Chapitre 6

L'inférence de types

Dans beaucoup de langages de programmation, comme Java, C, ..., les programmeurs doivent indiquer eux-mêmes le type des variables du programme, et écrire par exemple `fun x:nat -> x + 1`. Pourtant, sachant que `+` ne peut prendre que des arguments entiers, il n'est pas difficile de montrer que, dans le terme `fun x -> x + 1`, la variable `x` ne peut être que de type `nat`. On peut donc dispenser les programmeurs de l'écriture des types et laisser l'ordinateur inférer ces types. C'est l'objet des algorithmes d'*inférence de types*.

6.1 L'inférence de types monomorphes

6.1.1 L'attribution d'un type aux termes sans types

On revient donc à la syntaxe de PCF dans laquelle les variables ne sont pas explicitement typées. On n'écrit donc plus `fun x:nat -> x + 1`, mais, comme au chapitre 2, `fun x -> x + 1`.

Le langage des termes et le langage des types peuvent maintenant se définir indépendamment l'un de l'autre. Le langage PCF se définit comme au chapitre 2 et le langage des types est formé

- d'une constante `nat`,
- et d'un symbole `->` à deux arguments ne liant pas de variable dans ses arguments.

```
A = X
  | nat
  | A -> A
```

Comme précédemment, nous définissons inductivement la relation `e ⊢ t : A` « le terme `t` a le type `A` dans l'environnement `e` ».

$$\frac{}{\texttt{e} \vdash \texttt{x} : \texttt{A}} \text{ si e contient } \texttt{x} : \texttt{A}$$

$$\frac{\texttt{e} \vdash \texttt{u} : \texttt{A} \quad \texttt{e} \vdash \texttt{t} : \texttt{A -> B}}{\texttt{e} \vdash \texttt{t u} : \texttt{B}}$$

$$\frac{\texttt{(e, x : A)} \vdash \texttt{t} : \texttt{B}}{\texttt{e} \vdash \texttt{fun x -> t} : \texttt{A -> B}}$$

$$\frac{}{\texttt{e} \vdash \texttt{n} : \texttt{nat}}$$

$$\frac{\texttt{e} \vdash \texttt{u} : \texttt{nat} \quad \texttt{e} \vdash \texttt{t} : \texttt{nat}}{\texttt{e} \vdash \texttt{t} \otimes \texttt{u} : \texttt{nat}}$$

$$\frac{\texttt{e} \vdash \texttt{t} : \texttt{nat} \quad \texttt{e} \vdash \texttt{u} : \texttt{A} \quad \texttt{e} \vdash \texttt{v} : \texttt{A}}{\texttt{e} \vdash \texttt{ifz t then u else v} : \texttt{A}}$$

$$\frac{\texttt{(e, x : A)} \vdash \texttt{t} : \texttt{A}}{\texttt{e} \vdash \texttt{fix x t} : \texttt{A}}$$

$$\frac{\texttt{e} \vdash \texttt{t} : \texttt{A} \quad \texttt{(e, x : A)} \vdash \texttt{u} : \texttt{B}}{\texttt{e} \vdash \texttt{let x = t in u} : \texttt{B}}$$

Certains termes, par exemple le terme `fun x -> x`, ont maintenant plusieurs types, on peut par exemple dériver l'énoncé $\vdash$ `fun x -> x : nat -> nat` et l'énoncé $\vdash$ `fun x -> x : (nat -> nat) -> (nat -> nat)`. Également, un terme clos peut avoir un type qui contient des variables libres. C'est par exemple le cas du terme `fun x -> x` qui a le type `X -> X` dans l'environnement vide.

On peut démontrer que quand un terme clos `t` a un type `A` qui contient des variables dans l'environnement vide, alors le terme `t` a également le type θ`A` pour toute substitution θ. Par exemple, en substituant la variable `X` par le type `nat -> nat` dans le type `X -> X`, on obtient le type `(nat -> nat) -> (nat -> nat)` qui est bien l'un des types du terme `fun x -> x`.

