Programmation 1: sémantique, leçon 3

Jean Goubault-Larrecq
ENS Cachan
goubault@lsv.ens-cachan.fr

7 décembre 2020

1 Systèmes de types

La plupart des langages de programmation sont *typés* : les données manipulées sont catégorisées selon leur type, entier, chaîne de caractères, fonction, etc.

Typage statique, typage dynamique. On distingue :

- le *typage statique* : les types sont donnés et vérifiés à la compilation ;
- le *typage dynamique* : les informations de type font partie des données, et on peut tester le type à l'exécution.

Parmi les langages à typage statique, on peut citer Caml ou Haskell, ou bien C, ou bien encore PCF_V. Parmi les langages à typage dynamique, on peut citer Lisp. Lisp n'a pas de type à la compilation (si l'on excepte les déclarations de types de CommonLisp), mais on peut tester les types à l'exécution. Par exemple :

```
(defun f (x)
       (cond
         ((integerp x) "entier")
         ((listp x)    "liste")))
```

définit une fonction `f` qui retourne `"entier"` si on lui passe un entier en paramètre, `"liste"` si on lui passe une liste (et `nil` sinon).

A l'opposé, il n'y a aucune fonction pour tester les types des données en C ou en Caml. Certains langages proposent les deux. C'est le cas notamment de Java, dans une certaine mesure.

Typage fort, typage faible. On distingue aussi les langages à typage *fort* et ceux à typage *faible*. La distinction est plus floue, et l'idée est simplement d'évaluer à quel point le système de types est contraignant.

Celui de C est dit faible, car on peut toujours outrepasser la vérification de types en utilisant les *cast* : si e est une expression de type τ, alors $(\sigma)e$ (le $\ll$ cast de e au type $\sigma \gg$) est une

expression de type σ, et qui intuitivement est toujours e. Ce dernier point est assez flou lui-même, et recouvrent deux phénomènes distincts :

— la *coercition* de types, où la valeur du cast et celle de l'expression est réellement la même ; c'est le cas si σ et τ sont des types pointeurs : si e est de type `int *`, alors `(char *)`e dénote exactement la même adresse en mémoire que e ;

— la *conversion* de types de données, où le cast opère réellement un calcul. Par exemple, si e est de type `char` (entier 8 bits), alors `(int)`e est typiquement l'entier 32 bits obtenu à partir de e en ajoutant 24 bits égaux à 0 si $e \geq 0$, et tous égaux à 1 si $e < 0$ [1].

À l'opposé, celui de Pascal ou celui de Caml est considéré comme fort... nonobstant le fait que Delphi (version moderne de Pascal, orienté objet) a des casts, ainsi que Caml : c'est la fonction `Obj.magic : 'a -> 'b`. (N'essayez pas de l'utiliser ! C'est une arme dangereuse.)

Vérification de types, inférence de types Avant de compiler un programme à proprement parler, les compilateurs C ou Pascal font une étape de *vérification des types*. C'est une étape qui calcule les types des expressions et des affectations par des règles simples telles que :

— pour chaque variable x, x en tant qu'expression a le type τ si la variable x est déclarée de type τ dans la portée courante ;

— $e_1 \dot{+} e_2$ a le type `int` si e_1, e_2 sont bien typées et de type `int`, et est mal typé sinon.

En réalité, la situation en C et Pascal est un peu plus compliquée, car ces langages insèrent des conversions automatiques entre certains types. C'est ce qui permet d'écrire :

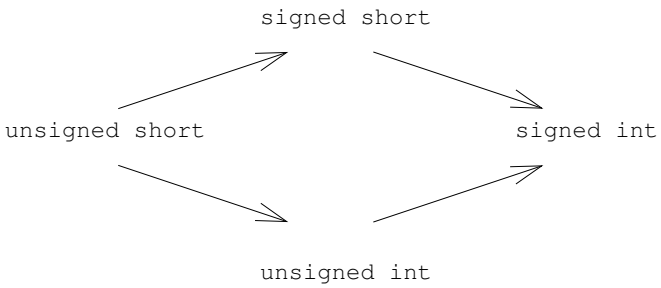
```
{
  double x, y;

  [...]
  y = x+1;
  [...]
}
```

et que le compilateur C comprenne ce que l'on veut dire. Ici, on doit comprend que C a en fait deux opérations d'addition, une pour les entiers, une pour les nombres à virgule flottante (de type `double`). Mais aucune des deux ne s'applique : si `x` est de type `double`, en revanche `1` est de type `int`. (C'est `1.0` qui serait de type `double`.)

Voici comment les compilateurs C procèdent. D'abord, le langage C fixe un ensemble fini de *coercitions implicites*. Ceci se représente aisément par une graphe fini G, dont les sommets sont les types de base (`char`, `signed char`, `unsigned char`, `int`, `unsigned int`, `short`, `long`, `double`, et quelques autres). Les arêtes $\sigma \xrightarrow{c} \tau$ de ce graphe sont étiquetées par des morceaux de code assembleur (typiquement) qui prennent en paramètre une valeur de type σ et retourne une valeur de type τ.

1. Sur une architecture 32 bits, en complément à deux, et en supposant que le compilateur comprenne `char` comme un type d'entiers signés... la norme C laisse le choix au compilateur de considérer `char` comme un type signé ou pas.

On définit le préordre $\preceq$ (relation réflexive et transitive, mais pas nécessairement asymétrique) par $\sigma \preceq \tau$ si et seulement s'il existe un chemin reliant σ à τ dans le graphe G. Pour chaque couple de sommets σ et τ avec $\sigma \preceq \tau$, disons $\sigma = \tau_1 \xrightarrow{c_1} \tau_2 \xrightarrow{c_2} \tau_3 \cdots \tau_{n-1} \xrightarrow{c_{n-1}} \tau_n = \tau$, la composée des coercitions c_1, c_2, …, c_{n-1} est une coercition de σ vers τ. Il peut y en avoir plusieurs ! Le cas typique est l'extrait suivant du graphe G :



On fixe donc pour chaque couple de sommets σ et τ avec $\sigma \preceq \tau$ un *unique* chemin, et donc une unique coercition canonique $c_{\sigma \to \tau}$ de σ vers τ.

On peut décrire le système de vérification de types des expressions C (par ailleurs relativement ad hoc) comme suit. Pour typer l'expression e, on la parcourt récursivement :

— Si e est une variable x, on déclare e typable et de type τ si la variable x est déclarée de type τ dans la portée courante ;

— Si $e = e_1 \dot{+} e_2$, on type e_1 et e_2 récursivement ; si l'une des deux n'est pas typable, alors e n'est pas typable, sinon soit τ_1 le type trouvé de e_1, τ_2 celui de e_2 ; si $\tau_1 \preceq \texttt{int}$ et $\tau_2 \preceq \texttt{int}$ alors e est de type `int`, et l'on doit compiler e en compilant e_1 suivi de la coercition implicite $c_{\tau_1 \to \texttt{int}}$, e_2 suivi de la coercition implicite $c_{\tau_2 \to \texttt{int}}$, puis l'addition entière des deux ; sinon, et si $\tau_1 \preceq \texttt{double}$ et $\tau_2 \preceq \texttt{double}$ alors e est de type `double`, les coercitions implicites étant gérées similairement ; sinon, e n'est pas typable.

— Ainsi de suite pour les autres opérateurs.

(En réalité, le signe $+$ en C peut aussi prendre deux `short` et retourner un `short`, deux `long` et retourner un `long`, ou un pointeur de type τ `*` et un entier et retourner un τ `*`… c'est assez compliqué au final.)

Le système de typage de Caml est beaucoup plus intéressant (même s'il n'offre pas les coercitions implicites) :

— D'abord, on peut non seulement vérifier le type des expressions, mais même l'*inférer* (ou plutôt, en inférer *un*, qui se trouvera être principal, voir plus bas), c'est-à-dire que l'on n'a pas besoin de déclarer les types des variables, et un algorithme, dû à Damas et Milner, va décider si votre programme est typable ou non, et de quel type, et sous quelles hypothèses de typage pour les variables.

— Ensuite, Caml a des types *polymorphes*, c'est-à-dire qu'une même expression peut avoir plusieurs, et même en général une infinité de types. Par exemple, la fonction `fun x -> x` a *tous* les types de la forme $\tau \to \tau$, et la liste vide `nil` a *tous* les types de la forme τ `list`.

— Enfin, le typage de Caml est si fort que l'on peut *prouver des théorèmes* sur les programmes typés. Le premier et le plus important est dû à Milner, et énonce « well-typed programs do not go Wrong », ce qui signifie par exemple que les seuls arguments jamais

fournis à une addition seront entiers. Ce n'est pas le cas en Lisp, où l'on peut allègrement écrire :

```
(+ "abc" 1)
```

(ce qui déclenche une erreur à l'exécution : le typage est *dynamique*). Ce n'est pas le cas non plus en C, où l'on peut écrire :

```
char *x;
int *y;
int z;

z = ((int) x) + ((int) y);
```

avec des résultats à l'exécution pour le moins variables (mais aucune erreur, ni à la compilation, ni à l'exécution).

En général, les systèmes de typage permettent de faire des preuves de certaines propriétés triviales de programmes, de façon complètement automatique (contrairement à la logique de Hoare, par exemple). Nous verrons une autre famille de techniques ayant des buts similaires, l'interprétation abstraite, dans la leçon suivante.

