Langages, interprétation, compilation

Thibaut Balabonski @ Université Paris-Saclay
http://www.lri.fr/~blsk/CompilationLDD3/
V2, automne 2022
Quatrième partie

5 Sémantique et types

Formalisation de ce qui signifient un programme ou une donnée, de la manière dont il faut les interpréter, et de ce qu'on peut en attendre.

5.1 Valeurs et opérations typées

À l'intérieur de l'ordinateur, une donnée est une séquence de bits. Voici par exemple un mot mémoire de 32 bits.

```
1110 0000 0110 1100 0110 0111 0100 1000
```

Pour faciliter la lecture, on représente souvent un tel mot au format hexadécimal. En l'occurrence, on l'écrirait

```
0x e0 6c 67 48
```

(le `0x` au début indique simplement le format hexadécimal, puis chaque caractère correspond à un groupe de 4 bits).

Que peut signifier cette donnée ? Pour le savoir, il faut une connaissance *très* précise du contexte :

- si la donnée est une adresse mémoire, il s'agira de l'adresse `3 765 200 712`,
- si la donnée est un nombre entier signé 32 bits en complément à 2, ce nombre sera −529 766 584,
- si la donnée est un nombre flottant simple précision de la norme IEEE754, ce nombre sera $15\,492\,936 \times 2^{42}$,
- si la donnée est une chaîne de caractères au format Latin-1, il s'agira de `"Holà"`.

Si on oublie le contexte dans lequel une séquence de bits a du sens, on est susceptible de faire n'importe quoi. *Par exemple : appliquer une opération d'addition entière aux représentations des chaînes de caractères* `"5"` *et* `"37"` *produit la nouvelle chaîne de caractères* `"h7"`.

Opération incohérentes En réalité, toutes les opérations proposées par un langage de programmation sont contraintes.

- L'addition `5 + 37` entre deux entiers est possible en caml
- Les opérations `"5" + 37` ou `5 + (fun x -> 37)` ou `5(37)` ne le sont pas.

On a déjà pu observer ce point dans l'interprète du langage FUN vu au chapitre 2. L'ensemble des valeurs que pouvait produire une expression y était séparé en deux catégories : les nombres et les fonctions.

```
type value =
  | VCst  of int
  | VClos of string * expr * value Env.t
```

et on pouvait voir pour certaines opérations des comportements distincts en fonction de la catégorie de valeur manipulée. Une opération arithmétique binaire par exemple était censée s'appliquer à des nombres. Elle produisait un résultat (de sorte `VCst`) si ses deux opérandes avaient bien des valeurs de la sorte `VCst`, et échouait sinon en interrompant l'exécution du programme avec `assert false`.

```
let rec eval_binop op e1 e2 env =
  match eval e1 env, eval e2 env with
  | VCst n1, VCst n2 -> VCst (op n1 n2)
  | _ -> assert false
```

Les types : classification des valeurs En général, un langage de programmation distingue de nombreuses classes différentes de valeurs, appelées ***types***. La classification dépend de chaque langage, mais on y retrouve souvent de nombreux éléments commun. On a d'une part des types de base du langage, pour les données simples. Par exemple :

— nombres : `int`, `double`,
— valeurs booléennes : `bool`,
— caractères : `char`,
— chaînes : `string`.

D'autre part, des types plus riches peuvent être construits à partir de ces types de base. Par exemple :

— tableaux : `int[]`,
— fonctions : `int -> bool`,
— structures de données : `struct point { int x; int y; };`,
— objets : `class Point { public final int x, y; ... }`.

Une fois cette classification établie, chaque opération va s'appliquer à des éléments d'un type donné.

Dans certains cas, un même opérateur peut s'appliquer à plusieurs types d'éléments, avec des significations différentes à chaque fois. On parle de ***surcharge***. En python et en java par exemple, l'opérateur + peut s'appliquer :

— à deux entiers, et désigne alors l'addition : `5 + 37 = 42`,
— à deux chaînes, et désigne alors la concaténation : `"5" + "37" = "537"`.

Les langages de programmation permettent également parfois le ***transtypage*** (*cast*), c'est-à-dire la conversion d'une valeur d'un type vers un autre. Cette conversion peut même être implicite. Ainsi l'opération `"5" + 37` mélangeant une chaîne et un entier aura comme résultat :

— `42` en php, où la chaîne `"5"` est convertie en le nombre `5`,
— `"537"` en java, où l'entier `37` est converti en la chaîne `"37"`.

Notez qu'une telle conversion peut demander une traduction de la donnée ! Le nombre `5` est représenté par le mot mémoire `0x 00 00 00 05`, mais la chaîne `"5"` par le mot mémoire `0x 00 00 00 35`. De même, le nombre `37` est représenté par le mot mémoire `0x 00 00 00 25`, mais la chaîne `"37"` par le mot mémoire `0x 00 00 37 33`. Dans un sens comme dans l'autre, une conversion d'un type à l'autre demande de calculer la nouvelle représentation.

Bilan. Le type d'une valeur donne une clé d'interprétation de cette donnée, et peut être utile à la sélection des bonnes opérations. En outre, une incohérence dans les types révèle un problème du programme, dont l'exécution doit donc être évitée.

Analyse statique des types Gérer les types des données au moment de l'exécution génère des coûts variés :

— de la mémoire pour accompagner chaque donnée d'une indication de son type,
— des tests pour sélectionner les bonnes opérations,
— des exécutions interrompues en cas de problème, ...

Dans des langages ***typés dynamiquement*** comme python, ces coûts sont la norme. En revanche, les langages ***typés statiquement*** comme C, java ou caml nous épargnent tout ou partie de ces coûts à l'exécution en gérant autant que possible tout ce qui concerne les types dès la compilation.

Dans l'analyse ***statique*** des types, c'est-à-dire l'analyse des types *à la compilation*, on associe à chaque expression d'un programme un type, qui prédit le type de la valeur qui sera produite par cette expression. Cette prédiction est basée sur des contraintes associées à chaque élément de la syntaxe abstraite. Par exemple, en présence d'une expression d'addition de la forme `Add(e1, e2)` nous pouvons noter deux faits :

— l'expression produira un nombre,
— les deux sous-expressions `e1` et `e2` doivent impérativement produire des valeurs numériques, sans quoi l'ensemble serait mal formé.

On associe de même un type à une variable, pour désigner le type de la valeur référencée par cette variable. Ainsi, dans `let x = e in x + 1` le type de `x` est le type de la valeur produite par l'expression `e` (et on s'attend à ce qu'il s'agisse d'un nombre entier). Enfin, le type d'une fonction va mentionner ce que sont les types attendus pour chacun des paramètres, ainsi que le type du résultat renvoyé.

Le slogan associé à cette vérification de la cohérence des types avant l'exécution des programmes, du à Robin Milner, est

Well-typed programs do not go wrong.

L'objectif du typage statique est ainsi de rejeter les programmes absurdes avant même qu'ils soient exécutés (ou livrés à un client...). Dans l'absolu, on ne peut pas identifier avec certitude tous les programmes problématiques (les questions de ce genre sont généralement algorithmiquement indécidables). On cherche donc à établir des critères décidables qui :

— apportent de la ***sûreté***, c'est-à-dire qui rejettent les programmes absurdes,
— tout en laissant de l'***expressivité***, c'est-à-dire qui ne rejettent pas trop de programmes non-absurdes.

Pour permettre cette analyse des types, on peut être amené à placer un certain nombre d'indications dans le texte d'un programme. Voici quelques possibilités que l'on pourrait imaginer.

1. Annoter toutes les sous-expressions.

```
fun (x : int) ->
  let (y : int) = ((x : int) + (1 : int) : int)
  in (y : int)
```

   Ici, le programmeur fait tout le travail, et le compilateur doit simplement ***vérifier*** la cohérence de l'ensemble. Heureusement, aucun langage n'impose cela.

2. Annoter seulement les variables et les paramètres des fonctions.

```
fun (x : int) -> let (y : int) = x+1 in y
```

   Dans ce cas, le compilateur déduit le type de chaque expression en se référant aux types fournis pour les variables. C'est ce qu'il faut faire en C ou en java.

3. Annoter seulement les paramètres des fonctions.

```
fun (x : int) -> let y = x+1 in y
```

4. Ne rien annoter.

```
fun x -> let y = x+1 in y
```

   Dans ce dernier cas, c'est au compilateur d'***inférer*** le type que doit avoir chaque variable et chaque expression, sans aide du programmeur. C'est ce qui se passe en caml.

Lorsque l'analyse des types est faite à la compilation, la sélection de la bonne instruction en cas d'opérateur surchargé est elle-même faite pendant la compilation, et ne coûte donc plus rien à l'exécution. En outre, la vérification statique de la cohérence des types permet la détection précoce des incohérences du programme : de nombreux problèmes sont corrigés plus tôt.

Dans la suite de ce chapitre, nous allons formaliser la notion de type et les contraintes associées, voir comment réaliser un vérificateur de types ou un programme d'inférence de types, et transformer notre vague notion de sûreté des programmes bien typés en un théorème.

5.2 Jugement de typage et règles d'inférence

Pour caractériser les programmes bien typés, on définit des règles permettant de justifier que « dans un contexte Γ, une expression e est cohérente et admet le type τ ». Cette phrase entre guillemets est appelée un ***jugement de typage*** et est notée

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

Le contexte Γ mentionné dans le jugement de typage est l'association d'un type à chaque variable de l'expression e.

Le jugement de typage n'est pas une fonction associant un type à chaque expression, mais simplement une relation entre ces trois éléments : contexte, expression, type. En particulier certaines expressions *e n'ont pas* de type (parce qu'elles sont incohérentes), et dans certaines situations on peut avoir plusieurs types possibles pour une même expression.

Règles de typage Pour illustrer la manière dont on peut formaliser la cohérence et le type d'une expression, concentrons-nous sur un fragment du langage FUN comportant des nombres, des variables et des fonctions :

$$
\begin{array}{rcl}
e & ::= & n \\
& | & e \texttt{ + } e \\
& | & x \\
& | & \texttt{let } x \texttt{ = } e \texttt{ in } e \\
& | & \texttt{fun } x \texttt{ -> } e \\
& | & e\, e
\end{array}
$$

Nous aurons donc besoin de manipuler un type de base pour les nombres, et des types de fonctions.

$$
\begin{array}{rcl}
\tau & ::= & \texttt{int} \\
& | & \tau \to \tau
\end{array}
$$

Un type de la forme $\tau_1 \to \tau_2$ est le type d'une fonction dont le paramètre attendu a le type τ_1 et le résultat renvoyé a le type τ_2.

À chaque construction du langage, on va associer une règle énonçant
- le type que peut avoir une expression de cette forme, et
- les éventuelles contraintes qui doivent être vérifiées pour que l'expression soit cohérente.

Commençons avec la partie arithmétique. On donnera chaque règle sous deux formes : une description en langue naturelle, et sa traduction comme une ***règle d'inférence*** (voir cours de logique !).

- Une constante entière n admet le type `int`.

$$\frac{}{\Gamma \vdash n \texttt{ : int}}$$

- Si les expressions e_1 et e_2 sont cohérentes et admettent le type `int`, alors l'expression e `+` e est cohérente et admet également le type `int`.

$$\frac{\Gamma \vdash e_1 \texttt{ : int} \qquad \Gamma \vdash e_2 \texttt{ : int}}{\Gamma \vdash e_1 \texttt{ + } e_2 \texttt{ : int}}$$

Dans la règle pour l'addition, les deux jugements $\Gamma \vdash e_1$ `:` `int` et $\Gamma \vdash e_2$ `:` `int` sont les prémisses, et le jugement $\Gamma \vdash e_1$ `+` e_2 `:` `int` est la conclusion. Autrement dit, si l'on a pu d'une manière ou l'autre justifier $\Gamma \vdash e_1$ `:` `int` et $\Gamma \vdash e_2$ `:` `int`, alors la règle permet d'en déduire $\Gamma \vdash e_1$ `+` e_2 `:` `int`. À l'inverse, la règle pour la constante entière n'a pas de prémisse (on l'appelle un ***axiome***, ou ***cas de base***). Cela signifie que nous n'avons besoin de rien d'autre que cette règle pour justifier un jugement $\Gamma \vdash n$ `:` `int`.

