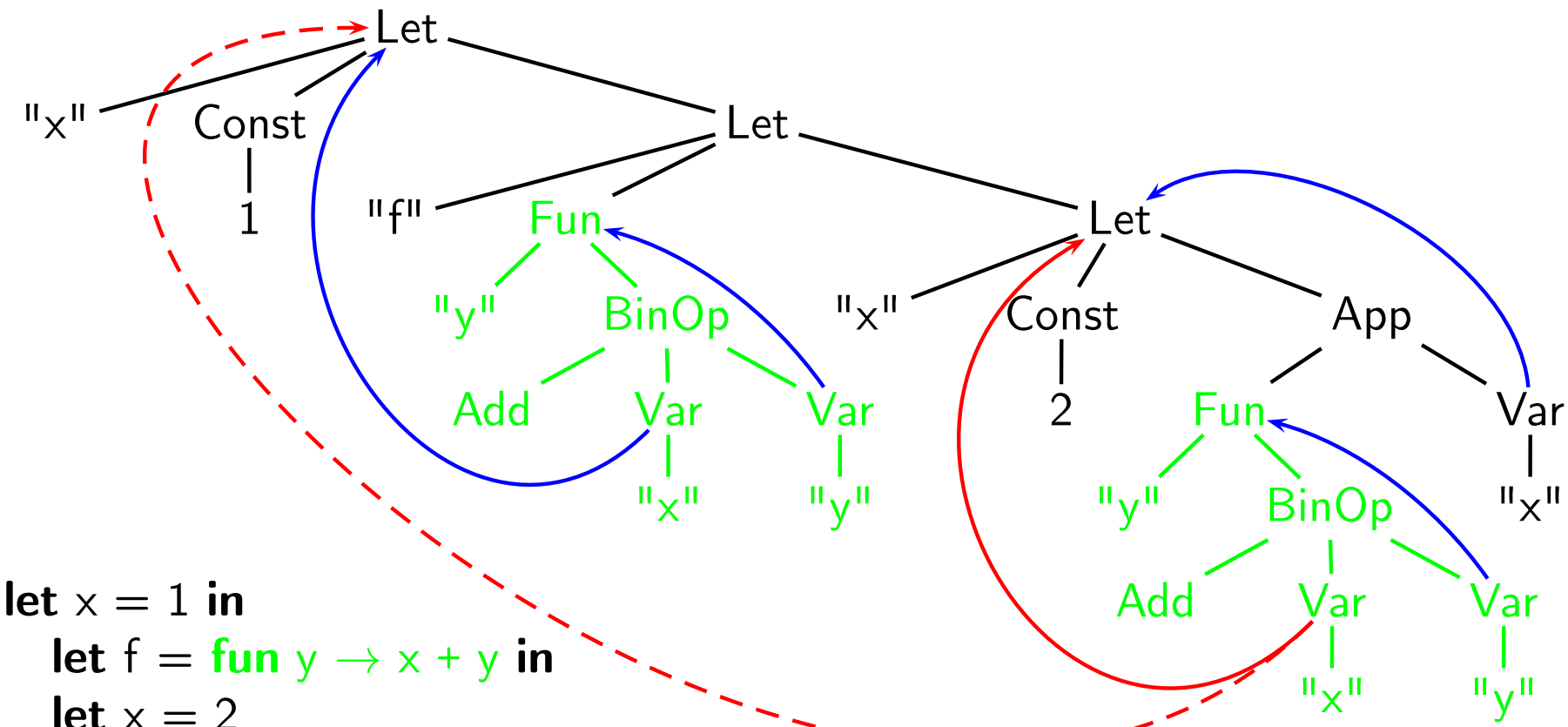
```
let x = 1 in
   let f = fun y → x + y in
   let x = 2
in (fun y → x + y) x
```

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

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

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 :

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

Une preuve d'équivalence (suite et fin)

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

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

Or la règle let (p. 36) est :

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

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)

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

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

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.

Analyses lexicales et syntaxiques

- L'analyse lexicale transforme une suite de caractères en une suite de lexèmes (mots).
- L'analyse syntaxique transforme une suite de lexèmes en une représentation arborescente (arbre de syntaxe abstraite).

Ces deux phases logiques sont implantées comme des fonctions OCaml : le pilote appelant la fonction d'analyse syntaxique qui appelle la fonction d'analyse lexicale qui lit le flux de caractères entrant et en retire les caractères reconnus comme constituant un lexème (il y a donc un effet de bord dû à l'intéraction avec le système de fichiers).

Enjeux des analyses lexicales et syntaxiques

- Les analyses lexicales et syntaxiques ont un domaine d'application bien plus large que celui de la compilation. On les retrouve comme première passe dans de nombreuses applications (analyses de commandes, de requêtes, de documents HTML etc.).
- Ces deux analyses emploient de façon essentielle les *automates*, que l'on retrouve donc dans de nombreux domaines de l'informatique et de la télématique.
- Les *expressions régulières* sont un langage de description d'automates ; elles sont utilisées dans de nombreux outils Unix (`emacs`, `grep` etc.) et sont fournies en bibliothèque avec la plupart des interprètes et compilateurs de langages de programmation (p.ex. Perl).

Objectifs

L'étude détaillée des automates et des grammaires formelles pourrait constituer un cours à part, nous nous contentons donc ici du minimum, avec comme but

- d'expliquer le fonctionnement des analyseurs de façon à pouvoir écrire soi-même des analyseurs lexicaux (*lexers* ou *scanners*) ou syntaxiques (*parsers*) ;
- de se familiariser aussi avec les expressions régulières et les automates, à cause de leur omniprésence.

Le but n'est donc ni d'écrire le cœur d'un analyseur, ni d'inventorier toutes les techniques d'analyse.

Langages formels

- Un *alphabet* est un ensemble fini non vide de *caractères*. On note souvent les alphabets Σ et les caractères a, b ou c.
- Un *mot* sur Σ est une suite, éventuellement vide, de caractères de Σ. Le mot vide est noté ε. Un mot non vide est noté par ses caractères séparés par un point (centré), par exemple $a \cdot b \cdot c$ avec $a, b, c \in \Sigma$. Le point dénote un opérateur dit de *concaténation*, que l'on peut généraliser simplement aux mots eux-mêmes : $x \cdot y$, où x et y sont des mots sur Σ.
- Le mot vide ε est un élément neutre pour la concaténation des mots : $x \cdot \varepsilon = \varepsilon \cdot x = x$ pour tout mot x.
- La concaténation est une opération associative : $x \cdot (y \cdot z) = (x \cdot y) \cdot z$
- On écrira simplement xy au lieu de $x \cdot y$

Langages formels (suite)

- Un *langage* L sur Σ est un ensemble de mots sur Σ. La concaténation peut s'étendre aux langages : $L_1 \cdot L_2 = \{xy \mid x \in L_1, y \in L_2\}$. Nous pouvons ainsi définir inductivement l'ensemble Σ^n des mots de longueur n sur l'alphabet Σ :

$$\begin{cases} \Sigma^0 = \{\varepsilon\} \\ \Sigma^{n+1} = \Sigma \cdot \Sigma^n \end{cases}$$

- L'ensemble de tous les mots sur Σ est alors $\Sigma^* = \bigcup_{n \geqslant 0} \Sigma^n$.
- L'ensemble de tous les mots *non vides* sur Σ est $\Sigma^+ = \bigcup_{n \geqslant 1} \Sigma^n$.
- On vérifie aisément que $\Sigma^* = \{\epsilon\} \cup \Sigma \cdot \Sigma^* = \{\epsilon\} \cup \Sigma^* \cdot \Sigma$
- Soient x, y et w trois mots de Σ^* tels que $w = xy$. Alors x est un *préfixe* de w et y est un *suffixe* de w. Si $x, y \in \Sigma^+$, alors x est un *préfixe propre* de w, et y est un *suffixe propre* de w.