2 Le système de typage de pureML

Considérons un fragment purement applicatif de Caml, c'est-à-dire sans références. (Les références posent effectivement un problème, sur lequel nous reviendrons à la fin.) Nous appelons notre langage *pureML*, et c'est à peu de choses près PCF_V sans aucune annotation de type sur les variables.

La syntaxe est :

$s, t, u, v, \ldots$	$::=$	$x, y, z, \ldots$	variables
	$\mid$	$\dot{n}$	constantes entières
	$\mid$	uv	application
	$\mid$	$\texttt{letrec}\ f(x) = u\ \texttt{in}\ v$	fonction récursive
	$\mid$	$\texttt{let}\ x = u\ \texttt{in}\ v$	définition locale
	$\mid$	$u \dot{+} v$	addition
	$\mid$	$\dot{-} u$	opposé
	$\mid$	$\texttt{if}\ u = 0\ \texttt{then}\ v\ \texttt{else}\ w$	conditionnelle.

Les esprits attentifs auront remarqué que nous avons rajouté la construction $\texttt{let}\ x = u\ \texttt{in}\ v$. En PCF_V, on pouvait définir cette construction par l'expression $\texttt{letrec}\ f(x) = v\ \texttt{in}\ f(u)$, où f est un nom de fonction frais, par exemple. Ici les deux expressions auront la même sémantique mais ne pourront pas en général être typées de la même façon.

$$\frac{}{\Gamma, x : \forall \alpha_1, \cdots, \alpha_n \cdot \sigma \vdash x : \sigma[\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]}\,(Var :) \qquad \frac{}{\Gamma \vdash \dot{n} : \texttt{int}}\,(\texttt{int} :)$$

$$\frac{\Gamma \vdash u : \sigma \to \tau \quad \Gamma \vdash v : \sigma}{\Gamma \vdash uv : \tau}\,(App :) \qquad \frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \forall \vec{\alpha} \cdot \sigma \vdash v : \tau}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau}\,(Let :)$$
où $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$

$$\frac{\Gamma, f : \forall \cdot \sigma \to \tau, x : \forall \cdot \sigma \vdash u : \tau \quad \Gamma, f : \forall \vec{\alpha} \cdot \sigma \to \tau \vdash v : \lambda}{\Gamma \vdash \texttt{letrec}\ f(x) = u\ \texttt{in}\ v : \lambda}\,(Letrec :)$$
où $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma \to \tau) \setminus \mathrm{ftv}(\Gamma)$

$$\frac{\Gamma \vdash u : \texttt{int} \quad \Gamma \vdash v : \texttt{int}}{\Gamma \vdash u \dot{+} v : \texttt{int}}\,(+ :) \qquad \frac{\Gamma \vdash u : \texttt{int}}{\Gamma \vdash \dot{-} u : \texttt{int}}\,(- :)$$

$$\frac{\Gamma \vdash u : \texttt{int} \quad \Gamma \vdash v : \tau \quad \Gamma \vdash w : \tau}{\Gamma \vdash \texttt{if}\ u = 0\ \texttt{then}\ v\ \texttt{else}\ w : \tau}\,(\texttt{if} :)$$

FIGURE 1 – Le système de typage de pureML

2.1 Le système de types

Les types sont définis par la grammaire :

$\sigma, \tau, \ldots$	$::=$	$\alpha, \beta, \ldots$	variables de type
	$\mid$	$\texttt{int}$	type des entiers
	$\mid$	$\sigma \to \tau$	type fonction.

On notera la présence de *variables* de types, qui n'existaient pas en PCF_V. On suppose l'ensemble des variables de types infini dénombrable. En Caml, par exemple, l'ensemble des variables de types est (en bijection avec) l'ensemble des chaînes de caractères obéissant à l'expression régulière `\'\'*[A-Za-z][A-Za-z0-9\'\_]*`.

On définit un système de types par les règles de dérivation de la figure 1. Il a ceci de particulier que :

— Les jugements sont de la forme $\Gamma \vdash u : \tau$, où Γ est un *contexte de typage*, c'est-à-dire un ensemble d'hypothèses de typage de la forme $x : \forall \vec{\alpha} \cdot \sigma$, où les variables x sont distinctes deux à deux ; la notation $\Gamma, x : \forall \vec{\alpha} \cdot \sigma$ dénote Γ, moins toute hypothèse de la forme $x : \cdots$, plus l'hypothèse $x : \forall \vec{\alpha} \cdot \sigma$; on verra aussi Γ comme une fonction de domaine fini qui à chaque x associe le schéma de types $\forall \vec{\alpha} \cdot \sigma$ correspondant, ce qui permet d'écrire $\Gamma(x)$, ou $\mathrm{dom}\ \Gamma$;

— dans ces hypothèses, une expression de la forme $\forall \vec{\alpha} \cdot \sigma$, où $\vec{\alpha}$ est un ensemble fini de variables de types, est un *schéma de types*, et non un type ; ceci permet le polymorphisme, et $x : \forall \vec{\alpha} \cdot \sigma$ se lit intuitivement comme « x a le type σ, pour n'importe quels types

remplaçant les variables de types $\vec{\alpha} \gg$. Cette lecture est justifiée par la règle $(Var\ :)$, où les types τ_1, …, τ_n sont quelconques. (La notation $\sigma[\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]$ sera expliquée plus bas.)

Attention ! L'ensemble $\vec{\alpha}$ peut être vide ; dans ce cas, on notera $\forall \cdot \sigma$ le schéma de type $\forall \vec{\alpha} \cdot \sigma$, et il ne devrait pas être confondu avec le type σ : seuls des schémas de types peuvent se trouver dans le contexte Γ, seul un type τ peut se trouver à l'extrémité droite du jugement $\Gamma \vdash u : \tau$. Bien sûr, sémantiquement $\forall \cdot \sigma$ et σ seront indistinguables.

— Les règles de typage $(Let\ :)$ et $(Letrec\ :)$, du `let` et du `letrec` respectivement, effectuent une *généralisation*, c'est-à-dire passent d'un type σ à un schéma de type $\forall \vec{\alpha} \cdot \sigma$.

Un objet $\theta = [\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]$ est appelé une *substitution* (de types), et dénote la fonction qui à chaque α_i associe τ_i, et à chaque variables $\alpha \notin \{\alpha_1, \cdots, \alpha_n\}$ associe α. En général, une substitution est une fonction θ des variables de types vers les types dont le *domaine* $\mathrm{dom}\ \theta$ est $\{\alpha \mid \theta(\alpha) \neq \alpha\}$. (Attention : le domaine n'est *pas* en général $\{\alpha_1, \cdots, \alpha_n\}$ et peut être plus petit, par exemple si $\tau_i = \alpha_i$ pour un certain i.)

La notation $\sigma[\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]$ utilisée dans la règle $(Var\ :)$, en abrégé $\sigma\theta$, opère le remplacement simultané des α_i par τ_i dans σ. Formellement :

$$\begin{array}{rcll} \alpha\theta & = & \theta(\alpha) & (\alpha \text{ variable de type}) \\ \texttt{int}\theta & = & \texttt{int} & \\ (\sigma \to \tau)\theta & = & \sigma\theta \to \tau\theta. & \end{array}$$

La généralisation effectuée dans les règles $(Let\ :)$ et $(Letrec\ :)$ permet notamment de considérer que lorsqu'on écrit `letrec` $f(x) = x$ `in` $\cdots$ la fonction f est non seulement de type $\alpha \to \alpha$ (via la règle $(Letrec\ :)$ avec $\sigma = \tau = \alpha$, et la règle $(Var\ :)$ avec la substitution identité), mais encore de (schéma de) type $\forall \alpha \cdot \alpha \to \alpha$. Ceci permettra d'appliquer f ultérieurement à des éléments de types arbitraires, à des entiers, à des fonctions, etc.

La généralisation est permise sur toutes les variables de types qui sont *libres* dans le type σ de u dans la règle $(Let\ :)$. (Le cas de $(Letrec)$ est similaire.) La notation $\mathrm{ftv}(\sigma)$ dénote l'ensemble de ces variables libres : ce sont juste les variables qui apparaissent dans σ. On étend ces notations aux schémas de types par $\mathrm{ftv}(\forall \vec{\alpha} \cdot \sigma) = \mathrm{ftv}(\sigma) \setminus \vec{\alpha}$, et aux contextes $\Gamma = x_1 : \forall \vec{\alpha}_1 \cdot \sigma_1, \cdots, x_n : \forall \vec{\alpha}_n \cdot \sigma_n$ par $\mathrm{ftv}(\Gamma) = \bigcup_{i=1}^{n} \mathrm{ftv}(\forall \vec{\alpha}_i \cdot \sigma_i)$.

Par exemple, dans le programme suivant :

`letrec` $f(x) = ($`letrec` $g(y) = x$ `in` $u)$ `in` v

on pourra avoir à typer le corps x de la fonction g dans un contexte Γ de la forme $f : \forall \cdot \alpha \to \beta, x : \forall \cdot \alpha, g : \forall \cdot \gamma \to \delta, y : \forall \cdot \gamma$, où α, β, γ, δ sont des variables de types distinctes. On obtiendra ainsi une dérivation de $\Gamma \vdash x : \alpha$ (par exemple). La règle $(Letrec\ :)$ s'appliquera alors avec $\sigma = \gamma$, $\tau = \alpha$, et l'on pourra ensuite typer u dans le contexte $f : \forall \cdot \alpha \to \beta, x : \forall \cdot \alpha, g : \forall \gamma \cdot \gamma \to \alpha$. On a donc généralisé ici sur γ (qui est libre dans $\gamma \to \alpha$ mais pas dans $f : \forall \cdot \alpha \to \beta, x : \forall \cdot \alpha$), mais on ne peut pas généraliser sur α, qui est libre dans $f : \forall \cdot \alpha \to \beta, x : \forall \cdot \alpha$.