Les règles concernant les variables vont faire intervenir le contexte Γ, aussi appelé ***environnement***, puisque c'est lui qui consigne les types associés à chaque variable.

- Une variable a le type donné par l'environnement.

$$\frac{}{\Gamma \vdash x \texttt{ : } \Gamma(x)}$$

  Notez ici que l'on considère Γ comme une fonction : $\Gamma(x)$ désigne le type associé par Γ à la variable x. En outre, l'application de cette règle suppose que $\Gamma(x)$ est bien définie, c'est-à-dire que x appartient au domaine de Γ.
- Une variable locale est associée au type de l'expression qui la définit.

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

  Dans une telle expression, e_2 peut contenir la variable locale x. Le typage de e_2 a donc lieu dans un environnement étendu noté $\Gamma, x : \tau_1$, qui reprend toutes les associations de Γ et y ajoute l'association du type τ_1 à la variable x. Cette variable x en revanche n'existe pas dans e_1, et n'est pas non plus une variable libre de l'expression complète : elle n'apparaît donc pas dans l'environnement de typage de e_1 ni de `let` x `=` e_1 `in` e_2.

Une fonction a un type de la forme $\alpha \to \beta$, où α désigne le type attendu du paramètre, et β le type du résultat.

— Une fonction doit être appliquée à un paramètre effectif du bon type.

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1\, e_2 \ : \ \tau_1}$$

— Dans le corps d'une fonction, le paramètre formel est vu comme une variable dont le type correspond au type attendu pour le paramètre.

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e \ : \ \tau_1 \to \tau_2}$$

Les règles des ***types simples***, sont donc intégralement contenues dans les six règles d'inférence suivantes.

$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \ \texttt{+}\ e_2 \ : \ \texttt{int}}$$

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

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e \ : \ \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1\, e_2 \ : \ \tau_1}$$

Expressions typables Pour justifier un jugement de typage pour une expression concrète dans un contexte donné, on enchaîne les déductions à l'aide des règles d'inférence. Ainsi, dans le contexte $\Gamma = \{x \ : \ \texttt{int},\ f \ : \ \texttt{int} \to \texttt{int}\}$ on peut tenir le raisonnement suivant.

1. $\Gamma \vdash x \ : \ \texttt{int}$ est valide, par la règle des variables.
2. $\Gamma \vdash f \ : \ \texttt{int} \to \texttt{int}$ est valide, par la règle des variables.
3. $\Gamma \vdash 1 \ : \ \texttt{int}$ est valide, par la règle des constantes.
4. $\Gamma \vdash f\ 1 \ : \ \texttt{int}$ est valide, par la règle d'application et avec les deux points 2. et 3. déjà justifiés.
5. $\Gamma \vdash x \ \texttt{+}\ f\ 1 \ : \ \texttt{int}$ est valide, par la règle d'addition et avec les deux points 1. et 4. déjà justifiés.

Ce raisonnement, appelé une ***dérivation***, peut également être présenté sous la forme d'un ***arbre de dérivation*** ayant à la racine la conclusion que l'on cherche à justifier.

$$\dfrac{\dfrac{}{\Gamma \vdash x \ : \ \texttt{int}} \qquad \dfrac{\dfrac{}{\Gamma \vdash f \ : \ \texttt{int} \to \texttt{int}} \qquad \dfrac{}{\Gamma \vdash 1 \ : \ \texttt{int}}}{\Gamma \vdash f\ 1 \ : \ \texttt{int}}}{\Gamma \vdash x \ \texttt{+}\ f\ 1 \ : \ \texttt{int}}$$

Dans un tel arbre, chaque barre correspond à une application de règle, et chaque sous-arbre à la justification d'un jugement intermédiaire (une prémisse).

Dans certaines situations, il est possible de dériver plusieurs jugements associant plusieurs types distincts à la même expression. On peut par exemple aussi bien justifier les deux jugements suivants :

$$\vdash \texttt{fun x -> x} \ : \ \texttt{int} \to \texttt{int}$$

$$\vdash \texttt{fun x -> x} \ : \ (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$$

Ici, l'absence de contexte Γ signifie que le typage est effectué dans le contexte vide.

Expressions non typables Si une expression e est incohérente, les règles de typage *ne permettront pas* de justifier de jugements de la forme $\Gamma \vdash e \ : \ \tau$, quels que soient le contexte Γ ou le type τ. On peut le voir en montrant que la construction d'un hypothétique arbre de dérivation justifiant un tel jugement arrive nécessairement à une impasse, c'est-à-dire une situation dans laquelle il est clair que plus aucune règle ne permet de conclure.

Prenons l'exemple de l'expression `5(37)`, que l'on peut encore écrire `5 37`. Il s'agit d'une application. La seule règle permettant de justifier un jugement $\Gamma \vdash \texttt{5 37} : \tau$ est la règle relative aux applications, qui demande de justifier au préalable les deux prémisses $\Gamma \vdash \texttt{5} : \tau' \to \tau$ et $\Gamma \vdash \texttt{37} : \tau'$ (pour un type τ' que l'on peut librement choisir, mais qui doit bien être le même dans les deux jugements). Or il est impossible de justifier une prémisse de la forme $\Gamma \vdash \texttt{5} : \tau' \to \tau$: aucune règle ne permet d'associer à une constante entière un type de fonction (en effet, la seule règle applicable à une constante entière donnerait $\Gamma \vdash \texttt{5} : \texttt{int}$).

Considérons le deuxième exemple **`fun`** `x -> x x`. De même, une seule règle étant applicable à chaque forme d'expression, un arbre de dérivation d'un jugement $\Gamma \vdash \texttt{fun x -> x x} : \tau$ aurait nécessairement la forme

$$\dfrac{\dfrac{\Gamma, \texttt{x} : \tau_1 \vdash \texttt{x} : \tau_1 \to \tau_2 \qquad \Gamma, \texttt{x} : \tau_1 \vdash \texttt{x} : \tau_1}{\Gamma, \texttt{x} : \tau_1 \vdash \texttt{x x} : \tau_2}}{\Gamma \vdash \texttt{fun x -> x x} : \tau_2}$$

Or la prémisse $\Gamma, \texttt{x} : \tau_1 \vdash \texttt{x} : \tau_1 \to \tau_2$ est injustifiable : les règles d'inférence ne permettent pas dans ce contexte d'associer à `x` un autre type que τ_1, et il n'existe pas de types τ_1 et τ_2 vérifiant l'équation $\tau_1 = \tau_1 \to \tau_2$.

Raisonner sur les expressions bien typées Démontrons que pour tout contexte Γ, toute expression e et tout type τ, si $\Gamma \vdash e : \tau$ est valide alors l'ensemble des variables libres de e est inclus dans le domaine de Γ.

Les jugements de typage valides étant définis par un système d'inférence, nous pouvons établir des propriétés vraies pour toutes les expressions bien typées en raisonnant par récurrence sur la structure de la dérivation de typage. Nous avons donc un cas par règle d'inférence, et chaque prémisse de la règle correspondante nous donne une hypothèse de récurrence.

Montrons donc que si $\Gamma \vdash e : \tau$ alors $\mathrm{fv}(e) \subseteq \mathrm{dom}(\Gamma)$, par récurrence sur $\Gamma \vdash e : \tau$.

- Cas $\Gamma \vdash n : \texttt{int}$. On a $\mathrm{fv}(n) = \emptyset$, avec bien sûr $\emptyset \subseteq \mathrm{dom}(\Gamma)$.
- Cas $\Gamma \vdash x : \Gamma(x)$. On a $\mathrm{fv}(x) = \{x\}$, et l'application de la règle suppose justement que $\Gamma(x)$ est bien définie, c'est-à-dire $x \in \mathrm{dom}(\Gamma)$.
- Cas $\Gamma \vdash e_1 + e_2 : \texttt{int}$, avec les deux prémisses $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$. Les deux prémisses nous donnent les deux hypothèses de récurrence $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ et $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$. Or $\mathrm{fv}(e_1 + e_2) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$. Des deux hypothèses de récurrence on déduit $\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$, et donc $\mathrm{fv}(e_1 + e_2) \subseteq \mathrm{dom}(\Gamma)$.
- Cas $\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2$ avec les deux prémisses $\Gamma \vdash e_1 : \tau_1$ et $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. Les deux prémisses nous donnent les deux hypothèses de récurrence $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ et $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$ (remarquez en effet que la prémisse relative à e_2 est dans un environnement étendu avec la variable x). Or $\mathrm{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus \{x\})$. Par la première hypothèse de récurrence nous avons $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$. Par la deuxième hypothèse de récurrence nous avons $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$, dont nous déduisons $\mathrm{fv}(e_2) \setminus \{x\} \subseteq \mathrm{dom}(\Gamma)$. Ainsi on a bien $\mathrm{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) \subseteq \mathrm{dom}(\Gamma)$.
- Les deux cas relatifs aux fonctions sont similaires à ceux déjà traités.

5.3 Vérification de types pour FUN

Si suffisamment d'annotations sont fournies dans le programme source, on peut facilement déduire des règles de typage un ***vérificateur*** de types, c'est-à-dire un (autre) programme qui dit si le programme analysé est cohérent ou non. On va écrire un programme caml pour la vérification des types dans le fragment du langage FUN pour lequel nous venons d'établir des règles de typage. Ce programme prendra la forme d'une fonction `type_expr` qui prend en paramètres une expression e et un environnement Γ et qui :

- renvoie l'unique type qui peut être associé à e dans l'environnement Γ si e est effectivement cohérente dans cet environnement,
- échoue sinon.

On définit un type de données (caml) pour manipuler en caml les types du langage FUN.

```
type typ =
  | TypInt
  | TypFun of typ * typ
```

On ajuste le type caml représentant les arbres de syntaxe abstraite du langage FUN pour y inclure des annotations de type. En l'occurrence on n'introduit cette annotation que pour l'argument d'une fonction : il s'agit du deuxième argument du constructeur `Fun`.

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Var of string
  | Let of string * expr * expr
  | Fun of string * typ * expr
  | App of expr * expr
```

Enfin, on va représenter les environnements comme des tables associatives associant des identifiants de variables (`string`) à des types du langage FUN (`typ`).

```
module Env = Map.Make(String)
type type_env = typ Env.t
```

Le vérificateur est alors une fonction récursive

```
type_expr: expr -> type_env -> typ
```

qui observe la forme de l'expression et traduit la règle d'inférence correspondante.