Langages réguliers

L'ensemble $\mathcal{R}(\Sigma^*)$ des *langages réguliers* (ou *rationnels*) sur Σ^* est défini inductivement comme étant la plus petite famille de parties de Σ^* (par définition, Σ^* est le plus grand langage sur Σ) vérifiant les propriétés

- $\varnothing \in \mathcal{R}(\Sigma^*)$
- $\{\varepsilon\} \in \mathcal{R}(\Sigma^*)$
- $\forall a \in \Sigma.\{a\} \in \mathcal{R}(\Sigma^*)$
- $\forall R_1, R_2 \in \mathcal{R}(\Sigma^*).R_1 \cup R_2 \in \mathcal{R}(\Sigma^*)$
- $\forall R_1, R_2 \in \mathcal{R}(\Sigma^*).R_1 \cdot R_2 \in \mathcal{R}(\Sigma^*)$
- $\forall R \in \mathcal{R}(\Sigma^*).R^* \in \mathcal{R}(\Sigma^*)$

Expressions régulières

Une *expression régulière* est une notation compacte et simplifiée pour représenter un langage régulier. Par exemple :

Expression régulière	**Langage régulier**	**Mots du langage**
$a \mid b$ ou $a + b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	aa, aba, $abba$ etc.
$(ab)^*$	$\{ab\}^*$	ε, ab, $abab$ etc.
$abba$	$\{abba\}$	$abba$

Expressions régulières (suite)

L'ensemble $\mathcal{E}(\Sigma)$ des expressions régulières sur un alphabet Σ est le plus petit ensemble vérifiant

- $\varnothing \in \mathcal{E}(\Sigma)$
- $\{\varepsilon\} \in \mathcal{E}(\Sigma)$
- $\forall a \in \Sigma.\{a\} \in \mathcal{E}(\Sigma)$
- $\forall e_1, e_2 \in \mathcal{E}(\Sigma).e_1 + e_2 \in \mathcal{E}(\Sigma)$
- $\forall e_1, e_2 \in \mathcal{E}(\Sigma).e_1 \cdot e_2 \in \mathcal{E}(\Sigma)$
- $\forall e \in \mathcal{E}(\Sigma).e^* \in \mathcal{E}(\Sigma)$

Des expressions régulières aux langages réguliers

Le passage des expressions régulières aux langages réguliers se fait simplement par la fonction $\mathrm{L} : \mathcal{E}(\Sigma) \to \mathcal{R}$ définie inductivement par

$$\begin{aligned}
\mathrm{L}(\epsilon) &= \varnothing \\
\mathrm{L}(a) &= \{a\} \\
\mathrm{L}(e_1 + e_2) &= \mathrm{L}(e_1) \cup \mathrm{L}(e_2) \\
\mathrm{L}(e_1 \cdot e_2) &= \mathrm{L}(e_1) \cdot \mathrm{L}(e_2) \\
\mathrm{L}(e^*) &= \mathrm{L}(e)^*
\end{aligned}$$

Extensions syntaxiques

Par commodité, les outils qui emploient les expressions régulières étendent la syntaxe donnée précédemment, sans augmenter la puissance d'expression. Par exemple :

- [abc] pour (a | b | c)
- [a-f] pour (a | b | c | d | e | f)
- [^abc] pour le complémentaire de (a | b | c) dans l'ensemble des caractères
- a ? pour a | ϵ
- _ ou . pour n'importe quel caractère.

Remarque

- Le symbole + n'est pas employé dans son sens de disjonction mais de répétition non vide.

Exemples d'expressions régulières étendues

- **Entiers décimaux**

  [0-9]+

- **Entiers hexadécimaux**

  0x([0-9a-fA-F])+

- **Nombres à la Pascal**

  [0-9]+ (.[0-9]*)? ([Ee][-+][0-9]+)?

- **Sources OCaml**

```
bash$ ls *.ml{,[ily]}
```

Application à la production d'analyseurs lexicaux

On spécifie chaque sorte de lexème par une expression régulière, comme par exemple

- les mots-clés **let**, **in** etc.
- les variables [a-z]+ [a-zA-Z0-9_]*
- les entiers [0-9]+
- les symboles : ( ) + * = etc.

mais aussi le texte à oublier :

- les espaces ('␣' | '\n' | '\t')
- et les commentaires.

Aucun lexème n'est alors associé à ces expressions régulières.

Application à la production d'analyseurs lexicaux (suite)

Un logiciel prend une telle spécification et produit un programme qui implante l'analyseur lexical correspondant dans le langage source de l'application. L'analyseur est alors compilé normalement et son code objet (cible) est lié au reste de l'application. Il prend un flux de caractères et tente d'y reconnaître un lexème. S'il réussit, il renvoie celui-ci et se déplace d'autant dans le flux, sinon il signale une erreur ou un flux vide.

En langage C, les générateurs d'analyseurs lexicaux connus sont flex et lex (dans la distribution Red Hat Linux ce dernier est en fait un lien symbolique vers le premier). En Java, il y a jlex et javacc, par exemple. Pour un catalogue, cf. `http://catalog.compilertools.net/` et, autour du forum comp.compilers, cf. `http://compilers.iecc.com/`

ocamllex

Le générateur d'analyseurs lexicaux du système OCaml est ocamllex.
Les expressions régulières définissant les lexèmes ont une forme habituelle, mais les caractères sont entourés par des apostrophes (conventions de OCaml), p.ex. ['a'-'z']+ ['a'-'z' 'A'-'Z' '0'-'9' '_']* au lieu de [a-z]+ [a-zA-Z0-9_]*
Le type OCaml représentant les lexèmes n'est généralement pas défini dans la spécification (qui possède une extension `.mll`). Par exemple, ce type peut être

type *token* = INT **of** *int* | IDENT **of** *string* | TRUE | FALSE
| PLUS | MINUS | TIMES | SLASH | EQUAL | ARROW
| LPAR | RPAR LET | IN | REC
| FUN | IF | THEN | ELSE | AND | OR | NOT | EOF

L'expérience recommande d'associer la fin de fichier à un lexème (ici EOF), en particulier en conjonction avec un analyseur syntaxique.