6.1.2 L'algorithme de Hindley

Venons-en maintenant à l'algorithme d'inférence de types proprement dit. Le premier algorithme que nous allons voir procède en deux étapes. La première ressemble beaucoup à l'algorithme de vérification de types : c'est une étape de parcours récursif du terme où on vérifie à chaque étape que les contraintes de types sont bien vérifiées et où on calcule le type du terme. Il y a cependant deux différences avec la vérification de types : quand on cherche à typer un terme de la forme `fun x -> t` dans un environnement `e`, comme on ne connaît pas le type de la variable `x` on crée une variable de type `X`, on ajoute à l'environnement `e` la déclaration `x : X` et on type le terme `t` dans cet environnement étendu. Seconde différence : quand on type une application `t u` après avoir calculé récursivement les types `A` de `u` et `B` de `t`, on ne peut pas se contenter de vérifier que le type `B` a la forme `A -> C`, mais comme ces deux types peuvent contenir des variables, on doit poser une équation entre types `B = A -> X`. La seconde étape de l'algorithme d'inférence de types consiste à résoudre ces équations.

Prenons un exemple : pour typer le terme `fun f -> 2 + (f 1)` on doit typer le terme `2 + (f 1)` dans l'environnement `f : X`. Pour cela, on doit typer

le terme 2 dont le type est `nat` et le terme `f 1`. Le type du terme `1` est `nat` et celui du terme `f` est `X`. On pose donc l'équation `X = nat -> Y` et le type de `f 1` est `Y`. Une fois le terme `2` et `f 1` typés, on pose les équations `nat = nat` et `Y = nat` et le type du terme `2 + (f 1)` est `nat`. Finalement, le type du terme `fun f -> 2 + (f 1)` est `X -> nat` et les équations à résoudre sont

```
X = nat -> Y
  nat = nat
    Y = nat
```

Ce système d'équations a une solution unique `X = nat -> nat`, `Y = nat`, et l'unique type du terme `fun f -> 2 + (f 1)` est donc `(nat -> nat) -> nat`.

Comme l'algorithme de vérification de types, la première étape de l'algorithme peut se décrire par un ensemble de règles de sémantique opérationnelle à grands pas, où le résultat de l'interprétation du terme n'est ni sa valeur, ni son type, mais un couple formé d'un type et d'un ensemble d'équations entre types. On écrit `e ⊢ t ⇝ A, E` la relation liant l'environnement `e`, le terme `t`, le type `A` et l'ensemble d'équations `E`.

$$\frac{}{\texttt{e} \vdash \texttt{x} \rightsquigarrow \texttt{A}, \varnothing} \text{ si e contient } \texttt{x} : \texttt{A}$$

$$\frac{\texttt{e} \vdash \texttt{u} \rightsquigarrow \texttt{A}, \texttt{E} \qquad \texttt{e} \vdash \texttt{t} \rightsquigarrow \texttt{B}, \texttt{F}}{\texttt{e} \vdash \texttt{t u} \rightsquigarrow \texttt{X}, \texttt{E} \cup \texttt{F} \cup \{\texttt{B = A -> X}\}}$$

$$\frac{(\texttt{e}, \texttt{x} : \texttt{X}) \vdash \texttt{t} \rightsquigarrow \texttt{A}, \texttt{E}}{\texttt{e} \vdash \texttt{fun x -> t} \rightsquigarrow \texttt{(X -> A)}, \texttt{E}}$$

$$\frac{}{\texttt{e} \vdash \texttt{n} \rightsquigarrow \texttt{nat}, \varnothing}$$