```
let rec type_expr e env = match e with
```

Une constante entière est toujours cohérente, et de type `int`.

$$\frac{}{\Gamma \vdash n \ : \ \texttt{int}}$$

```
  | Cst _ -> TypInt
```

Une variable est considérée comme cohérente si elle existe effectivement dans l'environnement.

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

```
  | Var(x) -> Env.find x env
```

Dans le cas contraire, c'est la fonction `Env.find` qui lèvera une exception (en l'occurrence : `Not_found`).

Une addition demande que chaque opérande soit cohérent, et du bon type.

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 \ \texttt{+} \ e_2 \ : \ \texttt{int}}$$

```
  | Add(e1, e2) ->
    let t1 = type_expr e1 env in
    let t2 = type_expr e2 env in
    if t1 = TypInt && t2 = TypInt then
      TypInt
    else
      failwith "type error"
```

Notez que ce code peut échouer à plusieurs endroits différents : pendant la vérification de `e1` ou `e2` si l'une ou l'autre de ces expressions n'est pas cohérente, ou explicitement avec la dernière ligne si `e1` et `e2` sont toutes deux cohérentes mais que l'une n'a pas le type attendu.

Lorsque l'on introduit une variable locale x, on déduit son type de l'expression e_1 définissant cette variable. Le type obtenu peut alors être ajouté à l'environnement pour la vérification du type de la deuxième expression e_2.

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

```
| Let(x, e1, e2) ->
    let t1 = type_expr e1 env in
    type_expr e2 (Env.add x t1 env)
```

Ce cas n'échoue jamais directement lui-même (les vérifications de `e1` et `e2` peuvent en revanche bien échouer).

Dans le cas d'une fonction, on utilise l'annotation pour fixer le type de l'argument. On peut ensuite vérifier le type du corps de la fonction, en déduire le type du résultat renvoyé, et reconstruire le type complet de la fonction.

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e \ : \ \tau_1 \to \tau_2}$$

```
| Fun(x, tx, e) ->
    let te = type_expr e (Env.add x tx env) in
    TypFun(tx, te)
```

Dans le cas d'une application, il faut vérifier que le terme de gauche a bien le type d'une fonction, puis que le terme de droite a bien le type attendu par la fonction. Ces deux points donnent deux raisons distinctes d'échouer dans la vérification.

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1\, e_2 \ : \ \tau_1}$$

```
| App(f, a) ->
    let tf = type_expr f env in
    let ta = type_expr a env in
    begin match tf with
      | TypFun(tx, te) ->
         if tx = ta then
           te
         else
           failwith "type error"
      | _ -> failwith "type error"
    end
```

5.4 Polymorphisme

Avec les types simples que nous avons vu jusqu'ici, une expression comme

```
fun x -> x
```

peut admettre plusieurs types distincts. En revanche, on ne peut lui donner qu'un seul type à la fois. En particulier, dans une expression comme

```
let f = fun x -> x in f f
```

on ne peut donner à `f` qu'un seul des types possibles et l'expression complète ne peut pas être typée. On parle de type ***monomorphe*** (littéralement : *une seule forme*). *Vous pouvez remarquer que caml en revanche n'a pas de difficultés à typer cette expression.*

On s'intéresse dans cette section au ***polymorphisme paramétrique***, c'est-à-dire à la possibilité d'exprimer des types paramétrés, recouvrant plusieurs variantes d'une même forme. On étend la grammaire des types τ en y ajoutant deux éléments :

- des ***variables***, ou ***paramètres***, de type, notées α, β, ... et désignant des types indéterminés,
- une quantification universelle $\forall\alpha.\tau$ décrivant un type ***polymorphe***, où la variable de type α peut, dans τ, désigner n'importe quel type.

Pour notre langage FUN, l'ensemble des types serait donc défini par la grammaire étendue

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \tau \to \tau \\ & | & \alpha \\ & | & \forall \alpha.\tau \end{array}$$

Instanciation L'utilisation d'une expression polymorphe suit le principe suivant : si une expression e admet un type polymorphe $\forall \alpha.\tau$, alors pour tout type τ' on peut encore considérer que e admet le type $\tau[\alpha := \tau']$, c'est-à-dire le type τ dans lequel chaque occurrence du paramètre α a été remplacée par τ'.

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

La notion de substitution de type $\tau[\alpha := \tau']$ est définie par des équations équivalentes à celles utilisées pour définir la substitution d'expressions au chapitre 2.

$$\begin{array}{rcl} \texttt{int}[\alpha := \tau] & = & \texttt{int} \\ \beta[\alpha := \tau] & = & \begin{cases} \tau & \text{si } \alpha = \beta \\ \beta & \text{si } \alpha \neq \beta \end{cases} \\ (\tau_1 \to \tau_2)[\alpha := \tau] & = & \tau_1[\alpha := \tau] \to \tau_2[\alpha := \tau] \\ (\forall \beta.\tau')[\alpha := \tau] & = & \begin{cases} \forall \beta.\tau' & \text{si } \alpha = \beta \\ \forall \beta.\tau'[\alpha := \tau] & \text{si } \alpha \neq \beta \text{ et } \beta \notin \mathrm{fv}(\tau) \end{cases} \end{array}$$

La notion de variable de type libre est de même définie similairement aux variables libres d'une expression.

$$\begin{array}{rcl} \mathrm{fv}(\texttt{int}) & = & \emptyset \\ \mathrm{fv}(\alpha) & = & \{\alpha\} \\ \mathrm{fv}(\tau_1 \to \tau_2) & = & \mathrm{fv}(\tau_1) \cup \mathrm{fv}(\tau_2) \\ \mathrm{fv}(\forall \alpha.\tau) & = & \mathrm{fv}(\tau) \setminus \{\alpha\} \end{array}$$

Généralisation Lorsqu'une expression admet un type τ contenant un paramètre α, et que ce paramètre *n'est contraint d'aucune manière* par le contexte Γ, alors on peut considérer l'expression e comme polymorphe et lui donner le type $\forall \alpha.\tau$.

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

Dans la règle d'inférence, notez que la condition demandant que le paramètre α ne soit pas contraint par le contexte est traduite par la non-apparition de α dans le contexte Γ.

Formellement, l'ensemble des variables de type libres d'un environnement $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ est défini par l'équation.

$$\mathrm{fv}(x_1 : \tau_1, \ldots, x_n : \tau_n) \quad = \quad \bigcup_{1 \leq i \leq n} \mathrm{fv}(\tau_i)$$

Notez que l'on ne parle ici que des variables *de type*. Les x_i, qui sont des variables du langage, ne sont pas concernées.

Exemples et contre-exemples Nous pouvons maintenant donner à la fonction identité `fun x -> x` le type polymorphe $\forall \alpha.\alpha \to \alpha$, exprimant que cette fonction admet un argument de *n'importe quel type* et renvoie un résultat *du même type*.

$$\dfrac{\dfrac{\dfrac{}{\texttt{x} : \alpha \vdash \texttt{x} : \alpha}}{\vdash \texttt{fun x -> x} : \alpha \to \alpha} \qquad \alpha \notin \mathrm{fv}(\emptyset)}{\vdash \texttt{fun x -> x} : \forall \alpha.\alpha \to \alpha}$$

Notez que la clé de ce raisonnement est la possibilité de donner à `fun x -> x` le type $\alpha \to \alpha$ dans le contexte vide, et donc a fortiori dans un contexte n'imposant aucune contrainte à α.

Il devient alors possibles de typer l'expression `let f = fun x -> x in f f`. En effet, dans la partie de l'arbre de dérivation concernant l'expression `f f`, nous avons un environnement $\Gamma = \{ f : \forall\alpha.\alpha \to \alpha \}$ qui permet de compléter la dérivation ainsi.

$$\dfrac{\dfrac{\overline{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \quad \dfrac{\overline{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : \texttt{int} \to \texttt{int}}}{\Gamma \vdash \texttt{f f} : \texttt{int} \to \texttt{int}}$$

Notez que ce n'est pas la seule solution possible : on aurait également pu laisser à la place du type concret `int` n'importe quelle variable de type β, et même aboutir à la conclusion que le type de `f f` pouvait être généralisé, puisque β n'apparaît pas libre dans Γ.

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : (\beta \to \beta) \to (\beta \to \beta)} \quad \dfrac{\overline{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \texttt{f} : \beta \to \beta}}{\Gamma \vdash \texttt{f f} : \beta \to \beta} \quad \beta \notin \text{fv}(\Gamma)}{\Gamma \vdash \texttt{f f} : \forall\beta.\beta \to \beta}$$

Notre système en revanche *ne permet pas* de donner à la fonction identité `fun x -> x` le type $\alpha \to \forall\alpha.\alpha$. En effet, pour cela il faudrait pouvoir, dans un contexte $\Gamma = \{ \texttt{x} : \alpha \}$, donner à x le type $\forall\alpha.\alpha$. Or notre axiome permet seulement de dériver le jugement $\Gamma \vdash \texttt{x} : \alpha$, dans lequel α ne peut pas être généralisé, puisque justement α apparaît dans Γ. Nous ne pouvons donc (heureusement) pas utiliser le polymorphisme pour typer l'expression mal formée `(fun x -> x) 5 37`.

Démontrons encore que la fonction de composition `fun f -> fun g -> fun x -> g (f x)` admet le type polymorphe $\forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$. On note Γ l'environnement $\{ \texttt{f} : \alpha \to \beta, \texttt{g} : \beta \to \gamma, \texttt{x} : \alpha \}$. On peut alors produire la dérivation suivante (dans cette dérivation, on a contracté les trois applications successives de la règle de généralisation, ainsi que les trois applications successives de la règle de typage des fonctions).

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash \texttt{f} : \alpha \to \beta} \quad \overline{\Gamma \vdash \texttt{x} : \alpha}}{\Gamma \vdash \texttt{f x} : \beta} \quad \overline{\Gamma \vdash \texttt{g} : \beta \to \gamma}}{\Gamma \vdash \texttt{g (f x)} : \gamma}}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : (\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)} \quad \alpha, \beta, \gamma \notin \text{fv}(\emptyset)$$
$$\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : \forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$$

Exercice : montrer que l'on peut également donner à cette même fonction de composition le type $\forall\alpha\beta.(\alpha \to \beta) \to \forall\gamma.(\beta \to \gamma) \to (\alpha \to \gamma)$.

Système de Hindley-Milner Sans annotations de la part du programmeur, les deux problèmes suivants relatifs au typage polymorphe de FUN sont indécidables :

- l'inférence : c'est-à-dire partant d'une expression e, dire s'il existe un type τ tel que l'on puisse justifier $\Gamma \vdash e : \tau$,
- la vérification : c'est-à-dire partant d'une expression e et d'un type τ, dire si l'on peut ou non justifier $\Gamma \vdash e : \tau$.

Ces résultats d'indécidabilité valent encore évidemment pour tout langage étendant ce noyau.

Pour permettre la vérification algorithmique du bon typage d'un programme, voire l'inférence des types, il faut donc soit imposer un certain nombre d'annotations au programmeur, soit restreindre les possibilités de recours au polymorphisme. Selon les langages, l'équilibre choisi entre la quantité d'annotations requise et l'expressivité du système de types varie.

En caml, le polymorphisme est restreint par le fait que les quantificateurs ne peuvent être qu'implicites : on ne peut pas écrire de quantificateur explicite dans un type, mais en revanche chaque variable de type globalement libre dans le type d'une expression est considérée comme quantifié universellement. Ainsi, le type caml de la première projection d'une paire

```
fst: 'a * 'b -> 'a
```

est en réalité le type universellement quantifié $\forall\alpha\beta.\alpha \times \beta \to \alpha$. De même, le type caml de l'itérateur de liste

```
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

doit être lu comme $\forall\alpha\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\,\texttt{list} \to \alpha$.

Ce polymorphisme restreint, partagé par tous les langages de la famille ML, est le ***système de Hindley-Milner***. Il autorise le quantificateur universel $\forall$ uniquement « en tête » et correspond à séparer la notion de ***type*** τ sans quantificateur de la notion de ***schéma de type*** σ qui est un type devant lequel on a éventuellement ajouté un ou plusieurs quantificateurs.

Ainsi pour FUN :

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \tau \to \tau \\ & | & \alpha \\ \sigma & ::= & \forall\alpha_1 \ldots \forall\alpha_n.\tau \end{array}$$

Dans ce système, on peut ainsi exprimer les schémas de type $\forall\alpha.\alpha \to \alpha$ et $\forall\alpha\beta\gamma.(\alpha \to \beta) \to (\beta \to \gamma) \to (\alpha \to \gamma)$, mais pas un type de la forme $(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$.