Nous aurons besoin d'une série de lemmes sur les substitutions. Nous en présentons quelques-uns ici, nous verrons les autres en section 2.5.

D'abord, on peut définir la *composée* $\theta_1\theta_2$ de deux substitutions, comme étant la substitution qui à toute variable associe $(\alpha\theta_1)\theta_2$. Plus concrètement, $\theta_1\theta_2$ est la substitution de domaine inclus

dans $\mathrm{dom}\ \theta_1 \cup \mathrm{dom}\ \theta_2$ qui à tout $\alpha \in \mathrm{dom}\ \theta_1$ associe $\theta_1(\alpha)\theta_2$, à tout $\alpha \in \mathrm{dom}\ \theta_2 \setminus \mathrm{dom}\ \theta_1$ associe $\theta_2(\alpha)$. (Une notation plus traditionnelle en mathématiques serait $\theta_2 \circ \theta_1$, donc dans l'autre sens ; mais ce n'est pas non plus exactement la notion de composition de deux *fonctions* que nous sommes en train de définir.) Formellement,

Lemme 1 *Pour tout type τ, pour toutes substitutions θ_1, θ_2, $\tau(\theta_1\theta_2) = (\tau\theta_1)\theta_2$.*
La composition de substitutions est associative.

Démonstration. La première partie du lemme est par une récurrence structurelle immédiate sur τ. La seconde s'en déduit facilement. □

On écrira donc $\tau\theta_1\theta_2$, ou $\theta_1\theta_2\theta_3$, sans parenthèses, sans que cela induise aucune ambiguïté.

On appellera *renommage* (de $\vec{\alpha}$) toute substitution ϱ telle que $\varrho(\alpha)$ est une variable de type pour toute variable de type α, et qui est injective (de domaine $\vec{\alpha}$). Par exemple, $[\alpha_1 := \beta_1, \alpha_2 := \beta_2]$ est un renommage si β_1 et β_2 sont deux variables de types distinctes.

On dira que deux schémas de type $\forall\vec{\alpha} \cdot \sigma$ et $\forall\vec{\beta} \cdot \tau$ sont *équivalents* (le nom usuel est « α-équivalent », mais le préfixe α n'apporte rien ici est prêterait confusion avec le nom de certaines variables de type) si et seulement s'il existe un renommage ϱ de domaine $\mathrm{dom}\ \varrho \subseteq \vec{\alpha}$ et de *codomaine* $\mathrm{dom}\ \varrho^{-1} = \vec{\beta}$ tel que $\sigma\varrho = \tau$ et $\tau\varrho^{-1} = \sigma$. Par exemple, $\forall\alpha \cdot \alpha \to \alpha$ et $\forall\beta \cdot \beta \to \beta$ sont équivalents, mais $\forall\alpha \cdot \alpha \to \alpha$ et $\forall\alpha, \beta \cdot \alpha \to \beta$ ne le sont pas.

Par extension, on dira que deux contextes de typage Γ et Γ' sont équivalents si et seulement si $\mathrm{dom}\ \Gamma = \mathrm{dom}\ \Gamma'$ et pour tout $x \in \mathrm{dom}\ \Gamma$, $\Gamma(x)$ et $\Gamma'(x)$ sont des schémas de type équivalents.

Lemme 2 *Si Γ et Γ' sont deux contextes de typage équivalents, et que $\Gamma \vdash u : \tau$ est dérivable, alors $\Gamma' \vdash u : \tau$ est aussi dérivable, par une dérivation de même taille.*

Démonstration. C'est pratiquement une évidence : on remplace Γ partout par Γ' dans la dérivation. Si l'on est formaliste, il s'agit d'une récurrence structurelle sur la dérivation donnée de $\Gamma \vdash u : \tau$. L'unique cas où il se passe quelque chose est la règle $(Var :)$. On a alors dérivé $\Gamma \vdash x : \sigma\theta$ pour une certaine substitution θ de domaine $\vec{\alpha}$, où $\Gamma(x) = \forall\vec{\alpha} \cdot \sigma$. Nécessairement, $\Gamma'(x) = \forall\vec{\beta} \cdot \sigma\varrho$ où ϱ est un renommage de domaine $\vec{\alpha}$ et de codomaine $\vec{\beta}$. On peut alors dériver $\Gamma' \vdash x : (\sigma\varrho)(\varrho^{-1}\theta)$ par $(Var :)$, c'est-à-dire $\Gamma' \vdash x : \sigma\theta$, en utilisant le lemme 1. □

Nous considérerons donc, plus ou moins implicitement, les schémas de types et les jugements *à équivalence près*.

Modulo équivalence, il est facile d'étendre la notion de substitution aux *schémas* de types. On pose ainsi $(\forall\vec{\alpha}\cdot\sigma)\theta = \forall\vec{\alpha}\cdot(\sigma\theta)\ldots$ mais ceci n'a de sens que si $\vec{\alpha} \cap \mathrm{dom}\ \theta = \emptyset$ (sinon, par exemple, on admettrait que $(\forall\alpha \cdot \alpha)[\alpha := \mathtt{int}] = \forall\alpha \cdot \mathtt{int}$, ce qui est absurde) et que si $\vec{\alpha} \cap \mathrm{ftv}(\theta(\beta)) = \emptyset$ pour tout $\beta \in \mathrm{dom}\ \theta$ (sinon, par exemple, on admettrait que $(\forall\alpha \cdot \beta)[\beta := \alpha] = \forall\alpha \cdot \alpha$, ce qui semble absurde aussi). Ces deux conditions sont faciles à assurer modulo équivalence : il suffit de renommer les variables $\vec{\alpha}$ en des variables de types fraîches, hors de $\mathrm{dom}\ \theta \cup \bigcup_{\beta \in \mathrm{dom}\ \Theta} \mathrm{ftv}(\beta\theta)$.

Le lemme suivant sera crucial. Il énonce que si on peut dériver un jugement de typage $\Gamma \vdash u : \tau$, avec des variables de types libres, on peut remplacer ces variables de types par n'importe quels types. Par exemple, si on peut dériver $f : \forall \cdot \alpha \to \alpha, x : \forall\beta \cdot \beta \to \alpha \to \mathtt{int} \vdash u : \alpha$, alors on pourra dériver $f : \forall \cdot \tau \to \tau, x : \forall\beta \cdot \beta \to \tau \to \mathtt{int} \vdash u : \tau$ pour n'importe quel type τ (… après renommage de β si nécessaire).

Lemme 3 *Si $\Gamma \vdash u : \sigma$ est dérivable dans le système de typage de la figure 1, et θ est une substitution, alors $\Gamma\theta \vdash u : \sigma\theta$ est aussi dérivable.*

Démonstration. Ceci peut paraître évident : appliquer la substitution θ à tous les jugements de la dérivation donnée. Ce qui n'est pas si évident, c'est que les conditions de bord des règles $(Let :)$ et $(Letrec :)$ restent satisfaites.

On effectue donc une récurrence sur la taille de la dérivation de typage donnée. Nous nous concentrons sur les règles $(Let :)$ et $(Letrec :)$, sans surprise. Le cas de la règle $(Var :)$ est en fait le plus délicat, et nous commençons par lui.

Règle $(Var :)$. Par hypothèse, nous avons une dérivation de $\Gamma, x : \forall\alpha_1, \cdots, \alpha_n \cdot \sigma \vdash x : \sigma\theta_1$, avec θ_1 de la forme $[\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]$, et nous voulons produire une dérivation de $\Gamma\theta, x : (\forall\alpha_1, \cdots, \alpha_n \cdot \sigma)\theta \vdash x : \sigma\theta_1\theta$. Par le lemme 2, on peut, quitte à renommer les variables de $\vec{\alpha}$, supposer qu'aucune variable α_i, $1 \leq i \leq n$, n'appartient à $\mathrm{dom}\ \theta \cup \bigcup_{\beta\in\mathrm{dom}\ \theta} \mathrm{ftv}(\beta\theta)$. La dérivation que nous devons trouver est donc une dérivation de $\Gamma\theta, x : \forall\alpha_1, \cdots, \alpha_n \cdot \sigma\theta \vdash x : \sigma\theta_1\theta$. Par $(Var :)$, ce que l'on peut obtenir à la place, c'est une dérivation de $\Gamma\theta, x : \forall\alpha_1, \cdots, \alpha_n \cdot \sigma\theta \vdash x : \sigma\theta\theta'_1$, pour une substitution arbitraire θ'_1 de domaine $\{\alpha_1, \cdots, \alpha_n\}$. Nous cherchons donc θ'_1 telle que $\theta\theta'_1 = \theta_1\theta$. Le choix est imposé : comme $\alpha_i \notin \mathrm{dom}\ \theta$, $\alpha_i\theta\theta'_1 = \theta'_1(\alpha_i)$, donc on est obligé de définir $\theta'_1(\alpha_i)$ comme égal à $\alpha_i\theta_1\theta$. Autrement dit, on pose θ'_1 égale à $\theta_1\theta$, restreinte à $\{\alpha_1, \cdots, \alpha_n\}$. On vérifie ensuite que ce choix de θ'_1 répond bien à nos attentes, ce qui est facile (mais un peu laborieux si on est méticuleux).