$$\frac{\texttt{e} \vdash \texttt{u} \rightsquigarrow \texttt{A}, \texttt{E} \qquad \texttt{e} \vdash \texttt{t} \rightsquigarrow \texttt{B}, \texttt{F}}{\texttt{e} \vdash \texttt{t} \otimes \texttt{u} \rightsquigarrow \texttt{nat}, \texttt{E} \cup \texttt{F} \cup \{\texttt{A = nat, B = nat}\}}$$

$$\frac{\texttt{e} \vdash \texttt{t} \rightsquigarrow \texttt{A}, \texttt{E} \qquad \texttt{e} \vdash \texttt{u} \rightsquigarrow \texttt{B}, \texttt{F} \qquad \texttt{e} \vdash \texttt{v} \rightsquigarrow \texttt{C}, \texttt{G}}{\texttt{e} \vdash \texttt{ifz t then u else v} \rightsquigarrow \texttt{B}, \texttt{E} \cup \texttt{F} \cup \texttt{G} \cup \{\texttt{A = nat, B = C}\}}$$

$$\frac{(\texttt{e}, \texttt{x} : \texttt{X}) \vdash \texttt{t} \rightsquigarrow \texttt{A}, \texttt{E}}{\texttt{e} \vdash \texttt{fix x t} \rightsquigarrow \texttt{A}, \texttt{E} \cup \{\texttt{X = A}\}}$$

$$\frac{\texttt{e} \vdash \texttt{t} \rightsquigarrow \texttt{A}, \texttt{E} \qquad (\texttt{e}, \texttt{x} : \texttt{A}) \vdash \texttt{u} \rightsquigarrow \texttt{B}, \texttt{F}}{\texttt{e} \vdash \texttt{let x = t in u} \rightsquigarrow \texttt{B}, \texttt{E} \cup \texttt{F}}$$

Dans la règle de l'application, la variable `X` est une variable quelconque qui n'apparaît pas dans `e`, `A`, `B`, `E` et `F`. Dans la règle du `fun` et du `fix`, c'est une variable quelconque qui n'apparaît pas dans `e`.

Soit `t` un terme clos et `A` et `E` le type et l'ensemble d'équations calculés par cet algorithme, c'est-à-dire tels que $\vdash$ `t` $\rightsquigarrow$ `A, E`. Une substitution σ = $\texttt{B}_1/\texttt{X}_1$, ..., $\texttt{B}_n/\texttt{X}_n$ est une *solution* de l'ensemble `E` si pour toute équation `C = D` de `E`, les types σ`C` et σ`D` sont identiques. On peut démontrer que pour toute substitution σ solution de l'ensemble `E`, le type σ`A` est un type de `t` dans l'environnement

vide. Plus généralement, si `e` $\vdash$ `t` $\leadsto$ `A, E`, alors pour toute substitution σ solution de l'ensemble `E`, le type σ`A` est un type de `t` dans l'environnement σ`e`. Réciproquement, pour tout type `A'` de `t` dans l'environnement vide, il existe une substitution σ telle que `A' =` σ`A` et σ est une solution de l'ensemble d'équations `E`.

La seconde étape de l'algorithme consiste à résoudre les équations sur les types. Le langage des types est un langage sans variables liées formé d'une constante `nat` et d'un symbole `->` à deux arguments. Pour résoudre les équations sur les types, on utilise l'algorithme d'unification de Robinson qui permet de résoudre des équations dans n'importe quel langage sans variables liées. Cet algorithme rappelle, par certains aspects, l'algorithme du pivot de Gauss. Il procède par transformations progressives du système d'équations

- si une équation du système a la forme `A -> B = C -> D`, la remplacer par les équations `A = C` et `B = D`,
- si une équation du système a la forme `nat = nat`, la supprimer,
- si une équation du système a la forme `nat = A -> B` ou `A -> B = nat`, échouer,
- si une équation du système a la forme `X = X`, la supprimer,
- si une équation du système a la forme `X = A` ou `A = X`, si `X` apparaît dans `A` et si `A` est distinct de `X`, échouer,
- si une équation du système a la forme `X = A` ou `A = X`, si `X` n'apparaît pas dans `A` et si `X` apparaît dans les autres équations du système, substituer `X` par `A` dans toutes les autres équations du système.