Dans le système de Hindley-Milner, on adapte les notions de contexte et de jugement de typage pour autoriser l'association d'un schéma de types à une variable ou une expression :

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \sigma$$

Notez cependant qu'un schéma avec zéro quantificateur n'est rien d'autre qu'un type simple : cette forme autorise donc de manière générale aussi bien les schémas que les types simples.

Les règles de typage sont également ajustées, de manière à n'autoriser les schémas de type qu'aux endroits où ils sont effectivement acceptables.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha := \tau]} \qquad \frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}$$

En l'occurrence la généralisation du type d'une expression est acceptée à deux endroits :

— à la racine, et

— pour l'argument d'un `let`.

Vous pouvez observer ceci dans la règle du typage du `let`. À l'inverse, vous pouvez également observer que les règles de typage d'une fonction ou d'une application imposent que les types de l'argument et du résultat soient tous les deux des types simples.

Le système de types de Hindley-Milner possède deux propriétés intéressantes :

— la vérification *et* l'inférence de types sont décidables (voir section suivante),

— le système est ***sûr***, c'est-à-dire que l'exécution d'un programme bien typé ne peut pas buter sur une incohérence (voir fin du chapitre).

À la fin de ce chapitre, nous verrons comment une telle notion de sûreté peut être formalisée et démontrée. Avant cela, nous allons formaliser la manière dont s'exécute un programme : c'est la question de la ***sémantique***.

5.5 Inférence de types

L'écriture de notre vérificateur de types simples pour FUN était relativement simple grâce à deux caractéristiques de ce système :

- les règles d'inférence étaient purement *syntaxiques*, c'est-à-dire que pour chaque forme d'expression il n'y avait qu'une seule règle d'inférence potentiellement applicable (en anglais on dit *syntax-directed*),
- on avait demandé une annotation pour aider le typage au seul endroit où on n'avait pas de manière simple de choisir le type d'un élément (en l'occurrence, le paramètre d'une fonction).

Dans le système de Hindley-Milner en revanche, les deux règles d'instanciation et de généralisation sont applicables à n'importe quelle forme d'expression a priori, et brisent donc la première propriété. En outre on vise l'***inférence complète***, c'est-à-dire sans aucune annotation.

Système de Hindley-Milner syntaxique Dans une première étape, nous allons donc donner une variante syntaxique du système de Hindley-Milner en restreignant les possibilités d'appliquer les règles de généralisation et d'instanciation.

- On n'autorise l'instanciation d'une variable de type qu'au moment de récupérer dans l'environnement le schéma de type associé à une variable. On obtient cet effet en supprimant la règle d'instanciation et en remplaçant l'axiome $\Gamma \vdash x : \Gamma(x)$ par une règle qui combine les deux effets suivants :
  1. récupérer le schéma de type σ associé à x dans Γ,
  2. instancier *toutes* les variables universelles de σ (on obtient donc un type simple).
- Symétriquement, on n'autorise la généralisation d'une variable de type qu'au moment d'une définition `let`. On obtient cet effet en supprimant la règle de généralisation et en remplaçant la règle d'inférence pour $\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2$ par une variante qui combine les effets suivants :
  1. typer l'expression e_1 dans l'environnement Γ, on note τ_1 le type obtenu,
  2. obtenir un schéma σ_1 en généralisant *toutes* les variables de τ_1 qui peuvent l'être,
  3. typer e_2 dans l'environnement étendu où x est associé à σ_1.

Notez au passage que l'on n'autorise plus les schémas de types ailleurs que dans le contexte.

Voici une dérivation de typage dans ce système, avec $\Gamma_1 = \{x : \alpha \to \alpha, y : \alpha\}$ et $\Gamma_2 = \{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{}{\Gamma_1 \vdash y : \alpha}}{\Gamma_1 \vdash x\,y : \alpha}}{x : \alpha \to \alpha, y : \alpha \vdash x\,(x\,y) : \alpha}}{x : \alpha \to \alpha \vdash \texttt{fun } y \texttt{ -> } x\,(x\,y) : \alpha \to \alpha}}{\vdash \texttt{fun } x \texttt{ -> fun } y \texttt{ -> } x\,(x\,y) : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \dfrac{\dfrac{}{\Gamma_2 \vdash f : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \quad \dfrac{\dfrac{\dfrac{}{\Gamma_2, z : \texttt{int} \vdash z : \texttt{int}} \quad \dfrac{}{\Gamma_2, z : \texttt{int} \vdash 1 : \texttt{int}}}{\Gamma_2, z : \texttt{int} \vdash z\texttt{+1} : \texttt{int}}}{\Gamma_2 \vdash \texttt{fun } z \texttt{ -> } z\texttt{+1} : \texttt{int} \to \texttt{int}}}{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \vdash f(\texttt{fun } z \texttt{ -> } z\texttt{+1}) : \texttt{int} \to \texttt{int}}$$

$$\vdash \texttt{let } f \texttt{ = fun } x \texttt{ -> fun } y \texttt{ -> } x\,(x\,y) \texttt{ in } f(\texttt{fun } z \texttt{ -> } z\texttt{+1}) : \texttt{int} \to \texttt{int}$$

On peut démontrer que cette variante syntaxique du système de Hindley-Milner est équivalente à la version d'origine, c'est-à-dire qu'elle permet de dériver les mêmes jugements de typage.

Génération de contraintes et unification Reste donc à réussir à se passer totalement d'annotations. L'algorithme W utilise pour cela la stratégie suivante.

- À chaque fois que l'on doit introduire un type que l'on ne sait pas calculer immédiatement, on introduit à la place une nouvelle variable de type. Cela vaut pour le type du paramètre d'une fonction, mais aussi pour les types instanciant les variables universelles d'un schéma $\Gamma(x)$.
- On détermine la valeur de ces variables de types *plus tard*, au moment de résoudre les contraintes associées aux règles de typage de l'application ou de l'addition.

Lorsqu'une règle de typage impose une égalité entre deux types τ_1 et τ_2 contenant des variables de types α_1, ..., α_n, on cherche à ***unifier*** ces deux types, c'est-à-dire à trouver une instanciation f des variables α_i telle que $f(\tau_1) = f(\tau_2)$.
Exemples d'unification :

- Si $\tau_1 = \alpha \to \texttt{int}$ et $\tau_2 = (\texttt{int} \to \texttt{int}) \to \beta$, on peut unifier τ_1 et τ_2 avec l'instanciation $[\alpha \mapsto \texttt{int} \to \texttt{int}, \beta \mapsto \texttt{int}]$.
- Si $\tau_1 = (\alpha \to \texttt{int}) \to (\alpha \to \texttt{int})$ et $\tau_2 = \beta \to \beta$, on peut unifier τ_1 et τ_2 avec l'instanciation $[\beta \mapsto \alpha \to \texttt{int}]$.
- On ne peut pas unifier $\alpha \to \texttt{int}$ et $\texttt{int}$.
- On ne peut pas unifier $\alpha \to \texttt{int}$ et α.

Critères d'unification :

- τ est toujours unifié à lui-même,
- pour unifier $\tau_1 \to \tau_1'$ et $\tau_2 \to \tau_2'$ on unifie τ_1 avec τ_2 et on unifie τ_1' avec τ_2',
- pour unifier τ et une variable α, lorsque α n'apparaît pas dans τ, on substitue α par τ partout (si α apparaît dans τ l'unification est impossible),
- dans tous les autres cas l'unification est impossible.

Algorithme W sur un exemple On regarde l'expression `let` f `=` `fun` x `->` `fun` y `->` $x\ (x\ y)$ `in` f `(fun` z`->` z`+1)`. Pour inférer son type, on s'intéresse d'abord à `fun` x `->` `fun` y `->` $x\ (x\ y)$. On procède ainsi.

- À x on donne le type α, avec α une nouvelle variable de type,
- à y on donne le type β, avec β une nouvelle variable de type,
- on type ensuite l'expression $x\ (x\ y)$.
  - L'application $x\ y$ demande que le type α de x soit un type fonctionnel, dont le paramètre correspond au type β de y. Il faut donc unifier α avec $\beta \to \gamma$, pour γ une nouvelle variable de type. On fixe ainsi $\alpha = \beta \to \gamma$.
  - L'application $x\ y$ a donc le type γ.
  - L'application $x\ (x\ y)$ demande que le type $\alpha = \beta \to \gamma$ de x soit un type fonctionnel dont le paramètre correspond au type γ de $x\ y$. Il faut donc unifier $\beta \to \gamma$ avec $\gamma \to \delta$, pour δ une nouvelle variable de type. On fixe ainsi $\gamma = \delta = \beta$.

  On en déduit que l'application $x\ (x\ y)$ a le type β.
- Finalement, `fun` x `->` `fun` y `->` $x\ (x\ y)$ reçoit le type $\alpha \to (\beta \to \beta)$, c'est-à-dire $(\beta \to \beta) \to (\beta \to \beta)$, et dans le contexte de typage vide on peut généraliser ce type en $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$.

Dans une deuxième étape, on s'intéresse à l'expression f `(fun` z`->` z`+1)`, dans un contexte où f a le type généralisé $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$. Pour inférer le type de cette application on commence par typer les deux sous-expressions puis on résoud les contraintes.

- Pour f on récupère le schéma de type $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$ dans le contexte, et on instantie la variable universelle β avec un nouvelle variable de type ζ. On obtient donc pour f le type $(\zeta \to \zeta) \to (\zeta \to \zeta)$.
- Typage de `fun` z `->` z`+1`.
  - À z on donne le type η, avec η une nouvelle variable de type,
  - puis on type l'addition z`+1`.
    - z a le type η, qu'il faut unifier avec `int`. On fixe donc $\eta = \texttt{int}$.
    - 1 a le type `int`, qu'il faut unifier avec `int` : rien à faire.

    Donc z`+1` a le type `int`.

  Donc `fun` z `->` z`+1` a le type $\eta \to \texttt{int}$, c'est-à-dire $\texttt{int} \to \texttt{int}$.
- Pour résoudre l'application elle-même, il faut unifier le type $(\zeta \to \zeta) \to (\zeta \to \zeta)$ de f avec le type $(\texttt{int} \to \texttt{int}) \to \theta$ d'une fonction prenant un paramètre du type $\texttt{int} \to \texttt{int}$ (le type de `fun` z `->` z`+1`), avec θ une nouvelle variable de type. On fixe ainsi $\zeta = \texttt{int}$ et θ, c'est-à-dire $\texttt{int} \to \texttt{int}$.

Finalement, l'expression f `(fun` z `->` z`+1)` a le type $\theta = \texttt{int} \to \texttt{int}$.