Règle $(Let :)$. Nous avons une dérivation de $\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau$, obtenue à partir de deux dérivations plus petites ;

(1) de $\Gamma \vdash u : \sigma$ d'une part,

(2) et de $\Gamma, x : \forall\vec{\alpha} \cdot \sigma \vdash v : \tau$ d'autre part, où $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$.

Par le lemme 2, on a une dérivation de même taille que (2) de $\Gamma, x : \forall\vec{\beta} \cdot \sigma\varrho \vdash v : \tau$, où ϱ est un renommage des $\vec{\alpha}$ en des variables fraîches $\vec{\beta}$. (Par « fraîche », il suffira de comprendre distinctes deux à deux, et hors de $\mathrm{dom}\ \theta \cup \bigcup_{\gamma\in\mathrm{dom}\ \theta} \mathrm{ftv}(\gamma\theta)$.) Par hypothèse de récurrence, on obtient ainsi une dérivation de $\Gamma\theta, x : (\forall\vec{\beta} \cdot \sigma\varrho)\theta \vdash v : \tau\theta$, c'est-à-dire de $\Gamma\theta, x : \forall\vec{\beta} \cdot \sigma\varrho\theta \vdash v : \tau\theta$. (C'est pour cette dernière formule que l'on est obligé d'introduire le renommage ϱ.) On souhaiterait appliquer la règle $(Let :)$ pour en déduire une dérivation de $\Gamma\theta \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau\theta$, et ce qui nous manque pour cela est une dérivation de la prémisse manquante, à savoir $\Gamma\theta \vdash u : \sigma\varrho\theta$. Par chance (vraiment ?), $\Gamma\theta = \Gamma\varrho\theta$, parce que $\Gamma = \Gamma\varrho : \varrho$ remplace les $\vec{\alpha}$, mais aucune $\vec{\alpha}$ n'est libre dans un schéma de type de Γ. On obtient donc la prémisse manquante par hypothèse de récurrence appliquée à (1), avec la substitution $\varrho\theta$ plutôt que θ.

Si l'on résume, on a trouvé une dérivation de $\Gamma\theta \vdash u : \sigma\varrho\theta$ et une de $\Gamma\theta, x : (\forall\vec{\beta} \cdot \sigma\varrho)\theta \vdash v : \tau\theta$. On peut appliquer $(Let :)$ et en déduire le jugement désiré $\Gamma\theta \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau\theta\ldots$ à condition de vérifier que $\vec{\beta} \subseteq \mathrm{ftv}(\sigma\varrho\theta) \setminus \mathrm{ftv}(\Gamma\varrho\theta)$. Or, d'une part, toute variable β_i s'écrit $\varrho(\alpha_i)$, et α_i est libre dans σ (car $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$) et $\beta_i \notin \mathrm{dom}\ \theta$ (par construction des variables fraîches $\vec{\beta}$) ; donc $\vec{\beta} \subseteq \mathrm{ftv}(\sigma\varrho\theta)$. Si l'une des variables β_i était dans $\mathrm{ftv}(\Gamma\varrho\theta)$, il y aurait une liaison $x : \tau$ dans Γ telle que β_i soit libre dans $\tau\varrho\theta$. Comme β_i n'est libre dans aucun terme $\gamma\theta$ avec $\gamma \in \mathrm{dom}\ \theta$, on voit que β_i doit être libre dans $\tau\varrho$. Ceci n'est possible que si α_i est libre dans τ, ce qui est impossible puisque $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$.

$$\frac{}{E \vdash x \Rightarrow E(x)} \; (Var) \quad \text{si } x \in \mathrm{dom}\ E \qquad \frac{}{E \vdash \dot{n} \Rightarrow \dot{n}} \; (Cst)$$

$$\frac{E \vdash u \Rightarrow \langle \mathtt{rec}\, f(x) = u', E' \rangle \quad E \vdash v \Rightarrow V \quad E'[f \mapsto \langle \mathtt{rec}\, f(x) = u', E' \rangle, x \mapsto V] \vdash u' \Rightarrow V'}{E \vdash uv \Rightarrow V'} \; (App)$$

$$\frac{E \vdash u \Rightarrow V \quad E[x \mapsto V] \vdash v \Rightarrow V'}{E \vdash \mathtt{let}\ x = u\ \mathtt{in}\ v \Rightarrow V'} \; (Let) \qquad \frac{E[f \mapsto \langle \mathtt{rec}\, f(x) = u, E \rangle] \vdash v \Rightarrow V}{E \vdash \mathtt{letrec}\ f(x) = u\ \mathtt{in}\ v \Rightarrow V} \; (Letrec)$$

$$\frac{E \vdash u \Rightarrow \dot{n}_1 \quad E \vdash v \Rightarrow \dot{n}_2}{E \vdash u \dot{+} v \Rightarrow \dot{n}} \; (\dot{+}) \quad \text{où } n = n_1 + n_2 \qquad \frac{E \vdash u \Rightarrow \dot{n}}{E \vdash \dot{-}u \Rightarrow \dot{-}\dot{n}} \; (\dot{-})$$

$$\frac{E \vdash u \Rightarrow \dot{0} \quad E \vdash v \Rightarrow V}{E \vdash \mathtt{if}\ u = 0\ \mathtt{then}\ v\ \mathtt{else}\ w \Rightarrow V} \; (\mathtt{if}_0) \qquad \frac{E \vdash u \Rightarrow \dot{n} \quad E \vdash w \Rightarrow V}{E \vdash \mathtt{if}\ u = 0\ \mathtt{then}\ v\ \mathtt{else}\ w \Rightarrow V} \; (\mathtt{if}_1) \quad \text{si } n \neq 0$$

FIGURE 2 – La sémantique à grands pas de pureML

Le cas de la règle $(Letrec\ :)$ est entièrement similaire. Les autres règles sont des appels immédiats à l'hypothèse de récurrence. □

Corollaire 4 *Si* $\Gamma \vdash u : \sigma$ *est dérivable dans le système de typage de la figure 1, et* θ *est une substitution de domaine* $\mathrm{dom}\ \theta$ *disjoint de* $\mathrm{ftv}(\Gamma)$*, alors* $\Gamma \vdash u : \sigma\theta$ *est aussi dérivable.*

2.2 La sémantique

Nous avons comme d'habitude le choix entre une sémantique opérationnelle à petits pas, une sémantique opérationnelle à grands pas, et une sémantique dénotationnelle.

Nous allons choisir de définir la sémantique de pureML par une sémantique opérationnelle à grands pas, d'abord pour sa ressemblance avec celle de PCF_V. Ensuite, le résultat principal de Milner que nous présentons plus loin ne parle que des exécutions qui terminent, et il n'y a aucun raison de préférer petits pas à grands pas dans ce contexte. Enfin, la sémantique dénotationnelle de pureML pose problème : pour la définir, nous devrions trouver un dcpo qui contienne son propre espace de fonctions. C'est possible, mais compliqué, et je préférerais éviter cette complication supplémentaire.

La sémantique de pureML est donc essentiellement celle de PCF_V, sans les types et annotations de types. Définissons d'abord les *ML-valeurs*, analogue des P-valeurs.

Par définition, l'ensemble des ML-valeurs est le plus petit ensemble contenant les constantes $\dot{n}$, $n \in \mathbb{Z}$, et les clôtures $\langle \mathtt{rec}\, f(x) = u, E \rangle$, où E est un *ML-environnement*, c'est-à-dire une

$$\frac{}{\triangleright \dot{n} : \texttt{int}}\,(Val : \texttt{int}) \qquad \frac{\triangleright E : \Gamma \quad \Gamma, f : \forall \cdot \sigma \to \tau, x : \forall \cdot \sigma \vdash u : \tau}{\triangleright \langle \texttt{rec}\, f(x) = u, E \rangle : \sigma \to \tau}\,(Val : Fun)$$

FIGURE 3 – Typage des valeurs

fonction partielle de domaine fini $\mathrm{dom}\ E$. On n'impose aucune contrainte de typage, contrairement à PCF$_V$. Mais la définition est toujours récursive, et peut être vue comme une définition de certains arbres finis.

La sémantique est donnée en figure 2, et est bien sûre exactement la même que pour PCF$_V$, sauf pour l'absence de types, et la règle sur la construction nouvelle $\texttt{let}\ x = u\ \texttt{in}\ v$.

2.3 Invariance des types le long des calculs

Le théorème de Milner sera une conséquence simple de la proposition 5 ci-dessous, dite d'*invariance des types* le long des calculs.

Ceci devrait être intuitif, et dit que si u de type τ s'évalue en une valeur V, alors V est aussi de type τ. Ce genre de résultat s'appelle souvent « subject reduction » (« auto-réduction ») ou « type soundness » dans la littérature. À dire vrai, l'appel par valeur en fait un théorème plus compliqué ici que ce qu'on voit usuellement dans la littérature. Le polymorphisme complique aussi passablement les choses.