On peut démontrer, bien que ce ne soit pas absolument trivial, que cet algorithme termine. S'il échoue, alors le système dont on était parti n'avait pas de solution. S'il termine sans échouer alors le système final a la forme $\mathtt{X_1} = \mathtt{A_1}, \ldots, \mathtt{X_n} = \mathtt{A_n}$, les $\mathtt{X_i}$ sont des variables distinctes et n'apparaissent pas dans les $\mathtt{A_i}$. Dans ce cas, la substitution $\sigma = \mathtt{A_1}/\mathtt{X_1}, \ldots, \mathtt{A_n}/\mathtt{X_n}$ est une solution du système initial. On peut également démontrer que cette substitution est une solution *principale* de ce système, c'est-à-dire que pour toute substitution θ solution du système initial, il existe une substitution η telle que $\theta = \eta \circ \sigma$. On écrit cette substitution $\sigma = mgu(E)$ — *most general unifier* : solution principale.

Soit `t` un terme clos, soient `A` et `E` tels que $\vdash$ `t` $\leadsto$ `A, E`, soit σ une solution principale de `E` alors le terme `t` est de type σ`A` dans l'environnement vide. De plus, σA est un type *principal* de `t`, c'est-à-dire que pour tout type B de t, il existe une substitution η telle que `B =` $\eta\sigma$`A`.

6.1.3 L'algorithme de Hindley avec résolution immédiate

Une variante de l'algorithme de Hindley consiste à ne pas attendre la fin de la première étape pour s'attaquer à la résolution des équations, mais à les résoudre au fur et à mesure qu'on les construit. Dans ce cas, au lieu de retourner un type et un ensemble d'équations, on retourne un type `A` et une substitution ρ, solution principale des équations. On peut également appliquer la substitution ρ au type `A` au fur et à mesure de sa construction.

La propriété de l'algorithme est donc que si $\mathtt{e} \vdash \mathtt{t} \leadsto \mathtt{A},\ \rho$, alors A est un type principal de t dans l'environnement ρe. L'algorithme se présente alors ainsi

$$\frac{}{\mathtt{e} \vdash \mathtt{x} \leadsto \mathtt{A},\ \varnothing} \text{ si e contient } \mathtt{x} : \mathtt{A}$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \leadsto \mathtt{A},\ \rho \qquad \rho\mathtt{e} \vdash \mathtt{t} \leadsto \mathtt{B},\ \rho'}{\mathtt{e} \vdash \mathtt{t\ u} \leadsto \sigma\mathtt{X},\ \sigma \circ \rho' \circ \rho}$$

si σ = mgu(B = ρ'A -> X)

$$\frac{(\mathtt{e},\ \mathtt{x} : \mathtt{X}) \vdash \mathtt{t} \leadsto \mathtt{A},\ \rho}{\mathtt{e} \vdash \mathtt{fun\ x\ ->\ t} \leadsto (\rho\mathtt{X}\ \mathtt{->}\ \mathtt{A}),\ \rho}$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n} \leadsto \mathtt{nat},\ \varnothing}$$

$$\frac{\mathtt{e} \vdash \mathtt{u} \leadsto \mathtt{A},\ \rho \qquad \sigma\rho\mathtt{e} \vdash \mathtt{t} \leadsto \mathtt{B},\ \rho'}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u} \leadsto \mathtt{nat},\ \sigma' \circ \rho' \circ \sigma \circ \rho}$$

si σ = mgu(A = nat) et σ' = mgu(B = nat)