On conclut donc bien

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

Algorithme W, en caml On conserve pour les expressions FUN la syntaxe abstraite suivante, sans annotation de types.

```
type expression =
  | Var of string
  | Cst of int
  | Add of expression * expression
  | Fun of string * expression
  | App of expression * expression
  | Let of string * expression * expression
```

On ajoute aux types simples une notion de variable de type, et on définit un schéma de type par une structure avec un type simple `typ` et un ensemble `vars` de variables de types quantifiées universellement.

```
type typ =
  | Tint
  | Tarrow of typ * typ
  | Tvar  of string

module VSet = Set.Make(String)
type schema = { vars: VSet.t; typ: typ }
```

Un environnement de typage associe un schéma de type à chaque variable de programme.

```
module SMap = Map.Make(String)
type env = schema SMap.t
```

On peut alors définir une fonction `type_inference: expression -> typ` calculant un type aussi général que possible pour l'expression donnée en argument. Cette fonction embarque une fonction auxiliaire `new_var: unit -> string` pour la création de nouvelles variables de types.

```
let type_inference t =
  let new_var =
    let cpt = ref 0 in
    fun () -> incr cpt; Printf.sprintf "tvar_%i" !cpt
  in
```

À mesure que des contraintes seront découvertes et analysées, les variables de types introduites avec `new_var` sont destinées à être associées à des types concrets (ou a minima à des types plus précis). Plutôt que de faire les remplacements partout à chaque fois qu'une nouvelle association est trouvée, on mémorise ces associations dans une table de hachage `subst` qui va grandir progressivement.

```
  let subst = Hashtbl.create 32 in
```

En conséquence, les types manipulés lors de l'inférence vont contenir des variables de types, dont certaines seront associées à une définition dans `subst`. Pour décoder un tel type, on se donne des fonctions auxiliaires de dépliage `unfold` et `unfold_full`, qui remplacent dans un type τ donné en argument les variables de types pour lesquelles on a une définition dans `subst`. La fonction `unfold` fait ce remplacement « en surface », pour découvrir la forme superficielle du type et permettre de distinguer entre les cas `Tint`, `Tarrow` ou `Tvar`. La fonction `unfold_full` fait un remplacement intégral pour connaître le type complet (cette dernière ne servira que pour afficher le verdict à la fin).

```
  let rec unfold t = match t with
    | Tint | Tarrow(_, _) -> t
    | Tvar a ->
       if Hashtbl.mem subst a then
         unfold (Hashtbl.find subst a)
       else
         t
  in
  let rec unfold_full t = match unfold t with
    | Tarrow(t1, t2) -> Tarrow(unfold_full t1, unfold_full t2)
    | t -> t
  in
```

Exemple d'utilisation du dépliage : pour déterminer si une variable de type α apparaît dans un type τ, on raisonne comme d'habitude sur la forme du type τ, mais en intercalant un appel à la fonction de dépliage pour réaliser au préalable les substitutions enregistrées dans `subst`.

```
  let rec occur a t = match unfold t with
    | Tint -> false
    | Tvar b -> a=b
    | Tarrow(t1, t2) -> occur a t1 || occur a t2
  in
  ...
```

Le cœur de l'algorithme W travaille classiquement sur la forme de l'expression analysée. Dans le cas d'une constante, on se contente de renvoyer le type de base `Tint`. Dans le cas d'une addition on infère un type pour chacun des opérandes et on vérifie que les types `t1` et `t2` obtenus sont bien compatibles avec le type `Tint` attendu. Cette vérification de compatibilité est faite par une fonction auxiliaire `unify` qui, éventuellement, enregistrera de nouvelles associations entre des variables de types et des types concrets.

```
let rec w e env = match e with
  | Cst _ ->
     Tint

  | Add(e1, e2) ->
     let t1 = w e1 env in
     let t2 = w e2 env in
     unify t1 Tint; unify t2 Tint;
     Tint
```

Dans le cas d'une variable, on instancie le schéma de types obtenu dans l'environnement à l'aide d'une fonction auxiliaire `instantiate` qui remplace chaque variable quantifiée universellement par une variable de types fraîche.

```
  | Var x ->
     instantiate (SMap.find x env)
```

Inversement, dans le cas d'un `let` on généralise le type inféré pour l'expression `e1` à l'aide d'une fonction auxiliaire `generalize` qui produit un schéma de type où chaque variable qui peut l'être est quantifiée universellement.

```
  | Let(x, e1, e2) ->
     let t1 = w e1 env in
     let st1 = generalize t1 env in
     let env' = SMap.add x st1 env in
     w e2 env'
```

Pour typer une fonction, on introduit une nouvelle variable pour le type du paramètre. Le type d'un paramètre ne pouvant pas être généralisé, on fixe immédiatement que l'ensemble des variables quantifiées universellement est vide. On infère alors le type du corps de la fonction dans cet environnement étendu.

```
  | Fun(x, e) ->
     let v = new_var() in
     let env = SMap.add x { vars = VSet.empty; typ = Tvar v } env in
     let t = w e env in
     Tarrow(Tvar v, t)
```

Pour typer une application on infère d'abord un type pour chacune des deux sous-expressions. Il faut ensuite résoudre la contrainte de la règle d'application, à savoir que le type `t1` du membre gauche `e1` doit être un type de fonction, et que le type `t2` du membre droit doit être le type attendu du paramètre de cette fonction.

```
  | App(e1, e2) ->
     let t1 = w e1 env in
     let t2 = w e2 env in
     let v = Tvar (new_var()) in
     unify t1 (Tarrow(t2, v));
     v

in
unfold_full (w t SMap.empty)
```

Définition des fonctions auxiliaires citées ci-dessus. Les contraintes sont résolues par un algorithme d'unification, qui prend en paramètres deux types τ_1 et τ_2 et cherche à donner des définitions aux variables de types contenues dans τ_1 et τ_2 de sorte que ces deux types deviennent égaux. Si les deux types ont la même forme, alors il n'y a rien de particulier à faire, si ce n'est poursuivre récursivement l'unification sur les éventuels sous-termes.

```
let rec unify t1 t2 = match unfold t1, unfold t2 with
  | Tint, Tint ->
    ()
  | Tarrow(t1, t1'), Tarrow(t2, t2') ->
    unify t1 t2; unify t1' t2'
```

Lorsqu'apparaît une variable en revanche, on a plusieurs issues possibles :

- Si τ_1 et τ_2 sont la même variable, il n'y a rien à faire : les deux types sont déjà identiques.
- Si l'un des types est une variable α, et si l'autre est une variable différente ou un type d'une autre forme τ, alors on va ajouter une association $[\alpha \mapsto \tau]$ dans `subst`. À noter cependant : si la variable α apparaît dans τ, alors on échoue à la place car il n'est pas possible que α fasse partie de sa propre définition.

```
  | Tvar a, Tvar b when a=a ->
    ()
  | Tvar a, t | t, Tvar a ->
    if occur a t then
      failwith "unification error"
    else
      Hashtbl.add subst a t
```

Enfin, dans tous les autres cas l'unification échoue.

```
  | _, _ ->
    failwith "unification error"
in
```

La fonction auxiliaire d'instanciation génère une nouvelle variable de type pour chaque variable universelle du schéma de types donné en argument, et opère un remplacement.

```
let instantiate s =
  let renaming = VSet.fold
           (fun v r -> SMap.add v (Tvar(new_var())) r)
           s.vars
           SMap.empty
  in
  let rec rename t = match unfold t with
    | Tvar a as t -> (try SMap.find a renaming with Not_found -> t)
    | Tint -> Tint
    | Tarrow(t1, t2) -> Tarrow(rename t1, rename t2)
  in
  rename s.typ
in
```

La fonction auxiliaire de généralisation prend en paramètres un type τ et un environnement Γ, et calcule l'ensemble des variables libres de τ qui n'apparaissent pas dans l'environnement Γ. Ces variables sont alors indiquées comme quantifiées universellement.

```
let rec fvars t = match unfold t with
  | Tint -> VSet.empty
  | Tarrow(t1, t2) -> VSet.union (fvars t1) (fvars t2)
  | Tvar x -> VSet.singleton x
in
let rec schema_fvars s =
  VSet.diff (fvars s.typ) s.vars
in
let generalize t env =
  let fvt = fvars t in
  let fvenv = SMap.fold
              (fun _ s vs -> VSet.union (schema_fvars s) vs)
              env
              VSet.empty
  in
  { vars = VSet.diff fvt fvenv; typ=t }
in
```

5.6 Sémantique naturelle

La ***sémantique*** décrit la signification et le comportement des programmes. Un langage de programmation est généralement accompagné d'une description plus ou moins informelle de ce qu'il faut attendre du comportement d'un programme écrit dans ce langage. Voici un extrait de la spécification de Java :

> *The Java programming language guarantees that the operands of operators appear to be evaluated in a specific order, namely, from left to right. It is recommended that code do not rely crucially on this specification.*

Ce genre de document comporte souvent une quantité plus ou moins grande d'imprécisions voire d'ambiguïtés. À l'inverse, on peut également donner à un langage une ***sémantique formelle***, c'est-à-dire une caractérisation mathématique des calculs décrits par un programme. Ce cadre plus rigoureux permet notamment de *raisonner* sur l'exécution des programmes.

Sémantique en appel par valeur Au chapitre 2 nous avons déjà pu voir comment définir une fonction d'interprétation des expressions d'un langage de programmation, c'est-à-dire une fonction eval qui, étant donnés une expression e et un environnement ρ associant des valeurs aux variables libres de e, renvoie le résultat de l'évaluation de e.

Pour notre fragment du langage FUN, une telle fonction peut être définie par les équations suivantes.

$$\begin{aligned}
\mathrm{eval}(n, \rho) &= n \\
\mathrm{eval}(e_1 \texttt{ + } e_2, \rho) &= \mathrm{eval}(e_1, \rho) + \mathrm{eval}(e_2, \rho) \\
\mathrm{eval}(x, \rho) &= \rho(x) \\
\mathrm{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2, \rho) &= \mathrm{eval}(e_2, \rho[x = \mathrm{eval}(e_1, \rho)]) \\
\mathrm{eval}(\texttt{fun } x \texttt{ -> } e, \rho) &= \mathrm{Clos}(x, e, \rho) \\
\mathrm{eval}(e_1\ e_2, \rho) &= \mathrm{eval}(e, \rho'[x = \mathrm{eval}(e_2, \rho)]) \\
&\quad \text{si } \mathrm{eval}(e_1, \rho) = \mathrm{Clos}(x, e, \rho')
\end{aligned}$$

Notez dans l'équation concernant l'addition que le symbole + dans e_1 `+` e_2 est un élément de syntaxe du langage FUN reliant deux expressions, tandis que l'opérateur + dans $\mathrm{eval}(e_1, \rho) + \mathrm{eval}(e_2, \rho)$ est l'addition mathématique des valeurs v_1 et v_2 produites par l'évaluation des deux expressions e_1 et e_2. Rappelons également que la forme $\mathrm{Clos}(x, e, \rho)$ désigne une fermeture, c'est-à-dire une fonction accompagnée de son environnement.

Notez que l'environnement ρ manipulé par cette fonction d'évaluation, et la nécessité qui en découle d'introduire une notion de fermeture pour représenter les fonctions comme des valeurs, est un dispositif qui était avant tout destiné à produire un interprète efficace. Pour une spécification mathématique de la valeur que doit produire l'évaluation d'une expression, et dans un tel cadre purement fonctionnel, on peut ne manipuler que des expressions dans lesquelles chaque variable est remplacée par sa valeur, et ainsi contourner cette notion d'environnement. On aurait ainsi une définition comme

$$\begin{aligned}
\mathrm{eval}(n) &= n \\
\mathrm{eval}(e_1 \texttt{ + } e_2) &= \mathrm{eval}(e_1) + \mathrm{eval}(e_2) \\
\mathrm{eval}(x) &= \text{indéfini} \\
\mathrm{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) &= \mathrm{eval}(e_2[x := \mathrm{eval}(e_1)]) \\
\mathrm{eval}(\texttt{fun } x \texttt{ -> } e) &= \texttt{fun } x \texttt{ -> } e \\
\mathrm{eval}(e_1\ e_2) &= \mathrm{eval}(e[x = \mathrm{eval}(e_2)]) \\
&\quad \text{si } \mathrm{eval}(e_1) = \texttt{fun } x \texttt{ -> } e
\end{aligned}$$

où la substitution $e[x := e']$ dans l'expression e de chaque occurrence de la variable x par l'expression e' est définie selon les lignes habituelles par

$$\begin{aligned}
n[x := e'] &= n \\
(e_1 \texttt{ + } e_2)[x := e'] &= e_1[x := e'] \texttt{ + } e_2[x := e'] \\
y[x := e'] &= \begin{cases} e' & \text{si } x = y \\ y & \text{sinon} \end{cases} \\
(\texttt{let } y \texttt{ = } e_1 \texttt{ in } e_2)[x := e'] &= \begin{cases} \texttt{let } y \texttt{ = } e_1[x := e'] \texttt{ in } e_2 & \text{si } x = y \\ \texttt{let } y \texttt{ = } e_1[x := e'] \texttt{ in } e_2[x := e'] & \text{si } x \neq y \text{ et } y \notin \mathrm{fv}(e') \end{cases} \\
(\texttt{fun } y \texttt{ -> } e)[x := e'] &= \begin{cases} \texttt{fun } y \texttt{ -> } e & \text{si } x = y \\ \texttt{fun } y \texttt{ -> } e[x := e'] & \text{si } x \neq y \text{ et } y \notin \mathrm{fv}(e') \end{cases} \\
(e_1\ e_2)[x := e'] &= e_1[x := e']\ e_2[x := e']
\end{aligned}$$

Cette approche de la sémantique d'un programme est surtout adaptée à la description de programmes déterministes et dont l'exécution de passe bien.