Nous aurons besoin des notions suivantes : On dit que la valeur V est de type τ, si et seulement si le jugement $\triangleright V : \tau$ est déductible dans le système de la figure 3. On écrit $\triangleright V : \forall \vec{\alpha} \cdot \tau$ pour dire que $\triangleright V : \tau\theta$ est dérivable pour toute substitution θ de domaine α (attention : on ne remplace *que* les variables quantifiées ; par exemple, $V : \forall \beta \cdot \beta \to \alpha$ signifie que $V : \tau \to \alpha$ pour tout type τ). On écrit aussi $\triangleright E : \Gamma$ pour énoncer que $\mathrm{dom}\ E \subseteq \mathrm{dom}\ \Gamma$ et que pour tout $x \in \mathrm{dom}\ E$, $\triangleright E(x) : \Gamma(x)$ est dérivable. Dans la règle $(Val : Fun)$, Γ est une contexte de typage quelconque : il suffit d'en trouver un satisfaisant les prémisses pour pouvoir dériver la conclusion.

Proposition 5 *Si $E \vdash u \Rightarrow V$ est dérivable dans la sémantique de pureML (figure 2), si $\Gamma \vdash u : \tau$ est dérivable dans le système de la figure 1, et si d'autre part $\triangleright E : \Gamma$ alors $\triangleright V : \tau$ est dérivable.*

Démonstration. On a ici deux dérivations, une de $E \vdash u \Rightarrow V$, l'autre de $\Gamma \vdash u : \tau$, et on peut imaginer effectuer une récurrence sur l'une ou sur l'autre. Il se trouve que seule la récurrence sur la dérivation de $E \vdash u \Rightarrow V$ (c'est-à-dire le long de l'exécution du programme) a une chance de réussir. Nous appliquerons un raisonnement par inversion sur l'autre dérivation. Les détails, relativement pénibles, sont en annexe A. □

$$\frac{E \vdash u \Rightarrow V}{E \vdash uv \Rightarrow \mathsf{Wrong}}\ (App_{\mathsf{Wrong}}) \quad \text{si } V \text{ pas une clôture}$$

$$\frac{E \vdash u \Rightarrow V}{E \vdash \mathtt{if}\ u = 0\ \mathtt{then}\ v\ \mathtt{else}\ w \Rightarrow \mathsf{Wrong}}\ (\mathtt{if}_{\mathsf{Wrong}}) \quad \text{si } V \text{ pas un entier}$$

$$\frac{E \vdash u \Rightarrow V_1 \quad E \vdash v \Rightarrow V_2}{E \vdash u \dot{+} v \Rightarrow \mathsf{Wrong}}\ (\dot{+}_{\mathsf{Wrong}}1) \quad \text{si } V_1 \text{ pas un entier}$$

$$\frac{E \vdash u \Rightarrow V_1 \quad E \vdash v \Rightarrow V_2}{E \vdash u \dot{+} v \Rightarrow \mathsf{Wrong}}\ (\dot{+}_{\mathsf{Wrong}}2) \quad \text{si } V_2 \text{ pas un entier}$$

$$\frac{E \vdash u \Rightarrow V}{E \vdash \dot{-}u \Rightarrow \mathsf{Wrong}}\ (\dot{-}_{\mathsf{Wrong}}) \quad \text{si } V \text{ pas un entier}$$

FIGURE 4 – Tests de types à l'exécution pour pureML

2.4 Le théorème de Milner

La sémantique de la figure 2 *bloque* lors d'erreurs de type à l'exécution. Par erreur de type, on entend pour l'instant la notion vague selon laquelle une fonction attendant des entiers (comme $\dot{+}$) aurait un ou plusieurs arguments non entiers, ou selon laquelle dans une application uv, u ne s'évalue pas en une clôture.

La sémantique, étant à grands pas, ne fait pas la différence entre un tel bloquage et la non-terminaison. La sémantique de la figure 2, complétée des règles de la figure 4 et de celles de la figure 5, fait cette différence. (Les règles de la figure 5 expriment, de façon par ailleurs extrêmement inélégante, que Wrong est une exception non capturée.) Cette sémantique utilise des jugements de la forme $E \vdash u \Rightarrow \mathsf{Wrong}$ exprimant que dans l'environnement E, u peut à l'exécution, un jour, fournir une erreur de type. Nous conservons la convention que V, V_1, V_2, dénotent des valeurs, et donc différentes de Wrong. Les nouvelles règles qui sont intéressantes sont celles de la figure 4. Les autres se contentent de propager Wrong : une façon de voir est que Wrong est une exception qui n'a pas de handler.

Cette nouvelle sémantique est une sémantique d'un langage à *typage dynamique*, comme Lisp. (Voir la section 1.) Le théorème de Milner dira alors que dans un langage comme pureML, les programmes bien typés ne peuvent jamais échouer à aucun test de typage à l'exécution. Ces tests sont donc redondants, et un compilateur peut se dispenser de produire du code assembleur pour ces tests.

Théorème 6 (« Well-typed programs do not go Wrong **»)** *Si* $\Gamma \vdash u : \tau$ *est dérivable et* $\triangleright E : \Gamma$*, alors il n'y a pas de dérivation de* $E \vdash u \Rightarrow \mathsf{Wrong}$ *par les règles des figures 2, 4 et 5.*

Démonstration. On réalise d'abord que toute dérivation de $E \vdash u \Rightarrow V$ dans le système formé par les règles des figures 2, 4 et 5, n'utilise en fait que des règles de la figure 2. Intuitivement, c'est parce qu'on ne peut pas produire l'exception Wrong et la capturer plus tard. Formellement,

$$\frac{E \vdash u \Rightarrow \mathsf{Wrong}}{E \vdash uv \Rightarrow \mathsf{Wrong}}\,(PApp_{\mathsf{Wrong}}1) \quad \frac{E \vdash u \Rightarrow \langle \mathtt{rec}\, f(x) = u', E' \rangle \quad E \vdash v \Rightarrow \mathsf{Wrong}}{E \vdash uv \Rightarrow \mathsf{Wrong}}\,(PApp_{\mathsf{Wrong}}2)$$

$$\frac{E \vdash u \Rightarrow \mathsf{Wrong}}{E \vdash \mathtt{let}\ x = u\ \mathtt{in}\ v \Rightarrow \mathsf{Wrong}}\,(PLet_{\mathsf{Wrong}}1) \quad \frac{E \vdash u \Rightarrow V \quad E[x \mapsto V] \vdash v \Rightarrow \mathsf{Wrong}}{E \vdash \mathtt{let}\ x = u\ \mathtt{in}\ v \Rightarrow \mathsf{Wrong}}\,(PLet_{\mathsf{Wrong}}2)$$

$$\frac{E[f \mapsto \langle \mathtt{rec}\, f(x) = u, E \rangle] \vdash v \Rightarrow \mathsf{Wrong}}{E \vdash \mathtt{letrec}\ f(x) = u\ \mathtt{in}\ v \Rightarrow \mathsf{Wrong}}\,(PLetrec_{\mathsf{Wrong}})$$

$$\frac{E \vdash u \Rightarrow \mathsf{Wrong} \quad E \vdash v \Rightarrow V_2}{E \vdash u \dot{+} v \Rightarrow \mathsf{Wrong}}\,(P\dot{+}_{\mathsf{Wrong}}1) \quad \frac{E \vdash u \Rightarrow V_1 \quad E \vdash v \Rightarrow \mathsf{Wrong}}{E \vdash u \dot{+} v \Rightarrow \mathsf{Wrong}}\,(P\dot{+}_{\mathsf{Wrong}}2)$$

$$\frac{E \vdash u \Rightarrow \mathsf{Wrong} \quad E \vdash v \Rightarrow \mathsf{Wrong}}{E \vdash u \dot{+} v \Rightarrow \mathsf{Wrong}}\,(P\dot{+}_{\mathsf{Wrong}}3) \quad \frac{E \vdash u \Rightarrow \mathsf{Wrong}}{E \vdash \dot{-}u \Rightarrow \mathsf{Wrong}}\,(P\dot{-}_{\mathsf{Wrong}})$$

$$\frac{E \vdash u \Rightarrow \mathsf{Wrong}}{E \vdash \mathtt{if}\ u = 0\ \mathtt{then}\ v\ \mathtt{else}\ w \Rightarrow \mathsf{Wrong}}\,(P\,\mathtt{if}_{\mathsf{Wrong}})$$

$$\frac{E \vdash u \Rightarrow \dot{0} \quad E \vdash v \Rightarrow \mathsf{Wrong}}{E \vdash \mathtt{if}\ u = 0\ \mathtt{then}\ v\ \mathtt{else}\ w \Rightarrow \mathsf{Wrong}}\,(P\,\mathtt{if}_{0\mathsf{Wrong}}) \quad \frac{E \vdash u \Rightarrow \dot{n} \quad E \vdash w \Rightarrow \mathsf{Wrong}}{E \vdash \mathtt{if}\ u = 0\ \mathtt{then}\ v\ \mathtt{else}\ w \Rightarrow \mathsf{Wrong}}\,(P\,\mathtt{if}_{1\mathsf{Wrong}})$$

si $n \neq 0$

FIGURE 5 – Propagation de Wrong

c'est évidemment un raisonnement par récurrence, le point essentiel étant que toute règle ayant une prémisse de la forme $\cdots \vdash \cdots \Rightarrow \mathsf{Wrong}$ a une conclusion de la même forme.