$$\frac{\mathtt{e} \vdash \mathtt{t} \leadsto \mathtt{A},\ \rho \qquad \sigma\rho\mathtt{e} \vdash \mathtt{u} \leadsto \mathtt{B},\ \rho' \qquad \rho'\sigma\rho\mathtt{e} \vdash \mathtt{v} \leadsto \mathtt{C},\ \rho''}{\mathtt{e} \vdash \mathtt{ifz\ t\ then\ u\ else\ v} \leadsto \sigma'\ \mathtt{C},\ \sigma' \circ \rho'' \circ \rho' \circ \sigma \circ \rho}$$

si σ = mgu(A = nat) et σ' = mgu(ρ'' B = C)

$$\frac{(\mathtt{e},\ \mathtt{x} : \mathtt{X}) \vdash \mathtt{t} \leadsto \mathtt{A},\ \rho}{\mathtt{e} \vdash \mathtt{fix\ x\ t} \leadsto \sigma\mathtt{A},\ \sigma \circ \rho} \text{ si } \sigma = \mathtt{mgu(A} = \rho\mathtt{X)}$$

$$\frac{\mathtt{e} \vdash \mathtt{t} \leadsto \mathtt{A},\ \rho \qquad (\rho\mathtt{e},\ \mathtt{x} : \mathtt{A}) \vdash \mathtt{u} \leadsto \mathtt{B},\ \rho'}{\mathtt{e} \vdash \mathtt{let\ x = t\ in\ u} \leadsto \mathtt{B},\ \rho' \circ \rho}$$

Ici encore, dans la règle de l'application, la variable X est une variable quelconque qui n'apparaît pas dans e, A, B, ρ et ρ', et dans les règles du fun et du fix, c'est une variable qui n'apparaît pas dans e.

Exercice 6.1 *Donner un type principal du terme* `fun x -> fun y -> (x (y + 1)) + 2` *? Quels sont tous ses types ? Donner un type principal du terme* `fun x -> x` *? Quels sont tous ses types ?*

Exercice 6.2 *(Unicité du type principal) Une substitution σ est appelée un* renommage *si c'est une injection et si elle associe une variable à chaque variable. Par exemple, la substitution* y/x, z/y *est un renommage. Soit* A *un type et σ et σ' deux substitutions, montrer que si $\sigma'\sigma$A = A alors $\sigma_{|\mathtt{VL(A)}}$ est un renommage.*

En déduire que si A *et* A' *sont deux types principaux d'un terme* t *alors il existe un renommage θ, dont le domaine est* VL(A)*, tel que* A' = θA*.*

Exercice 6.3 *Dans le cas général d'un langage sans variables liées on remplace les trois premières règles de l'algorithme de Robinson par les deux règles suivantes*

- *si une équation a la forme* $\texttt{f(u}_1\texttt{, ..., u}_n\texttt{) = f(v}_1\texttt{, ..., v}_n\texttt{)}$*, la remplacer par les équations* $\texttt{u}_1\texttt{ = v}_1$*, ...,* $\texttt{u}_n\texttt{ = v}_n$*,*
- *si une équation a la forme* $\texttt{f(u}_1\texttt{, ..., u}_n\texttt{) = g(v}_1\texttt{, ..., v}_p\texttt{)}$ *où* `f` *et* `g` *sont des symboles distincts, échouer.*

Dans un langage formé d'un symbole `+` *à deux arguments et des constantes entières, l'équation* `(2 + (3 + X)) = (X + (Y + 2))` *a-t-elle une solution ? Et l'équation* `X + 2 = 4` *?*

Quelle différence y a-t-il entre les équations dans ce langage et les équations sur les entiers que l'on étudie au collège ?

Comment peut-on définir, en utilisant la sémantique opérationnelle à petits pas de PCF, une autre notion de solution pour ces équations qui corresponde à la notion étudiée au collège ? L'équation `X + 2 = 4` *a-t-elle une solution dans ce cas ?*

6.2 Le polymorphisme

Nous avons vu qu'un type principal du terme `id = fun x -> x` était `X -> X`. De ce fait, le terme `id` a le type `A -> A` pour tout type `A`. On peut donc lui attribuer un nouveau type `∀X (X -> X)` et ajouter une règle selon laquelle si le terme `t` a le type `∀X A` alors il a le type `(B/X)A` pour tout type `B`. Un langage de types qui contient ainsi un quantificateur universel est appelé *polymorphe*.