Une approche limitant ces présupposés consiste à définir la sémantique comme une relation entre les expressions et les valeurs. On noterait ainsi

$$e \implies v$$

pour toute paire d'une expression e et d'une valeur v telle que l'expression e *peut* s'évaluer en la valeur v.

Cette relation, appelée ***sémantique naturelle***, ou ***sémantique à grands pas***, est définie par des règles d'inférence et spécifie les évaluations possibles des expressions. Un compilateur est tenu de respecter la sémantique de son langage source.

Pour asseoir notre formalisation, précisons l'ensemble des valeurs que nous considérons : les nombres entiers et les fonctions.

$$\begin{array}{rcl} v & ::= & n \\ & | & \texttt{fun } x \texttt{ -> } e \end{array}$$

Les règles d'inférence vont ensuite traduire les équations définissant notre précédente fonction eval.

- $\text{eval}(n) = n$. Une constante entière est sa propre valeur. La règle associée est un axiome.

$$\frac{}{n \implies n}$$

- $\text{eval}(e_1 \texttt{ + } e_2) = \text{eval}(e_1) + \text{eval}(e_2)$. La valeur d'une expression d'addition est obtenue en ajoutant les valeurs de chacune des deux sous-expressions.

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 \texttt{ + } e_2 \implies n_1 + n_2}$$

  Notez que cette règle ne peut s'appliquer que si les valeurs n_1 et n_2 associées à e_1 et e_2 sont bien des nombres.
- $\text{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \text{eval}(e_2[x := \text{eval}(e_1)])$. La valeur d'une expression e_2 comportant une variable locale x est obtenue en évaluant e_2 après substitution de x par la valeur de l'expression e_1 associée.

$$\frac{e_1 \implies v_1 \qquad e_2[x := v_1] \implies v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \implies v}$$

- $\text{eval}(\texttt{fun } x \texttt{ -> } e) = \texttt{fun } x \texttt{ -> } e$. Une fonction est sa propre valeur. Comme pour les constantes, la règle associée est un axiome.

$$\frac{}{\texttt{fun } x \texttt{ -> } e \implies \texttt{fun } x \texttt{ -> } e}$$

- $\text{eval}(e_1\ e_2) = \text{eval}(e[x = \text{eval}(e_2)])$ si $\text{eval}(e_1) = \texttt{fun } x \texttt{ -> } e$. Pour que la valeur d'une application soit bien définie, il faut que son membre gauche e_1 ait pour valeur une fonction. La valeur de l'application est alors obtenue en substituant le paramètre formel dans le corps de la fonction par la valeur de l'argument e_2, puis en évaluant l'expression obtenue.

$$\frac{e_1 \implies \texttt{fun } x \texttt{ -> } e \qquad e_2 \implies v_2 \qquad e[x := v_2] \implies v}{e_1\ e_2 \implies v}$$

Nous obtenons donc pour notre fragment du langage FUN cinq règles d'inférence, et nous pouvons justifier un jugement sémantique de la forme $e \implies v$ à l'aide d'une dérivation.

$$\dfrac{\dfrac{}{\texttt{fun x->x+x} \implies \texttt{fun x->x+x}} \qquad \dfrac{\dfrac{\dfrac{}{20 \implies 20} \qquad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21} \qquad \dfrac{\dfrac{}{21 \implies 21} \qquad \dfrac{}{21 \implies 21}}{21\texttt{+}21 \implies 42}}{(\texttt{fun } x \texttt{ -> } x \texttt{ + } x)\,(20 \texttt{ + } 1) \implies 42}}{\texttt{let } f \texttt{ = fun } x \texttt{ -> } x \texttt{ + } x \texttt{ in } f(20 \texttt{ + } 1) \implies 42}$$

Notez que la règle proposée pour l'application de fonction évalue l'argument e_2 avant de le substituer dans le corps de la fonction. Ce comportement vient de la fonction d'interprétation qui nous a servi de base, et qui réalisait la stratégie d'*appel par valeur*.

Sémantique en appel par nom On pourrait définir une variante de cette sémantique basée sur la stratégie d'*appel par nom*. Cette variante consiste essentiellement à remplacer la règle relative à l'application par la version plus simple suivante

$$\frac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e[x := e_2] \implies v}{e_1\ e_2 \implies v}$$

où l'argument e_2 est substitué tel quel.

Dans une sémantique en appel par nom on peut également, optionnellement, utiliser la variante suivante de la règle de `let`.

$$\frac{e_2[x := e_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

La sémantique en appel par nom est *presque* équivalente à la sémantique en appel par valeur : elles permettent en grande partie de dériver les mêmes jugements $e \implies v$ (la forme de la dérivation peut changer, mais le seul fait qui compte est qu'une dérivation *existe*). En l'occurrence, on peut dériver

$$\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f(20\ \texttt{+}\ 1) \implies 42$$

en appel par nom de la manière suivante.

$$\dfrac{\dfrac{}{\texttt{fun}\,x\texttt{->}x\texttt{+}x \implies \texttt{fun}\,x\texttt{->}x\texttt{+}x} \qquad \dfrac{\dfrac{\dfrac{}{20 \implies 20} \qquad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21} \qquad \dfrac{\dfrac{}{20 \implies 20} \qquad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21}}{(20\texttt{+}1)\texttt{+}(20\texttt{+}1) \implies 42}}{(\texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x)\,(20\ \texttt{+}\ 1) \implies 42}$$
$$\overline{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f(20\ \texttt{+}\ 1) \implies 42}$$

Question : comment l'appel par nom se traduit-il dans la forme de l'arbre ?

Insistons cependant sur le fait que ces deux sémantiques ne sont *pas totalement équivalentes* : il existe des jugements $e \implies v$ qui peuvent être dérivés dans l'une et non dans l'autre. *Question : pouvez-vous en trouver ?*

Raisonner sur la sémantique Du fait que la sémantique naturelle est définie par un système d'inférence, nous pouvons démontrer des propriétés des programmes et de leur sémantique en raisonnant par récurrence sur la dérivation d'un jugement $e \implies v$. Comme nous l'avons déjà vu avec la récurrence sur les jugements de typage, on a alors un cas de preuve par règle d'inférence de la sémantique, et les prémisses des règles fournissent à chaque fois des hypothèses de récurrence.

Utilisons la sémantique naturelle en appel par nom de FUN,

$$\frac{}{n \implies n} \qquad \frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1\ \texttt{+}\ e_2 \implies n_1 + n_2} \qquad \frac{e_2[x := e_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

$$\frac{}{\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e} \qquad \frac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e[x := e_2] \implies v}{e_1\ e_2 \implies v}$$

et démontrons que si $e \implies v$ alors v est une valeur et $\mathrm{fv}(v) \subseteq \mathrm{fv}(e)$, par récurrence sur la dérivation de $e \implies v$.

- Cas $n \implies n$: immédiat, car n est bien une valeur, et $\mathrm{fv}(n) \subseteq \mathrm{fv}(n)$.
- Cas $\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e$ immédiat de même.
- Cas $e_1\ \texttt{+}\ e_2 \implies n_1 + n_2$ avec $e_1 \implies n_1$ et $e_2 \implies n_2$. Par définition $n_1 + n_2$ est une valeur entière. En outre $\mathrm{fv}(n_1 + n_2) = \emptyset \subseteq \mathrm{fv}(e_1\ \texttt{+}\ e_2)$. *Note : les hypothèses de récurrence concernant e_1 et e_2 ne sont même pas utiles dans ce cas.*
- Cas $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v$ avec $e_2[x := e_1] \implies v$. La prémisse donne comme hypothèse de récurrence que $\mathrm{fv}(v) \subseteq \mathrm{fv}(e_2[x := e_1])$ (et que v est une valeur).
  On a besoin ici d'un lemme sur les variables libres d'un terme soumis à une substitution. On utilisera l'égalité suivante (démontrée juste après, par récurrence sur l'expression e).

$$\mathrm{fv}(e[x := e']) = (\mathrm{fv}(e) \setminus \{x\}) \cup \mathrm{fv}(e')$$

Avec le lemme, on a $\mathrm{fv}(v) \subseteq (\mathrm{fv}(e_2) \setminus \{x\}) \cup \mathrm{fv}(e_1)$. Or par définition

$$\mathrm{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus x)$$

On a donc bien $\mathrm{fv}(v) \subseteq \mathrm{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2)$.

— Cas $e_1\ e_2 \implies v$ avec $e_1 \implies \texttt{fun } x \texttt{ -> } e$ et $e[x := e_2] \implies v$. Les deux prémisses donnent comme hypothèses de récurrence que v est une valeur, que $\mathrm{fv}(\texttt{fun } x \texttt{ -> } e) \subseteq \mathrm{fv}(e_1)$ et que $\mathrm{fv}(v) \subseteq \mathrm{fv}(e[x := e_2])$. Avec le lemme précédent on a donc

$$\begin{array}{rcl} \mathrm{fv}(v) & \subseteq & \mathrm{fv}(e[x := e_2]) \\ & = & (\mathrm{fv}(e) \setminus \{x\}) \cup \mathrm{fv}(e_2) \\ & = & \mathrm{fv}(\texttt{fun } x \texttt{ -> } e) \cup \mathrm{fv}(e_2) \\ & \subseteq & \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \\ & = & \mathrm{fv}(e_1\ e_2) \end{array}$$

Premier lien entre typage et sémantique Avec cette formalisation de la sémantique naturelle, on peut démontrer l'énoncé suivant liant le typage et la sémantique d'une expression.

$$\text{Si} \quad \Gamma \vdash e : \tau \quad \text{et} \quad e \Longrightarrow v \quad \text{alors} \quad \Gamma \vdash v : \tau.$$

Cette propriété exprime que l'évaluation d'un programme préserve sa cohérence et son type.

Notez cependant que cette propriété part de l'hypothèse que l'évaluation du programme aboutit. Autrement dit, elle ne démontre pas que l'évaluation d'un programme bien typé aboutit, et ne dit rien des programmes qui plantent ni des programmes qui bouclent. Il va falloir préciser la formalisation de la sémantique pour permettre d'énoncer une véritable propriété de sûreté.

5.7 Sémantique opérationnelle à petits pas

Nous avons vu que la sémantique naturelle ne parle que des calculs qui aboutissent. En particulier, elle ne dit rien des calculs qui échouent à cause d'un blocage, comme l'évaluation de `5(37)`, ni des calculs qui ne terminent jamais, comme l'évaluation de `(fun x -> x x) (fun x -> x x)`. Elle n'est pas même capable de distinguer ces deux situations.

La ***sémantique à petits pas*** nous donne plus de précision en décomposant la relation d'évaluation $e \Longrightarrow v$ en une série d'étapes de calcul $e \to e_1 \to e_2 \to \ldots \to v$. On obtient alors les moyens de distinguer les trois situations suivantes :

— un calcul qui, après un nombre fini d'étapes, aboutit à un résultat :

$$e \to e_1 \to e_2 \to \ldots \to v$$

où v est une valeur,

— un calcul qui, après un certain nombre d'étapes, aboutit à un blocage :

$$e \to e_1 \to e_2 \to \ldots \to e_n$$

où e_n n'est pas valeur mais ne peut plus être évaluée,

— un calcul qui se poursuit indéfiniment :

$$e \to e_1 \to e_2 \to \ldots$$

où les étapes s'enchaînent à l'infini sans jamais aboutir à une expression finale.