Spécification d'analyseurs lexicaux avec ocamllex

Une spécification d'analyseur lexical pour ocamllex a la forme :

{ *Code OCaml optionnel en prologue* }
let r = *regexp*
...
rule $entrée_1$ = **parse**
  $regexp_{1,1}$ { *Code OCaml, dit action* }
| ...
| $regexp_{1,n}$ { *Code OCaml, dit action* }
and $entrée_2$ = **parse**
  ...
and ...
{ *Code OCaml optionnel en épilogue* }

Un exemple de spécification pour ocamllex

```
{ open Parser
  exception Illegal_char of string }
let ident = ['a'-'z'] ['_' 'A'-'Z' 'a'-'z' '0'-'9']*
rule token = parse
       [' ' '\n' '\t' '\r'] { token lexbuf }
     | "let"                { LET }
     | "rec"                { REC }
     | "="                  { EQUAL }
     ...
     | ident                { IDENT (Lexing.lexeme lexbuf)}
     | ['0'-'9']+           { INT (int_of_string (Lexing.lexeme lexbuf)) }
     | eof                  { EOF }
     | _                    { raise (Illegal_char (Lexing.lexeme lexbuf)) }
```

Un exemple de spécification pour ocamllex (suite)

- Le prologue ouvre le module Parser car celui-ci contient la définition du type token dont les constructeurs sont appliqués dans les actions (LET, REC etc.). C'est le style d'organisation quand on utilise conjointement un analyseur syntaxique produit par ocamlyacc (c'est la même configuration en langage C si on utilise lex avec yacc). Si on spécifie un analyseur lexical autonome (ne serait-ce que pour faire du test unitaire), on aurait alors probablement un module Token contenant la définition des lexèmes.
- On déclare les exceptions lexicales dans le prologue, ici simplement Illegal_char, qui doivent être filtrées au niveau du pilote de l'application.
- Une expression régulière nommée ident est définie, ainsi qu'une unique entrée token. Dans les actions, *les entrées sont des fonctions* dont le premier argument est toujours le flux de caractères entrant, toujours nommé lexbuf.

Un exemple de spécification pour ocamllex (suite et fin)

- Le module standard Lexing contient un certain nombre de fonctions qui servent à manipuler le flux de caractères. Par exemple Lexing.lexeme prend le flux et retourne le lexème qui a été reconnu par l'expression régulière *associée à l'action*.
- Notez l'appel récursif à token lorsque l'on veut ignorer certains caractères. Cela fonctionne car dans l'action, les caractères reconnus par l'expression régulière associée ont été ôté du flux.
- Il existe une pseudo-expression régulière eof qui sert à filtrer la fin de fichier. Il est recommandé de s'en servir pour produire un pseudo-lexème EOF, car les comportements implicites des applications vis-à-vis des fins de fichier peuvent varier d'un système d'exploitation à l'autre.
- Il existe une pseudo-expression régulière _ qui filtre n'importe quel caractère. *L'ordre des expressions est significatif*, donc cette expression devrait être la dernière.

Mise en œuvre de ocamllex

- Si la spécification ocamllex a pour nom lexer.mll, alors la compilation se fait en deux temps :

  1. `ocamllex lexer.mll`, qui produit soit une erreur soit lexer.ml, puis
  2. `ocamlc -c lexer.ml`, qui produit soit une erreur soit lexer.cmo et lexer.cmi (ce dernier, en l'absence de lexer.mli).

  En théorie, les actions associées aux expressions régulières ne sont pas tenues de renvoyer un lexème, car le programmeur est libre et peut, par exemple, écrire un préprocesseur, c.-à-d. une réécriture de fichiers, plutôt qu'un analyseur lexical.
- Pour créer un flux entrant de caractères à partir de l'entrée standard il faut **let** char_flow = Lexing.from_channel (stdin) **in** ...

Sous le capot

Le fichier OCaml résultant de la spécification a la forme

Prologue ...
let rec *entrée*$_1$ = **fun** lexbuf $\rightarrow$
    ... **match** ... **with**
            ... $\rightarrow$ *action*
        | ...
        | ... $\rightarrow$ *action*
and *entrée*$_2$ = **fun** lexbuf $\rightarrow$
    ...
and ...
Épilogue

où `lexbuf` est de type Lexing.lexbuf

Analyse des commentaires sur une ligne

Les commentaires sont reconnus durant l'analyse lexicale mais rien n'en est fait. Certains analyseurs analysent le contenu des commentaires et signalent donc des erreurs *à l'intérieur* de ceux-ci (ce qui peut être gênant si on y place des méta-données).
Le type le plus simple de commentaires est celui de C++ qui porte sur une ligne.

```
rule token = parse
    ...
  | "//"   [^ '\n']*   '\n'? { token lexbuf }
```

L'expression régulière reconnaît l'ouverture du commentaire, puis laisse passer tout caractère différent d'une fin de ligne et termine par une fin de ligne optionnelle (on suppose que le système d'exploitation est Unix et qu'une fin de ligne peut terminer le fichier).

Analyse des commentaires en blocs non imbriqués

L'entrée token reconnaît l'ouverture du commentaire, et *son action* appelle l'entrée supplémentaire in_comment qui saute tous les caractères jusqu'à la fermeture du bloc et signale une erreur si celle-ci manque (commentaire ouvert). Quand le bloc est fermé, puisqu'un commentaire ne produit pas de lexème, il faut faire un appel récursif à token pour en renvoyer un.

```
{ ... exception Open_comment }
rule token = parse
        ...
      | "/*" { in_comment lexbuf }
and in_comment = parse
        "*/" { token lexbuf }
      | eof  { raise Open_comment }
      | _    { in_comment lexbuf }
```

Analyse des commentaires en blocs imbriqués

Les commentaires du langage C peuvent être imbriqués pour isoler temporairement une partie du code qui est déjà commentée.

S'ils n'étaient pas imbriqués, on aurait pu écrire une seule expression régulière (nous ne l'avons pas fait pour des raisons de lisibilité et pour signaler facilement une absence de fermeture). Dans le cas imbriqué il n'existe pas une telle expression *pour des raisons théoriques*. On dit que les langages réguliers ne peuvent être bien parenthèsés.

L'idée est que les expressions régulières ne peuvent « garder la mémoire » du degré d'imbrication courant. Pour y parvenir on se sert donc de l'expressivité du code *des actions*. **Ainsi c'est abusivement que l'on réduit l'analyse lexicale aux seuls langages réguliers, spécifiés par des expressions régulières.**

Analyse des commentaires en blocs imbriqués (suite)

La technique consiste à modifier l'entrée in_comment de sorte que les actions soient *des fonctions dont l'argument est la profondeur d'imbrication courante.*

```
rule token = parse
        ...
      | "/*" { in_comment lexbuf 1 }
and in_comment = parse
        "*/" { fun depth → if depth = 1 then token lexbuf
                           else in_comment lexbuf (depth-1) }
      | "/*" { fun depth → in_comment lexbuf (depth+1) }
      | eof  { raise Open_comment }
      | _    { in_comment lexbuf }
```

Notez que in_comment lexbuf *est équivalent à* **fun** depth → in_comment lexbuf depth *et que* **fun** depth → raise Open_comment *serait moins efficace.*

Automates finis et expressions régulières

Les générateurs d'analyseurs lexicaux doivent combiner les expressions régulières de la spécification et les traduire vers du code source. Pour cela, elles sont d'abord traduites dans un formalisme de même expressivité, mais plus intuitif : les *automates finis*. Finalement, l'automate résultant des traitements du générateur est compilé en code source.