Supposons maintenant que, contrairement à ce que le théorème énonce, il existe une expression u, un P-environnement E, une dérivation de typage dérivant un jugement de la forme $\Gamma \vdash u : \tau$ avec $\triangleright E : \Gamma$, et une dérivation π de $E \vdash u \Rightarrow \mathsf{Wrong}$ par les règles des figures 2, 4 et 5. On peut supposer π de taille minimale. (Logiquement, cet argument de contre-exemple minimal est équivalent à une récurrence sur la taille de π, mais sera plus clair que ce dernier.)

Regardons la dernière règle de π. Nous montrons que ce ne peut pas être une des règles de vérification de types de la figure 4. Le raisonnement est le même dans chaque cas, ne traitons donc que de la règle (App_{Wrong}). La situation est la suivante : nous avons une dérivation minimale π, sa conclusion est le jugement $E \vdash uv \Rightarrow \mathsf{Wrong}$, et on peut dériver le jugement de typage $\Gamma \vdash uv : \tau$. Par inversion, on pouvait donc dériver $\Gamma \vdash u : \sigma \to \tau$ pour un certain type σ. D'autre part, la prémisse de (App_{Wrong}) est la conclusion d'une dérivation de $E \vdash u \Rightarrow V$ plus courte que π, et où V n'est pas une clôture (et pas Wrong non plus, par convention). Par la remarque par laquelle nous avons démarré la démonstration, cette dérivation est une dérivation dans le système de la figure 2 seule. Donc la proposition 5 s'applique, impliquant que $\triangleright V : \sigma \to \tau$ est dérivable. Mais ceci contredit le fait que V ne soit pas une clôture (voir la figure 3).

Nous montrons ensuite que la dernière règle de π ne peut pas être non plus une des règles de propagation de la figure 5. Ceci est clair pour la plupart des règles de propagation, en faisant appel à l'inversion sur la dérivation de typage, sauf $(PLet_{\mathsf{Wrong}}2)$ et $(PLetrec_{\mathsf{Wrong}})$. Pour $(PLet_{\mathsf{Wrong}}2)$, on suppose une dérivation π de $E \vdash \texttt{let}\ x = u\ \texttt{in}\ v \Rightarrow \mathsf{Wrong}$ se terminant par cette règle, et une dérivation de typage de $\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau$. Par inversion sur cette dernière, on a pu dériver $\Gamma \vdash u : \sigma$ et $\Gamma, x : \forall \vec{\alpha} \cdot \sigma \vdash v : \tau$ avec $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$. Par le corollaire 4, pour toute substitution θ de domaine inclus dans $\vec{\alpha}$, on a une dérivation de $\Gamma \vdash u : \sigma\theta$. Parmi les prémisses de $(PLet_{\mathsf{Wrong}}2)$, on a une dérivation plus courte que π de $E \vdash u \Rightarrow V$, et une autre de $E[x \mapsto V] \vdash v : \mathsf{Wrong}$. Par la remarque faite en début de démonstration, la dérivation de $E \vdash u \Rightarrow V$ se fait avec les règles de la figure 2 seule, donc la proposition 5 s'applique : $\triangleright V : \sigma\theta$ est dérivable. Comme θ est arbitraire, $\triangleright V : \forall \vec{\alpha} \cdot \sigma$, donc $\triangleright E[x \mapsto V] : (\Gamma, x : \forall \vec{\alpha} \cdot \sigma)$. Nous avons cependant une dérivation de typage $\Gamma, x : \forall \vec{\alpha} \cdot \sigma \vdash v : \tau$ et une dérivation de $E[x \mapsto V] \vdash v : \mathsf{Wrong}$ plus courte que π, et ceci contredit la minimalité de π. Le cas de $(PLetrec_{\mathsf{Wrong}})$ est similaire, mais utilise le lemme 3 plutôt que le corollaire 4.

Donc π se termine par une des règles de la figure 2. Mais ceci ne permet pas de déduire un jugement de la forme $\cdots \vdash \cdots \Rightarrow \mathsf{Wrong}$, ce qui conclut l'argument. □

2.5 Inférence de types, l'algorithme de Hindley

Un point agréable avec le langage ML, ainsi qu'avec ses cousins pureML (ici) ou Haskell, est que l'on n'a pas besoin de déclarer les types des variables, ou d'écrire les types de retour des fonctions. Ils seront inférés. Par exemple, si on écrit le code suivant en Caml :

```
let rec f x =
    if x<=1
       then 1
```

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\,(Var :_{\text{simple}}) \qquad \frac{}{\Gamma \vdash \dot{n} : \texttt{int}}\,(\texttt{int} :)$$

$$\frac{\Gamma \vdash u : \sigma \to \tau \quad \Gamma \vdash v : \sigma}{\Gamma \vdash uv : \tau}\,(App :) \quad \frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \sigma \vdash v : \tau}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau}\,(Let :_{\text{simple}})$$

$$\frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash u : \tau \quad \Gamma, f : \sigma \to \tau \vdash v : \lambda}{\Gamma \vdash \texttt{letrec}\ f(x) = u\ \texttt{in}\ v : \lambda}\,(Letrec :_{\text{simple}})$$

$$\frac{\Gamma \vdash u : \texttt{int} \quad \Gamma \vdash v : \texttt{int}}{\Gamma \vdash u \dot{+} v : \texttt{int}}\,(+ :) \qquad \frac{\Gamma \vdash u : \texttt{int}}{\Gamma \vdash \dot{-} u : \texttt{int}}\,(- :)$$

$$\frac{\Gamma \vdash u : \texttt{int} \quad \Gamma \vdash v : \tau \quad \Gamma \vdash w : \tau}{\Gamma \vdash \texttt{if}\ u = 0\ \texttt{then}\ v\ \texttt{else}\ w : \tau}\,(\texttt{if} :)$$

FIGURE 6 – Le système de typage de monomorphic-pureML

```
    else if x mod 2=0
       then f (x/2)
    else f (3*x+1)
in f 3;;
```

Caml trouve tout seul que le résultat est de type `int`, avant de compiler le code, de l'exécuter, et d'imprimer le résultat. Ceci s'appelle l'*inférence de types*.

Si nous n'avions pas les schémas de types, le problème de l'inférence serait simple. Considérons l'exemple du système de typage de la figure 6, pour un langage que nous appellerons monomorphic-pureML. (C'est pureML, mais sans types polymorphes. De façon alternative, c'est PCF_V, sans annotation de types, et avec `let`. Noter que les contextes de typage sont composés d'hypothèses de la forme $x : \sigma$, où σ est un type, pas un schéma de types.)

Étant donné une expression u de monomorphic-pureML, on peut répondre à la question : « existe-t-il un contexte Γ et un type τ tels que l'on peut dériver $\Gamma \vdash u : \tau$ dans le système de la figure 6 ? », et même en temps polynomial. C'est l'algorithme de Hindley, qui forme le sujet de l'exercice 1 ci-dessous.

On y manipulera des *systèmes d'équations de types*, où chaque équation est de la forme $\sigma \doteq \tau$, où σ et τ sont des types (avec des variables de types).

Un *unificateur*, c'est-à-dire une solution, d'un système $E = \{\sigma_1 \doteq \tau_1, \cdots, \sigma_n \doteq \tau_n\}$ est une substitution θ telle que $\sigma_i\theta = \tau_i\theta$ pour tout i, $1 \leq i \leq n$. On dit aussi que θ *unifie* E. Par exemple,

$$[\alpha := (\tau \to \texttt{int}) \to \tau \to \texttt{int}, \beta := \tau \to \texttt{int}, \gamma := \tau, \delta := \texttt{int}, \epsilon := \tau \to \texttt{int}]$$

est un unificateur du système :

$$\{\alpha \doteq \beta \to \beta, \beta \doteq \gamma \to \texttt{int}, \beta \to \gamma \to \delta \doteq \epsilon \to \beta\},$$

et ce pour n'importe quel type τ. (On rappelle que $\tau_1 \to \tau_2 \to \tau_3$ signifie $\tau_1 \to (\tau_2 \to \tau_3)$.) Nous verrons plus loin que l'on peut décider de l'existence d'un unificateur (et aussi en retourner un *le plus général*, c'est-à-dire à partir duquel on peut déduire tous les autres) en temps polynomial, plus loin. Dans l'exemple ci-dessus, la substitution :

$$[\alpha := (\gamma \to \texttt{int}) \to \gamma \to \texttt{int}, \beta := \gamma \to \texttt{int}, \delta := \texttt{int}, \epsilon := \gamma \to \texttt{int}]$$

est un tel unificateur le plus général.