Dans le système de la section précédente, le terme `let id = fun x -> x in id id` n'était pas typable. En effet, la règle de typage du `let` nous demande de typer les deux termes `fun x -> x` et `id id` et ce second terme n'est pas typable, car on ne peut pas attribuer le même type aux deux occurrences de la variable `id`. De ce fait, le terme complet `let id = fun x -> x in id id` n'est pas typable. Cela peut être considéré comme un défaut du système de type, car le terme `(fun x -> x) (fun x -> x)`, obtenu en remplaçant `id` par sa définition, est, en revanche, typable. Il suffit, en effet, pour typer ce terme, d'attribuer le type `nat -> nat` à la première variable liée `x` et le type `nat` à la seconde.

Donner le type `∀X (X -> X)` au symbole `id` dans le terme `let id = fun x -> x in id id` permet de donner un type différent à chacune des occurrences de `id` dans le terme `id id` et donc de typer ce terme.

On peut penser qu'il est anecdotique de pouvoir typer ou non le terme `let id = fun x -> x in id id` et qu'ajouter des quantificateurs au langage des types est un prix élevé pour obtenir une extension quelque peu marginale. Il n'en est rien. En effet, quand on étend PCF avec des listes — voir l'exercice 3.14 —, on peut ainsi développer un algorithme de tri unique, qui s'applique à toutes les listes quel que soit le type de leurs éléments : `let sort = t in u`. Le polymorphisme permet donc une beaucoup plus grande réutilisabilité du code, et donc une programmation beaucoup plus concise.

Nous allons donc donner un type quantifié aux variables liées par un `let`, mais nous laisserons un type ordinaire aux variables liées par un `fun` ou un `fix`.

6.2.1 Le langage PCF avec des types polymorphes

Nous devons distinguer les types sans quantificateurs — que nous continuerons d'appeler *types* — et les types quantifiés que nous appellerons *schémas de types*. Un schéma a la forme $\forall \mathtt{X}_1 \ \ldots \ \forall \mathtt{X}_n \ \mathtt{A}$ où `A` est un type. Nous définissons donc un langage à deux sortes : l'une pour les types et une autre pour les schémas. Comme l'ensemble des termes d'une sorte et d'une autre sont toujours disjoints dans un langage à plusieurs sortes, l'ensemble des types ne peut pas être un sous-ensemble de celui des schémas, et nous avons besoin d'un symbole `[ ]` pour injecter un type dans la sorte des schémas. Ainsi, si `A` est un type, `[A]` est le schéma formé du type `A` dans lequel aucune variable n'est quantifiée. Le langage des types et des schémas est donc formé

- d'une constante de type `nat`,
- d'un symbole de type `->` à deux arguments, dont les deux arguments sont des types, ne liant pas de variable dans ses arguments,
- d'un symbole de schéma `[ ]` à un argument, dont l'argument est un type, ne liant pas de variable dans son argument,
- d'un symbole de schéma $\forall$ à un argument, dont l'argument est un schéma, liant une variable de type dans son argument.

```
A = X
  | nat
  | A -> A
S = Y
  | [A]
  | ∀X S
```

Ce langage contient des variables de toutes les sortes, en particulier des variables de schéma. Cependant, ces variables ne seront pas utilisées.