Commençons par présenter un cas particulier d'automate fini, dit *déterministe* (AFD). Un AFD $\mathcal{A}$ est un quintuplet $(\Sigma, \mathcal{Q}, \delta, q_0, F)$ où

- Σ est un alphabet ;
- $\mathcal{Q}$ est un ensemble fini d'états ;
- $\delta : \mathcal{Q} \times \Sigma \to \mathcal{Q}$ est la fonction (partielle) de transition ;
- q_0 est l'état initial ;
- F est un ensemble d'états finaux.

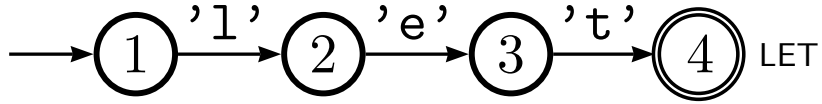
Automates finis déterministes

- Dans ce contexte, on parle aussi d'*étiquette* pour les éléments de Σ.
- Une transition est un triplet sur $\mathcal{Q} \times \Sigma \times \mathcal{Q}$
- On peut étendre δ sur $\mathcal{Q} \times \Sigma^* \to \mathcal{Q}$ par $\begin{cases} \delta(q, \epsilon) = q \\ \delta(q, aw) = \delta(\delta(q, a), w) \end{cases}$
- Le langage $\mathrm{L}(\mathcal{A})$ *reconnu* par l'automate $\mathcal{A}$ est l'ensemble $\{w \mid \delta(q_0, w) \in F\}$ des mots permettant d'atteindre un état final à partir de l'état initial.
- On pourrait considérer qu'il y a plusieurs états initiaux possibles au lieu d'un seul, mais cela n'apporterait rien quant à l'analyse lexicale (qui est une application particulière de la théorie des automates finis).
- Un automate est *complet* si pour tout état q et toute étiquette a, $\delta(q, a)$ est défini.

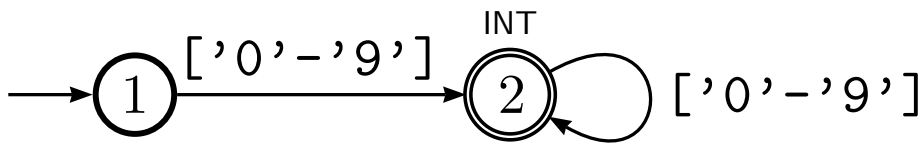
Exemples d'automates

Les automates permettent de reconnaître les lexèmes.

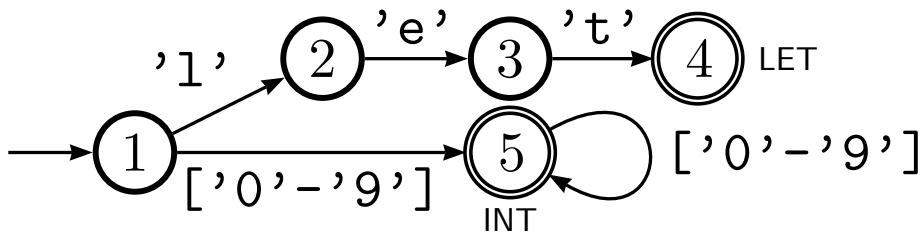
- les mots-clés :



- les entiers :



- l'un ou l'autre :



Si un état final (double cerclage) est atteint à partir de l'état initial (flèche entrante), un lexème est identifié (ici LET ou INT).

Analyse lexicale avec des automates

L'analyseur lexical considère deux informations :

- l'état courant dans l'automate spécifié,
- le caractère en tête du flux entrant.

Puis

- s'il existe une transition pour le caractère dans l'automate, alors
  - il est retiré du flux (et jeté) ;
  - l'état courant devient celui indiqué par la transition ;
  - on recommence à considérer les nouveaux état et caractère.
- s'il n'y a pas de transition (état bloquant), alors
  - si l'état courant est final alors le lexème associé est émis.
  - sinon il y a erreur (caractère illégal).

Ambiguïtés lexicales

Les problèmes qui peuvent se poser sont :

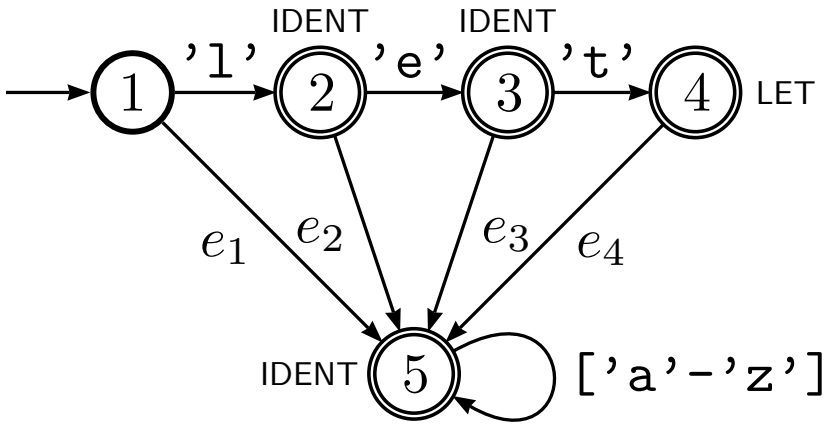
- la chaîne "let" pourrait être reconnue comme une variable et non tel mot-clé ;
- la chaîne "letrec" pourrait être reconnue comme la liste de lexèmes [LET ; IDENT "rec"] ou [LET ; REC] ou [IDENT "letrec"] etc.

La solution générale consiste à établir des règles de priorité :

- lorsque plusieurs lexèmes sont des préfixes possibles, retenir le plus long ;
- sinon suivre l'ordre de définition des sortes de lexèmes (p.ex. dans la spécification page 114 l'expression régulière "let" est écrite *avant* ident).

Ainsi la phrase `let letrec = 3 in 1 + funny` est reconnue comme la liste [LET ; IDENT "letrec" ; EQUAL ; INT 3 ; IN ; INT 1 ; PLUS ; IDENT "funny"].

Réalisation de la règle du lexème le plus long



e_1 = ['a'-'k' 'n'-'z']

e_2 = ['a'-'d' 'f'-'z']

e_3 = ['a'-'s' 'u'-'z']

e_4 = ['a'-'z]

Pour réaliser la règle du lexème le plus long, il faut ajouter une structure : une file de caractères (initialement vide) et reprendre l'algorithme page 126. Lorsque l'état courant est final et qu'une transition est possible, au lieu de jeter le caractère correspondant, il faut le conserver dans la file jusqu'à un état bloquant. Si cet état est final on renvoie le lexème associé, **sinon on retourne le lexème du dernier état final rencontré, les caractères de la file sont remis dans le flux entrant et on revient à l'état initial**.