Exercice 1 (Algorithme de Hindley) *Une expression est* rectifiée *si et seulement si chaque variable liée n'est liée qu'à un seul endroit ; les endroits où sont liés les variables sont* $\texttt{let}\ x = u\ \texttt{in}\ v$ *(où la variable liée est* x*),* $\texttt{letrec}\ f(x) = u\ \texttt{in}\ v$ *(où les variables liées sont* f *et* x*).*

Pour une expression rectifiée donnée u_0*, l'algorithme d'inférence de types de Hindley (pour monomorphic-pureML) commence par créer une variable de type fraîche* α_t *pour chaque sous-terme* t *de* u_0 *(y compris* u_0 *lui-même et chaque variable). Pour chaque sous-terme, on crée les équations suivantes :*

- *pour les variables, rien ;*
- *pour les constantes* $\dot{n}$*, l'équation* $\alpha_{\dot{n}} \doteq \texttt{int}$ *;*
- *pour les applications* uv*, l'équation* $\alpha_u \doteq \alpha_v \to \alpha_{uv}$ *;*
- *pour les expressions de la forme* $\texttt{let}\ x = u\ \texttt{in}\ v$*, les équations* $\alpha_x \doteq \alpha_u$ *et* $\alpha_{\texttt{let}\ x=u\ \texttt{in}\ v} \doteq \alpha_v$ *;*
- *pour les expressions de la forme* $\texttt{letrec}\ f(x) = u\ \texttt{in}\ v$*, les équations* $\alpha_f \doteq \alpha_x \to \alpha_u$ *et* $\alpha_{\texttt{letrec}\ f(x)=u\ \texttt{in}\ v} \doteq \alpha_v$ *;*
- *pour les expressions* $u \dot{+} v$*, les équations* $\alpha_u \doteq \texttt{int}$*,* $\alpha_v \doteq \texttt{int}$ *et* $\alpha_{u\dot{+}v} \doteq \texttt{int}$ *;*
- *pour les expressions* $\dot{-}u$*, les équations* $\alpha_u \doteq \texttt{int}$ *et* $\alpha_{\dot{-}u} \doteq \alpha_u$ *;*
- *pour les expressions* $\texttt{if}\ u = 0\ \texttt{then}\ v\texttt{else}w$*, les équations* $\alpha_u \doteq \texttt{int}$*,* $\alpha_{\texttt{if}\ u=0\ \texttt{then}\ v\texttt{else}w} \doteq \alpha_v$ *et* $\alpha_{\texttt{if}\ u=0\ \texttt{then}\ v\texttt{else}w} \doteq \alpha_v$*.*

On notera E_{u_0} *l'ensemble des équations ainsi obtenues lorsqu'on parcourt les sous-termes de* u_0*. On écrit* Γ_{u_0} *pour le contexte de typage formé des hypothèses* $x : \alpha_x$ *lorsque* x *parcourt les variables libres de* u_0*.*

Montrer que les jugements de typage $\Gamma \vdash u_0 : \tau$ *dérivables dans le système de la figure 6 sont exactement ceux de la forme* $\Gamma_{u_0}\theta, \Delta \vdash u_0 : \alpha_{u_0}\theta$*, lorsque* Δ *parcourt les contextes de typage portant sur les variables non libres dans* u_0*, et* θ *parcourt les unificateurs de* E_{u_0}*.*

En déduire que l'on peut décider en temps polynomial de l'existence d'un typage de u_0 *en monomorphic-pureML.*

L'existence d'un unificateur le plus général, que nous démontrerons plus bas, permet de plus d'obtenir, pour monomorphic-pureML, le *typage le plus général* de u_0. Ce sera $\Gamma_{u_0}\theta \vdash u_0 : \alpha_{u_0}\theta$, où θ est l'unificateur le plus général de E_{u_0}. Par exemple, si on applique l'algorithme de Hindley sur l'expression u_0 égale à :

$$\texttt{letrec}\ f(x) = x\ \texttt{in}\ \texttt{letrec}\ g(h) = h(h\dot{3})\ \texttt{in}\ gf$$

on obtiendra le système d'équations :

(pour u_0 lui-même :)	$\alpha_{u_0} \doteq \alpha_{\texttt{letrec } g(h)=h(h\dot{3}) \texttt{ in } gf} \qquad \alpha_f \doteq \alpha_x \to \alpha_x$
(pour le sous-terme $\texttt{letrec } g(h) = h(h\dot{3}) \texttt{ in } gf$:)	$\alpha_{\texttt{letrec } g(h)=h(h\dot{3}) \texttt{ in } gf} \doteq \alpha_{gf} \qquad \alpha_g \doteq \alpha_h \to \alpha_{h(h\dot{3})}$
(pour le sous-terme $h(h\dot{3})$:)	$\alpha_h \doteq \alpha_{h\dot{3}} \to \alpha_{h(h\dot{3})}$
(pour le sous-terme $h\dot{3}$:)	$\alpha_h \doteq \alpha_{\dot{3}} \to \alpha_{h\dot{3}}$
(pour le sous-terme $\dot{3}$:)	$\alpha_{\dot{3}} \doteq \texttt{int}$
(pour le sous-terme gf)	$\alpha_g \doteq \alpha_f \to \alpha_{gf}$.

Résoudre le système d'équations mène à l'unificateur (le plus général) qui à $\alpha_{\dot{3}}$, $\alpha_{h\dot{3}}$, $\alpha_{h(h\dot{3})}$, $\alpha_{\texttt{letrec } g(h)=h(h\dot{3}) \texttt{ in } gf}$, α_{gf}, et α_{u_0} associe `int`, et qui à α_f et à α_h associe $\texttt{int} \to \texttt{int}$.

On en déduit le typage le plus général $\vdash u_0 : \texttt{int}$ (note : pas de variable libre, donc contexte de typage vide).

2.6 Unification

Le problème de l'unification, plus généralement, est le suivant. On se donne d'abord une *signature* du premier ordre Σ (le concept vient de la logique du premier ordre), c'est-à-dire un ensemble d'objets appelés *symboles de fonctions* associés à un entier naturel appelé leur *arité*. Mathématiquement, une signature est donc juste une application d'un ensemble (de symboles de fonction) vers $\mathbb{N}$. On notera parfois f/n pour énoncer que f est d'arité n. Dans le cas de pureML, la signature est $\{\texttt{int}/0, \to/2\}$.

L'ensemble $\mathcal{T}(\Sigma, X)$ des *termes* du premier ordre sur un ensemble de variables X est le plus petit tel que :

— tout élément de X est dans $\mathcal{T}(\Sigma, X)$;

— si $f/n \in \Sigma$ et $t_1, \cdots, t_n \in \mathcal{T}(\Sigma, X)$ alors $f(t_1, \cdots, t_n)$ est dans $\mathcal{T}(\Sigma, X)$.

Autrement dit, les éléments de $\mathcal{T}(\Sigma, X)$ sont les arbres finis dont les variables sont des feuilles, et dont les nœuds non variables sont étiquetés par des symboles de fonction f ; si f est d'arité n, de plus, ce nœud a exactement n fils (ordonnés).

On voit que $\mathcal{T}(\{\texttt{int}/0, \to/2\}, X)$ est exactement l'ensemble des types de pureML, si X est l'ensemble des variables de types. On note pour cela bien sûr `int` plutôt que $\texttt{int}()$ (0 argument), et $\sigma \to \tau$ plutôt que $\to(\sigma, \tau)$.

Notre approche de l'unification va procéder par réécriture progressive des systèmes d'équations à résoudre, en des systèmes ayant exactement les mêmes solutions, mais plus simples. Il sera d'autre part commode de considérer les systèmes d'équations comme des *multi-ensembles*, c'est-à-dire, au choix, des listes considérées à permutation près, ou des ensembles finis dont les éléments peuvent être écrits plusieurs fois. (Formellement, un multi-ensemble d'éléments d'un ensemble T est une application m de T dans $\mathbb{N}$ telle que $m(t) = 0$ pour tous les T sauf un nombre fini : $m(t)$ est la *multiplicité* de t dans m, c'est-à-dire le nombre de fois qu'il apparaît.)

La première règle de simplification est : si dans notre système nous trouvons une équation de la forme $f(s_1, \cdots, s_m) \doteq f(t_1, \cdots, t_m)$ (avec le même f, donc le même n), on peut la remplacer par les m équations $s_1 \doteq t_1$, …, $s_m \doteq t_m$.

$$
\begin{array}{lrcl}
(\mathbf{Dec}) & (E \cup \{f(s_1, \cdots, s_m) \doteq f(t_1, \cdots, t_m)\}, \theta) & \to & (E \cup \{s_1 \doteq t_1, \cdots, s_m \doteq t_m\}, \theta) \\
(\mathbf{DecFail}) & (E \cup \{f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)\}, \theta) & \to & \mathsf{Fail} \quad \text{si } f \neq g \\
(\mathbf{Triv}) & (E \cup \{x \doteq x\}, \theta) & \to & (E, \theta) \\
(\mathbf{Bind}) & (E \cup \{x \doteq t\}, \theta) & \to & (E[x := t], \theta[x := t]) \\
 & & & \text{si } t \neq x, x \text{ pas libre dans } t \\
(\mathbf{Bind}') & (E \cup \{t \doteq x\}, \theta) & \to & (E[x := t], \theta[x := t]) \\
 & & & \text{si } t \neq x, x \text{ pas libre dans } t \\
(\mathbf{Check}) & (E \cup \{x \doteq t\}, \theta) & \to & \mathsf{Fail} \\
 & & & \text{si } t \neq x, x \text{ libre dans } t \\
(\mathbf{Check}') & (E \cup \{t \doteq x\}, \theta) & \to & \mathsf{Fail} \\
 & & & \text{si } t \neq x, x \text{ libre dans } t
\end{array}
$$

FIGURE 7 – Unification

De même, si l'on trouve une équation de la forme $f(s_1, \cdots, s_m) \doteq g(t_1, \cdots, t_n)$ avec $f \neq g$, on peut immédiatement conclure que notre système d'équations n'a pas de solution. Nous retournerons un symbole spécial Fail dans ce cas.