Règles de calculs La sémantique à petits pas est définie par la relation $e \to e'$, appelée ***relation de réduction*** et désignant l'application d'un unique pas de calcul. Cette relation est à nouveau définie par un système de règles d'inférence. On fournit d'une part des règles de calcul élémentaires, formant des cas de base, et d'autre part des règles d'inférence permettant d'appliquer les règles de calcul dans des sous-expressions.

Commençons par détailler les axiomes pour notre fragment de FUN. Ils correspondent aux règles de calcul de base, appliquées directement à la racine d'une expression.

— Axiome pour l'application de fonction en appel par valeur. Si une fonction `fun` x `->` e est appliquée à une valeur v, alors on peut substituer v à chaque occurrence du paramètre formel x dans le corps de la fonction e.

$$\frac{}{(\texttt{fun}\ x\ \texttt{->}\ e)\ v \to e[x := v]}$$

L'application de cette règle suppose que l'argument de l'application a été évalué lors des étapes précédentes du calcul.

— Axiome pour le remplacement d'une variable locale par sa valeur.

$$\frac{}{\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \to e[x := v]}$$

Comme pour l'application de fonction, cette règle n'est applicable que si la valeur v associée à la variable x a déjà été calculée.

— Axiome pour l'addition.

$$\frac{n_1 + n_2 = n}{n_1\ \texttt{+}\ n_2 \to n}$$

Attention au jeu d'écriture ici : on passe de l'expression n_1 `+` n_2, où le symbole + est un élément de syntaxe, au résultat n de l'addition mathématique des deux nombres n_1 et n_2. Encore une fois en revanche, l'application de cette règle présuppose que les deux opérandes ont déjà été évalués, et que les valeurs obtenues sont bien des nombres.

Les règles d'inférence décrivent ensuite la manière dont les règles de base peuvent être appliquées à des sous-expressions.

— Règles d'inférence pour l'addition. Dans une expression de la forme e_1 `+` e_2, il est possible d'effectuer un pas de calcul dans l'une ou l'autre des deux sous-expressions e_1 et e_2. On traduit cela par deux règles d'inférences, une concernant chaque expression.

$$\frac{e_1 \to e_1'}{e_1\ \texttt{+}\ e_2 \to e_1'\ \texttt{+}\ e_2} \qquad \frac{e_2 \to e_2'}{e_1\ \texttt{+}\ e_2 \to e_1\ \texttt{+}\ e_2'}$$

Avec ces règles, on peut dériver le fait qu'une étape de calcul mène de l'expression `((1+2)+3)+(4+5)` à l'expression `(3+3)+(4+5)`.

$$\frac{\dfrac{\dfrac{1 + 2 = 3}{1\ \texttt{+}\ 2 \to 3}}{(1\ \texttt{+}\ 2)\ \texttt{+}\ 3 \to 3\ \texttt{+}\ 3}}{((1\ \texttt{+}\ 2)\ \texttt{+}\ 3)\ \texttt{+}\ (4\ \texttt{+}\ 5) \to (3\ \texttt{+}\ 3)\ \texttt{+}\ (4\ \texttt{+}\ 5)}$$

Notez que ces règles n'imposent rien sur l'ordre dans lequel évaluer les deux sous-expressions e_1 et e_2. Elles permettent même d'alterner des étapes de calcul de manière arbitraire entre les deux côtés. On peut ainsi dériver la suite d'étapes de calcul suivante.

`((1+2)+3)+(4+5)` → `(3+3)+(4+5)` → `(3+3)+9` → `6+9` → `15`

Si on voulait forcer l'évaluation des opérandes de gauche à droite, il faudrait remplacer la deuxième règle par la variante suivante, qui autorise l'application d'une étape de calcul dans l'opérande de droite sous condition que l'opérande de gauche soit déjà une valeur.

$$\frac{e_2 \to e_2'}{v_1\ \texttt{+}\ e_2 \to v_1\ \texttt{+}\ e_2'}$$

— Règles d'inférence pour une définition de variable locale. La règle ci-dessous autorise l'application d'un pas de calcul dans l'expressions e_1 définissant la valeur d'une variable locale x.

$$\frac{e_1 \to e_1'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2}$$

— Règles d'inférence pour les applications. Les deux règles ci-dessous autorisent l'application d'un pas de calcul du côté gauche d'une application sans condition, et du côté droit d'une application dont le côté gauche a déjà été évalué.

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

Notez qu'aucune règle n'autorise l'application d'un pas de calcul à l'intérieur du corps e d'une fonction `fun` x `->` e. En effet, une telle fonction est déjà une valeur, et n'a pas à être évaluée plus à ce stade ! Une fois que cette fonction aura reçu un argument v en revanche, et que cet argument aura été substitué à x dans e, alors le calcul pourra se poursuivre à cet endroit.

Bilan du système d'inférence définissant une sémantique à petits pas pour FUN, en appel par valeur avec évaluation de gauche à droite des opérandes.

$$\frac{e_1 \to e_1'}{e_1\ \texttt{+}\ e_2 \to e_1'\ \texttt{+}\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ \texttt{+}\ e_2 \to v_1\ \texttt{+}\ e_2'} \qquad \frac{n_1 + n_2 = n}{n_1\ \texttt{+}\ n_2 \to n}$$

$$\frac{e_1 \to e_1'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2} \qquad \frac{}{\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \to e[x := v]}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \frac{}{(\texttt{fun}\ x\ \texttt{->}\ e)\ v \to e[x := v]}$$

Notez qu'avec ce niveau de détail dans le déroulement des calculs, on peut très facilement imaginer des variantes correspondant à d'autres stratégies d'évaluation.

Exercice : définir une sémantique à petits pas pour FUN en appel par nom.

Séquences de réduction La relation $e \to e'$ décrit les pas de calcul élémentaires. On note donc

— $e \to e'$ lorsque 1 pas de calcul mène de e à e', et
— $e \to^* e'$ lorsque qu'un calcul mène de e à e' en 0, 1 ou plusieurs pas (on parle alors d'une ***séquence*** de calcul ou d'une séquence de réduction).

Une expression ***irréductible*** est une expression e à partir de laquelle on ne peut plus effectuer de pas de calcul, c'est-à-dire pour laquelle il n'existe pas d'expression e' telle que $e \to e'$. Une expression irréductible peut être deux choses :

— une valeur, c'est-à-dire le résultat attendu d'un calcul,
— une expression bloquée, c'est-à-dire une expression décrivant un calcul qui n'est pas terminé, mais pour laquelle aucune règle ne permet de poursuivre.

Exemple de réduction aboutissant à une valeur.

```
let f = fun x -> x + x in f (20 + 1)
→  (fun x -> x + x) (20 + 1)
→  (fun x -> x + x) 21
→  21 + 21
→  42
```

Exemple de réduction aboutissant à un blocage.

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

5.8 Équivalence petits pas et grands pas

Les sémantiques à grands pas et à petits pas donnent des points de vue légèrement différents : la première donne une vision directe du résultat qui peut être attendu d'un programme, alors que la deuxième donne une vision plus précise des différentes opérations effectuées. Ces deux modes de présentation de la sémantique sont cependant *équivalents*, dans le sens qu'ils spécifient les mêmes relations d'évaluation. Autrement dit,

$$e \implies v \qquad \text{si et seulement si} \qquad e \to^* v$$

Démontrons-le pour les deux versions de la sémantique en appel par valeur de FUN. Nous prenons la sémantique à grands pas définie par les règles d'inférence

$$\frac{}{n \implies n} \qquad \frac{}{\texttt{fun } x \texttt{ -> } e \implies \texttt{fun } x \texttt{ -> } e}$$

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 \texttt{ + } e_2 \implies n_1 + n_2} \qquad \frac{e_1 \implies v_1 \qquad e_2[x := v_1] \implies v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \implies v}$$

$$\frac{e_1 \implies \texttt{fun } x \texttt{ -> } e \qquad e_2 \implies v_2 \qquad e[x := v_2] \implies v}{e_1\ e_2 \implies v}$$

et la sémantique à petits pas définie par les règles d'inférence suivantes.

$$\frac{e_1 \to e_1'}{e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2} \qquad \frac{e_2 \to e_2'}{v_1 \texttt{ + } e_2 \to v_1 \texttt{ + } e_2'} \qquad \frac{n_1 + n_2 = n}{n_1 \texttt{ + } n_2 \to n}$$

$$\frac{e_1 \to e_1'}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \to \texttt{let } x \texttt{ = } e_1' \texttt{ in } e_2} \qquad \frac{}{\texttt{let } x \texttt{ = } v \texttt{ in } e \to e[x := v]}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \qquad \frac{}{(\texttt{fun } x \texttt{ -> } e)\ v \to e[x := v]}$$

$e \implies v$ **implique** $e \to^* v$ Démontrons cette implication par récurrence sur la dérivation de $e \implies v$.

- Cas $n \implies n$. On a bien $n \to^* n$ avec une séquence de 0 pas.
- Cas $\texttt{fun } x \texttt{ -> } e \implies \texttt{fun } x \texttt{ -> } e$ immédiat de même.
- Cas $e_1 \texttt{ + } e_2 \implies n$ avec $e_1 \implies n_1$, $e_2 \implies n_2$ et $n = n_1 + n_2$. Les deux prémisses nous donnent les hypothèses de récurrence $e_1 \to^* n_1$ et $e_2 \to^* n_2$. De $e_1 \to^* n_1$ on déduit $e_1 \texttt{ + } e_2 \to^* n_1 \texttt{ + } e_2$ (à strictement parler, il s'agit d'un lemme, à démontrer par récurrence sur la longueur de la séquence de réduction $e_1 \to^* n_1$). De même, de $e_2 \to^* n_2$ on déduit $n_1 \texttt{ + } e_2 \to^* n_1 \texttt{ + } n_2$. En ajoutant une dernière étape avec la règle de réduction de base de l'addition nous obtenons la séquence

  $$\begin{array}{lll} e_1 \texttt{ + } e_2 & \to^* & n_1 \texttt{ + } e_2 \\ & \to^* & n_1 \texttt{ + } n_2 \\ & \to & n \end{array}$$

  qui conclut.
- Cas $\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \implies v$ avec $e_1 \implies v_1$ et $e_2[x := v_1] \implies v$. Les deux prémisses nous donnent les hypothèses de récurrence $e_1 \to^* v_1$ et $e_2[x := v_1] \to^* v$. On en déduit la séquence de réduction

  $$\begin{array}{lll} \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 & \to^* & \texttt{let } x \texttt{ = } v_1 \texttt{ in } e_2 \\ & \to & e_2[x := v_1] \\ & \to^* & v \end{array}$$

- Cas $e_1\ e_2 \implies v$ avec $e_1 \implies \texttt{fun } x \texttt{ -> } e$, $e_2 \implies v_2$ et $e[x := v_2] \implies v$. Les trois prémisses nous donnent les hypothèses de récurrence $e_1 \to^* \texttt{fun } x \texttt{ -> } e$, $e_2 \to^* v_2$ et $e[x := v_2] \to^* v$. On en déduit la séquence de réduction

  $$\begin{array}{lll} e_1\ e_2 & \to^* & (\texttt{fun } x \texttt{ -> } e)\ e_2 \\ & \to^* & (\texttt{fun } x \texttt{ -> } e)\ v_2 \\ & \to & e[x := v_2] \\ & \to^* & v \end{array}$$

$e \to^* v$ **implique** $e \implies v$ Notez que dans cet énoncé, en écrivant $e \to^* v$ on suppose que v est une valeur. Démontrons cette implication par récurrence sur la longueur de la séquence de réduction $e \to^* v$.