Automates finis non-déterministes asynchrones

Pour construire de façon intuitive des automates à partir d'expressions régulières et leur appliquer des transformations, nous avons besoin d'une classe d'automates un peu différente : les automates finis *non-déterministes asynchrones* (AFNA). Un AFNA diffère sur deux points des AFD :

- On étend les étiquettes par le mot vide ε (on parle de transitions spontanées), c.-à-d. qu'on remplace Σ par $\Sigma \cup \{\varepsilon\}$: c'est l'asynchronisme.
- Il peut y avoir plusieurs transitions de même étiquette à partir d'un même état. Techniquement, δ est alors une *relation* (ternaire) sur $\mathcal{Q} \times (\Sigma \cup \{\varepsilon\}) \times \mathcal{Q}$ et non plus une fonction partielle : c'est le non-déterminisme.

Typage statique

Le but du *typage statique* est de détecter et de rejeter dès la compilation (jusqu'ici les erreurs étaient détectées à l'exécution) un certain nombre de programmes absurdes, comme 1 2 ou
$\mathrm{BinOp}\,(\mathrm{Add}, (\mathbf{fun}\,\mathrm{x} \to \mathrm{x}), 1)$.Pour cela un *type* est attribué à chaque sous-expression du programme (p.ex. *int* pour une expression arithmétique, ou $\mathit{int} \to \mathit{int}$ pour une fonction des entiers vers les entiers) et la *cohérence* de ces types est vérifiée.

Déterminer toutes les erreurs d'exécution pour tous les programmes est un problème indécidable. Or les systèmes de types sont souvent décidables car on souhaite que le compilateur termine pour tous les programmes. *Il est donc impossible de ne rejeter que les programmes erronés*. Tout système de types rejette des programmes innocents, c'est pourquoi la quête de meilleurs systèmes est sans fin.

Définition formelle des types de la calculette

- Syntaxe concrète

```
Type ::= "int" | Type -> Type | "(" Type ")"
       | "'a" | "'b" | ...
```

  On dénote les variables de type avec les méta-variables $\overline{\alpha}$, $\overline{\beta}$ etc.

- Syntaxe abstraite

  type *type_expr* = TEint | TEfun **of** *type_expr* * *type_expr* | TEvar **of** *string*;;

  On dénote les types avec la méta-variable τ.

Définition formelle des types de la calculette (suite)

- Analyse syntaxique
  La flèche est associative à droite.
  $\langle\langle int \rangle\rangle$ = TEint et $\langle\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\rangle$ = TEfun $(\langle\langle \overline{\tau}_1 \rangle\rangle, \langle\langle \overline{\tau}_2 \rangle\rangle)$ et
  $\langle\langle$ 'a $\rangle\rangle$ = TEvar "a" etc.
  On note τ au lieu de $\langle\langle \overline{\tau} \rangle\rangle$. On note les variables de type α, β, γ etc. au lieu de TEvar "a", TEvar "b", TEvar "c" etc.
- Variables libres $\mathcal{L}\langle\langle int \rangle\rangle = \varnothing$ et $\mathcal{L}\langle\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\rangle = \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2)$ et $\mathcal{L}\langle\langle \overline{\alpha} \rangle\rangle = \{\alpha\}$.

Système de types — typage monomorphe

Un jugement de typage est de la forme $\Gamma \vdash e : \tau$ et se lit « Dans l'environnement de typage Γ, l'expression e a pour type τ. » Un environnement de typage Γ lie des variables x à leur type $\Gamma(x)$. Un liaison de typage se note $x : \tau$. Soit

$$\Gamma \vdash \langle\!\langle \overline{n} \rangle\!\rangle : \langle\!\langle int \rangle\!\rangle \quad \text{Tconst} \qquad \frac{\Gamma \vdash e_1 : \langle\!\langle int \rangle\!\rangle \qquad \Gamma \vdash e_2 : \langle\!\langle int \rangle\!\rangle}{\Gamma \vdash \langle\!\langle \overline{e}_1 \ \overline{o} \ \overline{e}_2 \rangle\!\rangle : \langle\!\langle int \rangle\!\rangle} \ \text{Tbin}$$

$$\Gamma \vdash \langle\!\langle x \rangle\!\rangle : \Gamma(x) \quad \text{Tvar} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \oplus x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \langle\!\langle \textbf{let } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle : \tau_2} \ \text{Tlet}$$

Système de types — typage monomorphe (suite)

$$\frac{\Gamma \oplus x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \langle\!\langle \textbf{fun}\ x \to \overline{e} \rangle\!\rangle : \langle\!\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\!\rangle}\ \mathsf{Tfun}$$

$$\frac{\Gamma \vdash e_1 : \langle\!\langle \overline{\tau}' \to \overline{\tau} \rangle\!\rangle \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \langle\!\langle \overline{e}_1\ \ \overline{e}_2 \rangle\!\rangle : \tau}\ \mathsf{Tapp}$$

Exemple de preuve de typage (ou dérivation de typage)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{x} : \langle\langle \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{x} \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle} \qquad \dfrac{}{\mathsf{x} : \langle\langle \mathit{int} \rangle\rangle \vdash \langle\langle 1 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}}{\mathsf{x} : \langle\langle \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{x} + 1 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}}{\varnothing \vdash \langle\langle \textbf{fun}\ \mathsf{x} \to \mathsf{x} + 1 \rangle\rangle : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle} \qquad \dfrac{\dfrac{}{\mathsf{f} : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{f} \rangle\rangle : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle} \qquad \dfrac{}{\mathsf{f} : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle \vdash \langle\langle 2 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}}{\mathsf{f} : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle \vdash \langle\langle \mathsf{f}\ 2 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}}{\varnothing \vdash \langle\langle \textbf{let}\ \mathsf{f} = \textbf{fun}\ \mathsf{x} \to \mathsf{x} + 1\ \textbf{in}\ \mathsf{f}\ 2 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle}$$

Voici d'autres jugements de typage dérivables :
$\varnothing \vdash \langle\langle \textbf{fun}\ \mathsf{x} \to \mathsf{x} \rangle\rangle : \langle\langle \overline{\alpha} \to \overline{\alpha} \rangle\rangle$ et $\varnothing \vdash \langle\langle \textbf{fun}\ \mathsf{x} \to \mathsf{x} \rangle\rangle : \langle\langle \mathit{int} \to \mathit{int} \rangle\rangle$
Voici des jugements non dérivables :
$\varnothing \vdash \langle\langle \textbf{fun}\ \mathsf{x} \to \mathsf{x} + 1 \rangle\rangle : \langle\langle \mathit{int} \rangle\rangle$ et $\varnothing \vdash \langle\langle \textbf{fun}\ \mathsf{x} \to \mathsf{x} + 1 \rangle\rangle : \langle\langle \overline{\alpha} \to \mathit{int} \rangle\rangle$

Quid de l'auto-application ?