Un environnement est maintenant une liste qui associe un schéma à chaque variable, et on définit inductivement la relation « le terme `t` a le schéma `S` dans l'environnement `e` »

$$\frac{}{\mathtt{e} \vdash \mathtt{x} : \mathtt{S}} \text{ si } \mathtt{e} \text{ contient } \mathtt{x} : \mathtt{S}$$

$$\frac{\mathtt{e} \vdash \mathtt{u} : \mathtt{[A]} \quad \mathtt{e} \vdash \mathtt{t} : \mathtt{[A \to B]}}{\mathtt{e} \vdash \mathtt{t\ u} : \mathtt{[B]}}$$

$$\frac{\mathtt{(e,\ x} : \mathtt{[A])} \vdash \mathtt{t} : \mathtt{[B]}}{\mathtt{e} \vdash \mathtt{fun\ x \to t} : \mathtt{[A \to B]}}$$

$$\frac{}{\mathtt{e} \vdash \mathtt{n} : \mathtt{[nat]}}$$

$$\frac{\mathtt{e} \vdash \mathtt{u} : \mathtt{[nat]} \quad \mathtt{e} \vdash \mathtt{t} : \mathtt{[nat]}}{\mathtt{e} \vdash \mathtt{t} \otimes \mathtt{u} : \mathtt{[nat]}}$$

$$\frac{\mathtt{e} \vdash \mathtt{t} : \mathtt{[nat]} \quad \mathtt{e} \vdash \mathtt{u} : \mathtt{[A]} \quad \mathtt{e} \vdash \mathtt{v} : \mathtt{[A]}}{\mathtt{e} \vdash \mathtt{ifz\ t\ then\ u\ else\ v} : \mathtt{[A]}}$$

$$\frac{\texttt{(e, x : [A])} \vdash \texttt{t : [A]}}{\texttt{e} \vdash \texttt{fix x t : [A]}}$$

$$\frac{\texttt{e} \vdash \texttt{t : S} \quad \texttt{(e, x : S)} \vdash \texttt{u : [B]}}{\texttt{e} \vdash \texttt{let x = t in u : [B]}}$$

$$\frac{\texttt{e} \vdash \texttt{t : S}}{\texttt{e} \vdash \texttt{t :}\forall\texttt{X S}} \text{ si X n'apparaît pas libre dans e}$$

$$\frac{\texttt{e} \vdash \texttt{t : } \forall\texttt{X S}}{\texttt{e} \vdash \texttt{t : (A/X)S}}$$

Cette définition inductive attribue un schéma à chaque terme, en particulier aux variables. De ce fait, dans l'environnement, les variables sont associées à des schémas. Cependant, quand on cherche à typer un terme de la forme `fun x -> t` ou `fix x t`, on type le terme `t` dans un environnement étendu dans lequel la variable `x` est associée à un schéma sans quantificateurs de la forme `[A]`. C'est seulement quand on type un terme de la forme `let x = t in u` que l'on peut donner un schéma quelconque au terme `t` et attribuer ensuite ce même schéma à la variable `x`.

Pour pouvoir introduire des quantificateurs dans le schéma associé au terme `t`, on utilise l'avant dernière règle qui permet de quantifier une variable du schéma `S` à condition que cette variable n'apparaisse pas dans `e`. Ainsi, dans l'environnement vide, après avoir donné le schéma `[X -> X]` au terme `fun x -> x` on peut lui donner le schéma $\forall$`X [X -> X]`. En revanche, dans l'environnement `x : [X]`, après avoir donné le schéma `[X]` à la variable `x`, on ne peut pas lui donner le schéma $\forall$`X [X]`.

Enfin, quand on a attribué un schéma quantifié à une variable, ou à un terme quelconque, on peut supprimer un quantificateur et substituer la variable ainsi libérée en utilisant la dernière règle. Par exemple dans l'environnement `x` : $\forall$`X [X -> X]` on peut donner le schéma `[nat -> nat]` à la variable `x`.