— Cas d'une séquence de longueur 0. L'expression e est donc déjà une valeur, c'est-à-dire de la forme n ou de la forme `fun` x `->` e'. On conclut donc immédiatement avec l'un des axiomes

$$\frac{}{n \implies n} \qquad \frac{}{\texttt{fun}\ x\ \texttt{->}\ e' \implies \texttt{fun}\ x\ \texttt{->}\ e'}$$

— Cas d'une séquence $e \to^* v$ de longueur $n+1$, en supposant que pour toute séquence de réduction $e' \to^* v$ de longueur n on a bien $e' \implies v$ (c'est notre hypothèse de récurrence). Notons

$$e \to e' \to \ldots \to v$$

notre séquence de réduction $e \to^* v$ de $n+1$ étapes, avec e' l'expression obtenue au terme de la première étape. Nous avons donc $e' \to^* v$ en n étapes, et donc par hypothèse de récurrence $e' \implies v$.
Pour conclure, on démontre que de manière générale, si $e \to e'$ et $e' \implies v$ alors $e \implies v$.

Lemme : si $e \to e'$ et $e' \implies v$ alors $e \implies v$. Démonstration par récurrence sur la dérivation de $e \to e'$.

— Cas $n_1 + n_2 \to n$ avec $n = n_1 + n_2$. On a dans ce cas également $n \implies v$, qui n'est possible qu'avec $v = n$. On conclut donc avec la dérivation

$$\frac{\dfrac{}{n_1 \implies n_1} \qquad \dfrac{}{n_2 \implies n_2}}{n_1 + n_2 \implies n}$$

— Cas $\texttt{let}\ x\ \texttt{=}\ w\ \texttt{in}\ e \to e[x := w]$, avec w une valeur et où l'hypothèse $e' \implies v$ s'écrit $e' = e[x := w] \implies v$. On conclut donc avec la dérivation

$$\frac{w \implies w \qquad e[x := w] \implies v}{\texttt{let}\ x\ \texttt{=}\ w\ \texttt{in}\ e \implies v}$$

Notez que nous n'avons pas utilisé d'hypothèse de récurrence ici !

— Cas $e_1 + e_2 \to e_1' + e_2$ avec $e_1 + e_1'$ et où l'hypothèse $e' \implies v$ s'écrit $e_1' + e_2 \implies v$. La prémisse $e_1 + e_1'$ nous donne comme hypothèse de récurrence « pour toute valeur v_1 telle que $e_1' \implies v_1$ on a $e_1 \implies v_1$ ».
Par inversion du jugement $e_1' + e_2 \implies v$ on déduit que v s'exprime $n_1 + n_2$ avec n_1 et n_2 tels que $e_1' \implies n_1$ et $e_2 \implies n_2$. Avec l'hypothèse de récurrence on déduit donc $e_1 \implies n_1$. Grâce à ce jugement, on peut construire la dérivation suivante.

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 + e_2 \implies n}$$

— Autres cas similaires.

Ainsi, nos deux présentations de la sémantique associent les mêmes expressions aux mêmes valeurs. La sémantique à petits pas en revanche nous dit des choses plus précises des évaluations qui n'aboutissent pas. Elle va permettre un énoncé satisfaisant des propriétés de sûreté, c'est-à-dire d'absence de problèmes d'évaluation, des programmes bien typés.

5.9 Sûreté du typage

Le slogan associé au typage était *well-typed programs do not go wrong*. Avec une sémantique à petits pas, ce slogan peut se traduire plus formellement par le fait suivant : l'évaluation d'un programme bien typé ne bloque jamais.

On traduit ce résultat par la conjonction des deux lemmes suivants.

— Lemme de ***progression*** : une expression bien typée n'est pas bloquée. Autrement dit, si une expression e bien typée n'est pas déjà une valeur, alors on peut encore effectuer au moins un pas de calcul à partir de e.

$$\textit{Si} \quad \Gamma \vdash e : \tau \quad \textit{alors } e \textit{ est une valeur ou il existe } e' \textit{ telle que } \quad e \to e'.$$

— Lemme de ***préservation du typage*** : la réduction préserve les types. Si une expression e est cohérente, alors toute expression e' obtenue en calculant à partir de e est encore cohérente et de même type.

$$\textit{Si} \quad \Gamma \vdash e : \tau \quad \textit{et} \quad e \to e' \quad \textit{alors} \quad \Gamma \vdash e' : \tau.$$

Historiquement, le lemme de préservation est appelé en anglais ***subject reduction*** (explication : e est le « sujet » du prédicat $\Gamma \vdash e : \tau$).

Ces deux lemmes rassemblés, et appliqués de manière itérée, promettent le comportement suivant de l'évaluation d'une expression e_1 cohérente et de type τ : si e_1 n'est pas déjà une valeur, alors elle se réduit en e_2, qui est encore cohérente et de type τ et qui donc, si elle n'est pas déjà une valeur, se réduit en e_3 cohérente et de type τ, etc.

$$(e_1 : \tau) \;\to\; (e_2 : \tau) \;\to\; (e_3 : \tau) \;\to\; \ldots$$

À l'extrémité droite de cette séquence, deux scénarios sont possibles : soit on aboutit à une valeur v (accessoirement : cohérente et de type τ), soit la réduction se poursuit indéfiniment. Il est en revanche impossible d'aboutir à un blocage.

Progression Si $\Gamma \vdash e : \tau$, alors e est une valeur, ou il existe e' telle que $e \to e'$.

Reprenons les types simples du langage FUN définis par les règles suivantes

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

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

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

et démontrons le lemme par récurrence sur la dérivation de $\Gamma \vdash e : \tau$.

— Cas $\Gamma \vdash n : \texttt{int}$. Alors n est une valeur.

— Cas $\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2$. Alors de même $\texttt{fun } x \texttt{ -> } e$ est une valeur.

— Cas $\Gamma \vdash e_1\, e_2 : \tau_1$, avec $\Gamma \vdash e_1 : \tau_2 \to \tau_1$ et $\Gamma \vdash e_2 : \tau_2$. Les hypothèses de récurrence sur les deux prémisses nous donnent les deux disjonctions suivantes :

1. e_1 est une valeur ou $e_1 \to e_1'$,
2. e_2 est une valeur ou $e_2 \to e_2'$.

On raisonne par cas sur ces disjonctions.

— Si $e_1 \to e_1'$, alors $e_1\, e_2 \to e_1'\, e_2$: conclusion.

— Sinon, e_1 est une valeur v_1.

— Si $e_2 \to e_2'$, alors $v_1\, e_2 \to v_1\, e_2'$: conclusion.

— Sinon, e_2 est une valeur v_2. Analysons la forme de la valeur v_1. Les deux formes possibles pour une valeur sont : n ou $\texttt{fun } x \texttt{ -> } e$. Or nous avons l'hypothèse de typage $\Gamma \vdash v_1 : \tau_2 \to \tau_1$, donc v_1 ne peut avoir que la forme $\texttt{fun } x \texttt{ -> } e$ (lemme de ***classification*** détaillé plus bas). Nous avons donc

$$e_1\, e_2 = (\texttt{fun } x \texttt{ -> } e)\, v_2 \to e[x := v_2]$$

ce qui conclut.

— Autres cas similaires.

Lemme de classification des valeurs typées. Soit v une valeur telle que $\Gamma \vdash v : \tau$. Alors :

- si $\tau = \texttt{int}$, alors v a la forme n,
- si $\tau = \tau_1 \to \tau_2$, alors v a la forme $\texttt{fun } x \texttt{ -> } e$.

Preuve par cas sur la dernière règle de la dérivation du jugement $\Gamma \vdash v : \tau$.

Préservation du typage Si $\Gamma \vdash e : \tau$ et $e \to e'$ alors $\Gamma \vdash e' : \tau$. Preuve par récurrence sur la dérivation de $e \to e'$.

- Cas $n_1 \texttt{ + } n_2 \to n$ avec $n = n_1 + n_2$. De l'hypothèse $\Gamma \vdash n_1 \texttt{ + } n_2 : \texttt{int}$ on a nécessairement $\tau = \texttt{int}$ (lemme d'***inversion*** détaillé plus bas), et en outre $\Gamma \vdash n : \texttt{int}$.
- Cas $e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2$ avec $e_1 \to e_1'$. La prémisse nous donne l'hypothèse de récurrence « si $\Gamma \vdash e_1 : \tau'$, alors $\Gamma \vdash e_1' : \tau'$ ».
  De l'hypothèse $\Gamma \vdash e_1 \texttt{ + } e_2 : \tau$ on a nécessairement $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$ (lemme d'inversion). Donc par hypothèse de récurrence $\Gamma \vdash e_1' : \texttt{int}$, dont on déduit encore la dérivation

$$\frac{\Gamma \vdash e_1' : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

- Cas $(\texttt{fun } x \texttt{ in } e)\ v \to e[x := n]$. *Note : pas de prémisse à cette règle de réduction, et donc pas d'hypothèse de récurrence non plus.*
  De l'hypothèse $\Gamma \vdash (\texttt{fun } x \texttt{ in } e)\ v : \tau$ il existe τ' tel que $\Gamma \vdash \texttt{fun } x \texttt{ in } e : \tau' \to \tau$ et $\Gamma \vdash v : \tau'$ (lemme d'inversion) et de $\Gamma \vdash \texttt{fun } x \texttt{ in } e : \tau' \to \tau$ on a encore $\Gamma, x : \tau' \vdash e : \tau$ (lemme d'inversion toujours).
  On a donc d'une part $\Gamma, x : \tau' \vdash e : \tau$ et d'autre part $\Gamma \vdash v : \tau'$, ce dont on peut déduire que $\Gamma \vdash e[x := v] : \tau$, par le lemme de ***substitution*** détaillé ci-dessous.
- Autres cas similaires.

Lemme d'inversion.

- Si $\Gamma \vdash e_1 \texttt{ + } e_2 : \tau$ alors $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ et $\Gamma \vdash e_2 : \texttt{int}$.
- Si $\Gamma \vdash e_1\ e_2 : \tau$ alors il existe τ' tel que $\Gamma \vdash e_1 : \tau' \to \tau$ et $\Gamma \vdash e_2 : \tau'$.
- Si $\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau$ alors il existe τ_1 et τ_2 tels que $\tau = \tau_1 \to \tau_2$ et $\Gamma, x : \tau_1 \vdash e : \tau_2$.

Preuve par cas sur la dernière règle de la dérivation de typage.

Lemme de substitution : remplacer une variable typée par une expression du même type préserve le typage.

$$\text{Si} \quad \Gamma, x : \tau' \vdash e : \tau \quad \text{et} \quad \Gamma \vdash e' : \tau' \quad \text{alors} \quad \Gamma \vdash e[x := e'] : \tau.$$

Preuve par récurrence sur la dérivation de typage $\Gamma, x : \tau' \vdash e : \tau$.

Théorème de sûreté du typage Des lemmes de progression et de préservation du typage, on déduit l'énoncé suivant.

$$\textit{Si} \quad \Gamma \vdash e : \tau \quad \textit{et} \quad e \to^* e' \quad \textit{avec } e' \textit{ irréductible, alors } e' \textit{ est une valeur.}$$

La preuve est par récurrence sur la longueur de la séquence de réduction $e \to^* e'$.

Bilan : la propriété de sûreté du typage est un lien entre une propriété statique d'un programme (sa cohérence du point de vue des types) et une propriété dynamique de ce programme (l'absence de blocage à l'exécution). Il reste possible que le programme boucle, ou diverge. De manière générale, un langage de programmation doté d'une discipline de types stricte va permettre une détection précoce de nombreuses erreurs (à la compilation), et donc éviter que certains bugs n'apparaissent que plus tard (à l'exécution).