Pour typer **fun** f $\to$ f f il faudrait construire une dérivation de la forme suivante :

$$\dfrac{\dfrac{\Gamma \oplus \mathsf{f} : \tau_1 \vdash \mathsf{f} : \langle\!\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\!\rangle \qquad \Gamma \oplus \mathsf{f} : \tau_1 \vdash \mathsf{f} : \tau_2}{\Gamma \oplus \mathsf{f} : \tau_1 \vdash \langle\!\langle \mathsf{f}\ \mathsf{f} \rangle\!\rangle : \tau_2}}{\Gamma \vdash \langle\!\langle \textbf{fun}\ \mathsf{f} \to \mathsf{f}\ \ \mathsf{f} \rangle\!\rangle : \langle\!\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\!\rangle}$$

Pour que les feuilles de la dérivation soient justifiée par l'axiome Tvar, il faudrait que $\tau_1 = \langle\!\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\!\rangle$ et $\tau_1 = \tau_2$. La première de ces égalités est impossible, car τ_1 serait un sous-terme strict de lui-même, ce qui est impossible pour tout terme τ_1 fini.

Quelques propriétés du typage

- Expressions bien typées ou typables
  Une expression e est typable s'il existe un environnement de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$
- Typage Un typage est une paire (Γ, τ) ou une dérivation de typage.
- Stabilité par substitution de variables de types
  Si on peut dériver un jugement non clos, c.-à-d. contenant des variables de types libres, p.ex. $\mathsf{f} : \langle\!\langle \overline{\alpha} \to \overline{\alpha} \rangle\!\rangle \oplus \mathsf{x} : \alpha \vdash \mathsf{f}(\mathsf{x}) : \alpha$, alors on peut aussi dériver tous les jugements obtenus en remplaçant ces variables par des types arbitraires, p.ex.
  $\mathsf{f} : \langle\!\langle \mathit{int} \to \mathit{int} \rangle\!\rangle \oplus \mathsf{x} : \mathit{int} \vdash \mathsf{f}(\mathsf{x}) : \mathit{int}$
- Sûreté du typage
  Si $\Gamma \vdash e : \tau$ et $\rho \vdash e \twoheadrightarrow r$ alors $r \neq \mathsf{Err}(\ldots)$
  Cette propriété est la motivation même du typage statique.

Normalisation forte

Le système de typage monomorphe présenté ici est *fortement normalisant*, c.-à-d. que les programmes bien typés terminent toujours.

Ces systèmes de types ne sont donc pas intéressants en programmation car on souhaite que le langage soit Turing-complet (p. 67), c.-à-d. qu'il accepte *tous* les programmes qui terminent. Si le système de types rejetait de plus tous les programmes qui ne terminent pas, on aurait résolu le problème de l'arrêt de la machine de Turing, qui est connu pour être indécidable (p. 69).

Tout langage Turing-complet muni d'un système de types décidable contient donc des programmes qui ne terminent pas.

Normalisation faible et opérateur de point fixe

C'est pourquoi on considère habituellement des systèmes de types qui ne garantissent pas que les programmes typables terminent (on parle de normalisation faible). Un bon exemple est le système de types monomorphe de mini-ML muni d'un opérateur de point fixe fix ou d'un **let rec** natif :

$$\frac{\Gamma \oplus x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma \oplus x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \langle\!\langle \textbf{let rec } x = \overline{e}_1 \textbf{ in } \overline{e}_2 \rangle\!\rangle : \tau_2} \text{ Tlet-rec}$$

Il n'y a ici aucune difficulté par rapport à Tlet, il faut juste ajouter la liaison $x : \tau_1$ dans l'environnement de typage de la première prémisse.

De la vérification pure à l'inférence complète

Pour un langage statiquement typé, un *typeur* est un algorithme qui, étant donné un programme, détermine si celui-ci est typable et, si oui, en produit un type. Si le programme admet plusieurs types, un typeur devra produire un *type principal*, c.-à-d. un type « plus général » que tous les autres types possibles (cette notion dépend du système de type considéré).

Exemple En mini-ML, le programme **fun** x $\rightarrow$ x a les types $\tau \rightarrow \tau$ pour n'importe quel type τ, mais le type $\alpha \rightarrow \alpha$ est principal, puisque tous les autres types s'en déduisent par substitution (de la variable α).

Selon la nature et la quantité d'informations, sous forme d'*annotations* de types, que le langage exige, la tâche du typeur est plus ou moins complexe. Il existe de nombreuses configurations.

Vérification pure

Dans le cas de la pure vérification de types, toutes les sous-expressions du programme, ainsi que tous les identificateurs, doivent être annotés par leur type.

Exemple

fun (x : *int*)→ (
    let y : *int* = (+ : *int* ∗ *int* → *int*) (x : *int*) (1 : *int*)
    in y : *int*
) : *int*

Le typeur est alors très simple, puisque le programmeur n'écrit en fait plus uniquement une expression, mais une dérivation de typage complète.

Bien sûr, un tel langage est inutilisable en pratique ; aucun langage réaliste n'adopte cette approche extrême.

Déclaration des types des variables et propagation des types

Le programmeur doit déclarer les types des paramètres de fonction et des variables locales. Le typeur infère alors le type de chaque expression à partir des types de ses sous-expressions. Autrement dit, l'information de typage est propagée à travers l'expression des feuilles vers la racine.

Exemple Sachant que x est de type *int*, le typeur peut non seulement vérifier que l'expression x + 1 est bien typée, mais aussi inférer qu'elle a le type *int*. Ainsi l'exemple précédent devient :

$$\textbf{fun}\ (\mathrm{x} : \mathit{int}) \rightarrow \textbf{let}\ \mathrm{y} : \mathit{int} = \mathrm{x} + 1\ \textbf{in}\ \mathrm{y}$$

Le typeur infère le type $\mathit{int} \rightarrow \mathit{int}$ pour cette expression.

Une approche similaire à celle-ci est adoptée par la plupart des langages impératifs, tels Pascal, C, Java etc.

Déclaration des types des paramètres et propagation des types

Le programmeur doit déclarer uniquement les types des paramètres. La différence par rapport au cas précédent est donc que les variables locales (liées par **let**) ne sont plus obligatoirement annotées lorsqu'elles sont introduites.

Le typeur en détermine alors le type en se fondant sur le type de l'expression à laquelle elles sont associées.

Exemple Notre exemple devient

$$\textbf{fun}\ (\mathrm{x} : \mathit{int}) \rightarrow \textbf{let}\ \mathrm{y} = \mathrm{x} + 1\ \textbf{in}\ \mathrm{y}$$

Ayant déterminé que x + 1 est de type *int*, le typeur associe le type *int* à y.

Inférence complète des types

Plus aucune annotation de types n'est exigée. Le typeur détermine le type des paramètres d'après l'utilisation qui en est faite par la fonction.

Exemple Notre exemple devient

$$\textbf{fun}\ \mathrm{x} \rightarrow \textbf{let}\ \mathrm{y} = \mathrm{x} + 1\ \textbf{in}\ \mathrm{y}$$

Puisque l'addition + n'opère que sur des entiers, x est nécessairement de type *int*. Cet intéressant processus de déduction, ou *inférence*, est celui utilisé par les langages de la famille ML (dont OCaml).

Inférence de types pour mini-ML avec typage monomorphe

Pour réaliser l'inférence de types pour mini-ML avec typage monomorphe, on procède en trois temps :

1. on annote l'arbre de syntaxe abstraite par des variables de types ;
2. à partir de cet arbre décoré on construit un système d'équations entre types, qui caractérise tous les typages possibles pour le programme ;
3. on résout ce système d'équations sachant que s'il n'a pas de solution le programme est alors mal typé, sinon on détermine une solution *principale* qui nous permet de déduire un typage principal du programme.