6.2.2 L'algorithme de Damas et Milner

Venons-en maintenant à l'algorithme d'inférence de types proprement dit. Comme dans la seconde variante de l'algorithme de Hindley, nous allons résoudre les équations au fur et à mesure. L'algorithme s'applique donc à un terme `t` et un environnement `e` et il retourne un type `A` et une substitution ρ, tels que le type `t` ait le schéma `[A]` dans l'environnement ρ`e`. La seule différence avec la seconde variante de l'algorithme de Hindley réside dans la première et dans la dernière règle

$$\frac{}{\texttt{e} \vdash \texttt{x} \rightsquigarrow \texttt{(}\texttt{Y}_1\texttt{/X}_1 \ \ldots \ \texttt{Y}_n\texttt{/X}_n\texttt{)A,}\varnothing}$$

si `e` contient `x` : $\forall$`X`$_1$ ... $\forall$`X`$_n$ `[A]` et `Y`$_1$, ..., `Y`$_n$ sont de nouvelles variables

$$\frac{\texttt{e} \vdash \texttt{t} \rightsquigarrow \texttt{A,}\rho \quad \texttt{(}\rho\texttt{e, x : Gen(A,}\rho\texttt{e)} \vdash \texttt{u} \rightsquigarrow \texttt{B,}\rho'}{\texttt{e} \vdash \texttt{let x = t in u} \rightsquigarrow \texttt{B, } \rho' \circ \rho}$$

où `Gen(A,e)` est le schéma obtenu en quantifiant dans `[A]` toutes les variables de type qui sont libres dans `[A]` mais pas dans `e`.

On peut alors démontrer que si `t` est un terme clos, alors le type `A` calculé par l'algorithme de Damas et Milner est un type principal de `t` en ce sens que si $\vdash$ `t :[B]` alors `B` est une instance de `A`.

Exercice 6.4 *On étend PCF en ajoutant un symbole de type* `list` *à un argument, dont l'argument est un type. On note alors* `nat list` *le type des listes d'entiers,* `(nat -> nat) list` *le type des listes de fonctions des entiers dans les entiers et* `(nat list) list` *le type des listes de listes d'entiers.*

On ajoute les constructions suivantes : une constante `nil` *de type* `(A list)` *pour tout type* `A` *et qui est la liste vide,* `cons a l` *qui est de type* `(A list)` *pour tout type* `A` *si* `a` *est de type* `A` *et* `l` *de type* `A list`*, et qui est la liste dont le premier élément est* `a` *et le reste* `l`*,* `ifnil t then u else v` *qui est de type* `A` *si* `t` *est de type* `B list` *et* `u` *et* `v` *sont de type* `A` *et qui teste si la liste* `t` *est vide ou non,* `hd l` *qui est de type* `A` *si* `l` *est de type* `A list` *et qui est le premier élément de la liste* `l` *et* `tl l` *qui est de type* `A list` *si* `l` *est de type* `A list` *et qui est la liste* `l` *privée de son premier élément. Donner les règles de typage pour cette extension de PCF. Écrire un vérificateur de types pour cette extension de PCF.*

Programmer la fonction `map` *qui a une fonction* `f` *et une liste* $\texttt{t}_1, \ldots, \texttt{t}_n$ *associe la liste* $\texttt{f t}_1, \ldots, \texttt{f t}_n$*. Quel est son type ?*

Programmer un algorithme de tri. Quel est son type ?

Dans le système de types présenté dans ce chapitre, on peut donner un type quantifié aux variables liées par un `let`. On peut, de même, vouloir donner un type quantifié aux variables liées par un `fun`. Par exemple, donner le type `∀X (X -> X)` à la variable `x` dans le terme `fun x -> x x` rend ce terme typable. Le langage de types obtenu s'appelle le système F de Girard et Reynolds. Cependant, la typabilité est indécidable dans le système F — théorème de Wells — et de ce fait, il ne peut y avoir d'algorithme d'inférence de type pour le système F. De même, rendre polymorphe la variable liée par un `fix` rend la typabilité indécidable — théorème de Kfoury. Restreindre le polymorphisme au `let` est donc un bon compromis qui permet à la fois la réutilisabilité du code et l'inférence de types.