En combinant ces phases nous obtenons un algorithme qui détermine si un programme est typable, et, si oui, en fournit un typage principal.

Substitution

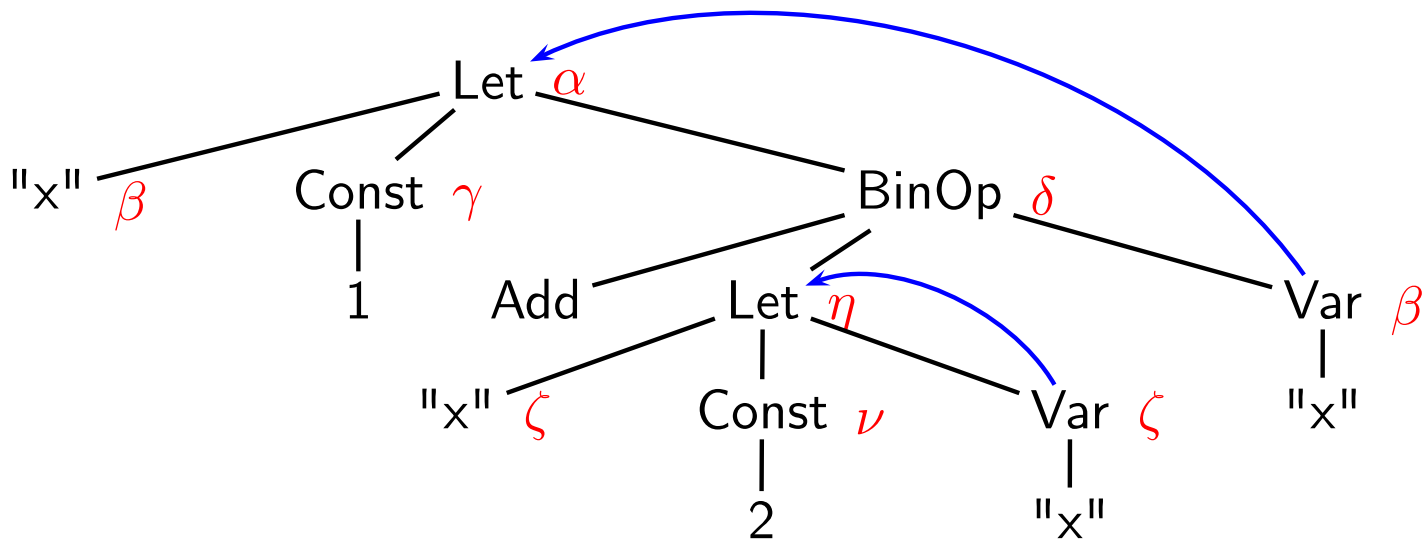
Nous avons besoin d'un nouveau concept pour décrire l'inférence de types : la *substitution*. C'est une fonction dont l'application a la forme générale $\tau[\alpha \leftarrow \tau']$, qu'on lit « La substitution de la variable de type α par le type τ' dans le type τ. » Elle se définit par induction sur la structure des types :

$$
\begin{aligned}
\langle\langle \mathit{int} \rangle\rangle[\alpha \leftarrow \tau'] &= \langle\langle \mathit{int} \rangle\rangle \\
\alpha[\alpha \leftarrow \tau'] &= \tau' \\
\beta[\alpha \leftarrow \tau'] &= \beta && \text{si } \beta \neq \alpha \\
\langle\langle \overline{\tau}_1 \to \overline{\tau}_2 \rangle\rangle[\alpha \leftarrow \tau'] &= \langle\langle \overline{\tau}'_1 \to \overline{\tau}'_2 \rangle\rangle && \text{où } \tau'_1 = \tau_1[\alpha \leftarrow \tau'] \text{ et } \tau'_2 = \tau_2[\alpha \leftarrow \tau']
\end{aligned}
$$

La notion de substitution peut s'étendre à d'autres objets, comme les expressions. On notera φ, ψ ou θ une substitution.

Décoration de l'arbre de syntaxe abstraite par des variables de types

Reprenons l'arbre de syntaxe abstraite page 51 et décorons-le avec des variables de type uniques respectant les liaisons, c'est-à-dire qu'une variable liée est annotée par la même variable de type que celle de son lieur et les autres expressions sont annotées par des variables uniques :



Notons en exposant d'une expression la variable de type qui l'annote, par exemple $\text{Let}^{\alpha}(x^{\beta}, {e_1}^{\gamma}, {e_2}^{\delta})$.

Construction du système d'équations (collection de contraintes)

À partir d'une expression annotée e^{α} nous construisons un système d'équations $C(e^{\alpha})$ capturant les contraintes de typage entre les sous-expressions de e. Ce système est défini inductivement sur la structure de e :

$$
\begin{aligned}
C(\mathsf{Var}^{\alpha}\, x) &= \varnothing \\
C(\mathsf{Const}^{\alpha}\, n) &= \{\alpha = \langle\!\langle \mathit{int} \rangle\!\rangle\} \\
C(\mathsf{BinOp}^{\alpha}(\_, {e_1}^{\beta}, {e_2}^{\gamma})) &= \{\alpha = \langle\!\langle \mathit{int} \rangle\!\rangle; \beta = \alpha; \gamma = \alpha\} \\
&\quad \cup\, C({e_1}^{\beta}) \cup C({e_2}^{\gamma}) \\
C(\mathsf{Let}^{\alpha}(x^{\beta}, {e_1}^{\gamma}, {e_2}^{\delta}) &= \{\beta = \gamma; \alpha = \delta\} \cup C({e_1}^{\gamma}) \cup C({e_2}^{\delta}) \\
C(\mathsf{Fun}^{\alpha}(x^{\beta}, e^{\gamma})) &= \{\alpha = \langle\!\langle \overline{\beta} \to \overline{\gamma} \rangle\!\rangle\} \cup C(e^{\gamma}) \\
C(\mathsf{App}^{\alpha}({e_1}^{\beta}, {e_2}^{\gamma})) &= \{\beta = \langle\!\langle \overline{\gamma} \to \overline{\alpha} \rangle\!\rangle\} \cup C({e_1}^{\beta}) \cup C({e_2}^{\gamma})
\end{aligned}
$$

Construction du système d'équations (exemple)

Poursuivons avec l'exemple précédent. Nous avons alors :

$$
\begin{aligned}
C(e^{\alpha}) = \quad \{ \quad & \beta = \gamma; \alpha = \delta; \\
& \gamma = \langle\langle int \rangle\rangle; \\
& \delta = \langle\langle int \rangle\rangle; \eta = \delta; \beta = \delta; \\
& \zeta = \nu; \eta = \zeta; \\
& \nu = \langle\langle int \rangle\rangle \}
\end{aligned}
$$

Lien entre jugements de typage et solutions des équations

Une *solution* de l'ensemble d'équations $C(e^\alpha)$ est une substitution φ telle que pour toute équation $\tau_1 = \tau_2 \in C(e^\alpha)$ on a $\varphi(\tau_1) = \varphi(\tau_2)$. Autrement dit, une solution est un *unificateur* du système d'équations. Les propositions suivantes établissent que les solutions de $C(e^\alpha)$ caractérisent exactement les typages de e.

Proposition (Correction des équations)

Si φ est une solution de $C(e^\alpha)$ alors $\Gamma \vdash e^\alpha : \varphi(\alpha)$, où Γ est l'environnement de typage $\{x^\beta : \varphi(\beta) \mid x^\beta \in \mathcal{L}(e)\}$.

Proposition (Complétude des équations)

Soit e une expression. S'il existe un environnement de typage Γ et un type τ tels que $\Gamma \vdash e : \tau$, alors le système d'équations $C(e^\alpha)$ admet une solution φ telle que $\varphi(\alpha) = \tau$ et $\Gamma = \{x^\beta : \varphi(\beta) \mid x^\beta \in \mathcal{L}(e)\}$.

Résolution des équations (l'unificateur de Robinson)

- Définissons $\varphi \leqslant \psi$ s'il existe une substitution θ telle que $\psi = \theta \circ \varphi$.
- Par définition, une solution φ de $C(e^\alpha)$ est dite *principale* si toute solution ψ de $C(e^\alpha)$ vérifie $\varphi \leqslant \psi$.
- Il existe un algorithme mgu qui, étant donné un système d'équations C, soit échoue soit produit une solution principale de C :

$$\begin{aligned}
\mathsf{mgu}(\varnothing) &=_1 \forall x.x \mapsto x \\
\mathsf{mgu}(\{\tau = \tau\} \cup C') &=_2 \mathsf{mgu}(C') \\
\mathsf{mgu}(\{\alpha = \tau\} \cup C') &=_3 \mathsf{mgu}(C'[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \notin \mathcal{L}(\tau) \\
\mathsf{mgu}(\{\tau = \alpha\} \cup C') &=_4 \mathsf{mgu}(\{\alpha = \tau\} \cup C') \\
\mathsf{mgu}(C \cup C') &=_5 \mathsf{mgu}(\{\tau_1 = \tau_1'; \tau_2 = \tau_2'\} \cup C')
\end{aligned}$$

where $C = \{\langle\langle \overline{\tau_1} \to \overline{\tau_2} \rangle\rangle = \langle\langle \overline{\tau_1}' \to \overline{\tau_2}' \rangle\rangle\}$

Dans tous les autres cas, mgu échoue (pas de solutions).

Résolution des équations (exemple)

$$\mathsf{mgu}(C(e^{\alpha})) =_3 \varphi_0 \circ [\beta \leftarrow \gamma]$$

$$\begin{aligned}
\varphi_0 &= \mathsf{mgu}(\{\alpha = \delta; \gamma = \langle\!\langle \mathit{int} \rangle\!\rangle; \delta = \langle\!\langle \mathit{int} \rangle\!\rangle; \eta = \delta; \\
&\qquad\qquad \gamma = \delta; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
&=_3 \varphi_1 \circ [\alpha \leftarrow \delta] \\
\varphi_1 &= \mathsf{mgu}(\{\gamma = \langle\!\langle \mathit{int} \rangle\!\rangle; \delta = \langle\!\langle \mathit{int} \rangle\!\rangle; \eta = \delta; \gamma = \delta; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
&=_3 \varphi_2 \circ [\gamma \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \\
\varphi_3 &= \mathsf{mgu}(\{\delta = \langle\!\langle \mathit{int} \rangle\!\rangle; \eta = \delta; \langle\!\langle \mathit{int} \rangle\!\rangle = \delta; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
&=_3 \varphi_4 \circ [\delta \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \\
\varphi_4 &= \mathsf{mgu}(\{\eta = \langle\!\langle \mathit{int} \rangle\!\rangle; \langle\!\langle \mathit{int} \rangle\!\rangle = \langle\!\langle \mathit{int} \rangle\!\rangle; \zeta = \nu; \eta = \zeta; \nu = \langle\!\langle \mathit{int} \rangle\!\rangle\}) \\
&=_3 \varphi_5 \circ [\eta \leftarrow \langle\!\langle \mathit{int} \rangle\!\rangle] \\
\varphi_5 &= \mathsf{mgu}(\{\langle\!\langle \mathit{int} \rangle\!\rangle = \langle\!\langle \mathit{int} \rangle\!\rangle; \zeta = \nu; \langle\!\langle \mathit{int} \rangle\!\rangle = \zeta; \nu = \langle\!\langle \mathit{int} \rangle\!\rangle\})
\end{aligned}$$

Résolution des équations (fin de l'exemple)

$$
\begin{array}{rcl}
\varphi_5 & =_2 & \mathsf{mgu}(\{\zeta = \nu; \langle\!\langle int \rangle\!\rangle = \zeta; \nu = \langle\!\langle int \rangle\!\rangle\}) \\
 & =_3 & \varphi_6 \circ [\zeta \leftarrow \nu] \\
\varphi_6 & = & \mathsf{mgu}(\{\langle\!\langle int \rangle\!\rangle = \nu; \nu = \langle\!\langle int \rangle\!\rangle\}) \\
 & =_4 & \varphi_5 \circ [\nu \leftarrow \langle\!\langle int \rangle\!\rangle] \\
\varphi_5 & = & \mathsf{mgu}(\{\langle\!\langle int \rangle\!\rangle = \langle\!\langle int \rangle\!\rangle; \langle\!\langle int \rangle\!\rangle = \langle\!\langle int \rangle\!\rangle\}) \\
 & = & \mathsf{mgu}(\{\langle\!\langle int \rangle\!\rangle = \langle\!\langle int \rangle\!\rangle\}) \\
 & =_2 & \mathsf{mgu}(\varnothing) \ =_1 \ \forall x.x \mapsto x \\
\mathsf{mgu}(C(e^\alpha)) & = & [\nu \leftarrow \langle\!\langle int \rangle\!\rangle] \circ [\zeta \leftarrow \nu] \circ [\eta \leftarrow \langle\!\langle int \rangle\!\rangle] \circ [\delta \leftarrow \langle\!\langle int \rangle\!\rangle] \\
 & & \circ [\gamma \leftarrow \langle\!\langle int \rangle\!\rangle] \circ [\alpha \leftarrow \delta] \circ [\beta \leftarrow \gamma] \\
\mathsf{mgu}(C(e^\alpha))(\alpha) & = & \alpha([\delta \leftarrow \langle\!\langle int \rangle\!\rangle] \circ [\alpha \leftarrow \delta]) = \delta[\delta \leftarrow \langle\!\langle int \rangle\!\rangle] = \langle\!\langle int \rangle\!\rangle
\end{array}
$$

L'algorithme d'inférence et ses propriétés

Proposition (Correction de mgu)

Si mgu$(C) = \varphi$ *alors* φ *est une solution de* C*.*

Proposition (Complétude de mgu)

Si C *admet une solution* ψ *alors mgu*(C) *réussit et produit une solution* φ *telle que* $\varphi \leqslant \psi$*.*