Ayant traité de tous ces cas, il ne nous restera que des équations ayant une variable d'un côté, disons $x \doteq t$. Il y a alors trois cas à considérer :

- le cas trivial où t est la variable x elle-même : on peut alors effacer l'équation, c'est-à-dire $x \doteq x$, directement, sans changer l'ensemble des solutions ;
- le cas où $t \neq x$, et x n'est pas libre dans t : toute solution doit remplacer x par t (puis éventuellement les variables libres dans t par d'autres termes, que nous trouverons plus tard), on peut alors remplacer x par t dans tout le système ;
- le cas où $t \neq x$, mais x est libre dans t : c'est le cas dit de l'*occurs-check*, où nous devrons de nouveau retourner Fail tout de suite. Il n'y a en effet aucune substitution θ telle que $x\theta = t\theta$, puisque $x\theta$ apparaîtra comme sous-terme strict de $t\theta$: quoi que l'on fasse, la taille de $t\theta$ sera strictement supérieure à celle de $x\theta$, ces deux termes ne pourront donc jamais être rendus égaux par une substitution quelconque θ. (On appelle *taille* $|t|$ la quantité définie par récurrence par $|x| = 1$, $|f(t_1, \cdots, t_n)| = 1 + |t_1| + \cdots + |t_n|$.)

Nous formalisons ces étapes par les règles de la figure 7. Ces règles réécrivent des couples (E, θ) formés d'un système (multi-ensemble) d'équations E, et d'une substitution (courante) θ, en d'autres couples où en le symbole spécial Fail.

Appelons *état* de la procédure d'unification tout couple (E, θ) tel qu'aucune variable de $\mathrm{dom}\ \theta$ ne soit libre dans E, ou bien le symbole Fail. Notons que si St est un état et $St \to St'$, alors St' est encore un état. Dans le cas des règles $(\mathbf{Bind})$ et $(\mathbf{Bind}')$, c'est notamment dû au fait qu'elles ne s'appliquent que si x n'est pas libre dans t, de sorte que x n'apparaît plus du tout dans $E[x := t]$.

On dira que Fail n'a *aucune solution*. Les *solutions* de (E, θ) sont les substitutions de la forme $\theta\theta'$ où θ' est un unificateur de E. (On dira que les substitutions de la forme $\theta\theta'$ pour un certain θ' sont les *instances* de θ.) Comme par définition dans un état les variables de $\mathrm{dom}\ \theta$ ne sont pas

libres dans E, il est équivalent de demander que θ' ou bien que $\theta\theta'$ soit un unificateur de E. Les solutions de (E, θ) sont des les instances de θ qui unifient E.

La correction de ces règles vient de :

Lemme 7 *Les règles de la figure 7 préservent les solutions : Si $St \to^* St'$, alors les solutions de St' sont les mêmes que celles de St.*

Démonstration. Par récurrence sur le nombre d'étapes dans la réécriture donnée de St à St'. Le cas où ce nombre d'étape est nul est trivial, et le seul cas intéressant est celui où $St \to St'$. On examine chacune des 7 règles possibles. Les seuls cas intéressants sont les 4 derniers, qui se réduisent à 2 par symétrie.

$(\mathbf{Bind})$. Supposons que $\theta\theta'$ soit une solution de $(E \cup \{x \doteq t\}, \theta)$, où, donc, θ' unifie E. Posons θ'' la restriction de θ' aux variables autres que x. On va montrer que $(\theta[x := t])\theta'' = \theta\theta'$ et que θ'' est un unificateur de $E[x := t]$:

— On a $[x := t]\theta'' = \theta'$, parce que x n'est pas libre dans t. Donc θ'' est un unificateur de $E[x := t]$.

— D'autre part, $[x := t]\theta'' = \theta'$, puisque les deux substitutions envoient les variables y autres que x vers $y\theta'' = y\theta'$, et x vers $t\theta'' = t\theta'$ (car x n'est pas libre dans t) $= x\theta'$ (car θ' unifie $x \doteq t$). Donc $(\theta[x := t])\theta'' = \theta([x := t]\theta'') = \theta\theta'$, par associativité de la composition des substitutions (lemme 1, deuxième partie).

Réciproquement, si $\theta\theta''$ est une solution de $(E[x := t], \theta[x := t])$ (donc θ'' est un unificateur de $E[x := t]$), posons $\theta' = [x := t]\theta''$. Par le lemme 1 (première partie), θ' est un unificateur de E, et par associativité $\theta\theta' = (\theta[x := t])\theta''$.

$(\mathbf{Check})$. Il suffit de montrer que $(E \cup \{x \doteq t\}, \theta)$ n'a pas de solution. S'il en avait une, $\theta\theta'$, on aurait $x\theta' = t\theta'$. Or quel que soit θ', la taille de $t\theta'$ est strictement supérieure à celle de $x\theta'$, puisque x apparaît libre dans t et que $t \neq x$. □

Lemme 8 *Les règles de la figure 7 terminent : il n'y a pas de réécriture infinie partant d'aucune état St.*

Démonstration. Notons $|t|$ la taille d'un terme t, définissons la taille d'une équations $|s \doteq t|$ comme égale à $|s| + |t|$, la taille $|E|$ d'un système d'équations comme la somme des tailles des équations qui composent E. La taille $|(E, \theta)|$ de (E, θ) vaut $|E| + 1$, et $|\mathsf{Fail}| = 0$. Toutes les règles sauf $(\mathbf{Bind})$ et $(\mathbf{Bind}')$ font décroître la taille, et ne peuvent donc être appliquées qu'un nombre fini de fois. Le problème vient des règles $(\mathbf{Bind})$ et $(\mathbf{Bind}')$, et se résout parce que ces règles font diminuer le nombre de variables libres dans E d'au moins 1. (Noter bien que x n'est pas libre dans $E[x := t]$ *parce que x n'est pas libre dans t*.)

Supposons qu'il existe une réécriture infinie par les règles de la figure 7, partant d'un état St. Clairement, $St \neq \mathsf{Fail}$. On peut donc demander une réécriture infinie pour laquelle $St = (E, \theta)$ a un nombre de variables libre dans E *minimal*, et à nombre de variables libre égal, est de *taille minimale*. Écrivons-la $St \to St' \to \cdots$. Si la première règle appliquée, de St à St', est différente de $(\mathbf{Bind})$ et de $(\mathbf{Bind}')$, alors le nombre de variables libre diminue ou reste égal, et la taille diminue strictement : on obtient ainsi une dérivation infinie (partant de St') qui contredit notre

hypothèse de minimalité. Si c'est $(\mathbf{Bind})$ ou $(\mathbf{Bind}')$, alors c'est le nombre de variables libres qui diminue strictement, contredisant la minimalité de nouveau. □

Ceci fournit donc un *algorithme* :

— Partant d'un système d'équations E, on pose $St_0 = (E, [])$;
— En utilisant une stratégie quelconque d'applications des règles de la figure 7, on produit une réécriture $St_0 \to St_1 \to St_2 \to \cdots \to St_n$, où plus aucune règle ne s'applique à St_n : l'existence d'un tel n est garantie par la terminaison (lemme 8).
— Par inspection, St_n ne peut être que de la forme Fail ou $(\emptyset, \theta)$. Dans le premier cas, E n'a aucun unificateur. Dans le second cas, E a au moins un unificateur, à savoir θ. Dans chacun des cas, l'argument est le même : E a les mêmes solutions que St_n par le lemme 7.

Cet algorithme permet donc de décider si E a un unificateur ou non.

L'algorithme nous donne bien plus que cela. Dans le cas où $St_n = (\emptyset, \theta)$, le lemme 7 nous dit que les solutions de E sont *exactement* les instances de θ. On dira que θ est un *unificateur le plus général* de E. Le nom vient des observations suivantes :

— La relation $\preceq$ entre substitutions, définie par $\theta_1 \preceq \theta_2$ si et seulement s'il existe une substitution θ' telle que $\theta_1\theta' = \theta_2$, est un préordre, c'est-à-dire une relation réflexive et transitive (pas nécessairement antisymétrique). C'est le préordre « est plus générale que ».
— Les unificateurs de E sont les substitutions de la forme $\theta\theta'$, où θ' varie : ce sont donc les substitutions *moins générales* que θ. A contraio, θ est la plus générale parmi toutes les solutions de E.

Et l'on a donc démontré :

Proposition 9 (Mgu) *Pour tout système d'équations E, soit E n'a aucun unificateur, soit il en existe un le plus général. On note* $\mathrm{mgu}(E)$ *(« most general unifier ») l'unificateur le plus général de E obtenu par les règles de la figure 7.*

On peut décider si E a un unificateur, et calculer $\mathrm{mgu}(E)$ *le cas échéant.*

Exercice 2 *On dit qu'une substitution θ est* idempotente *si et seulement si α n'est pas libre dans $\beta\theta$, pour aucun couple de variables de types α, β dans* $\mathrm{dom}\ \theta$*. Montrer que :*

— *θ est idempotente si et seulement si $\theta\theta = \theta$ (la définition usuelle de l'idempotence).*
— *Si $(E, []) \to^* (E', \theta')$ alors θ' est idempotente.*

En déduire que $\mathrm{mgu}\ (E)$*, s'il existe, est idempotent.*

On peut vérifier que l'algorithme que nous avons donné tourne en temps *au moins exponentiel*, voir l'exercice 3. On ne peut pas faire mieux en principe pour ce qui est de calculer $\mathrm{mgu}(E)$, car il existe des systèmes E dont l'unificateur le plus général est de taille exponentielle.

Malgré cela, l'algorithme n'est pas optimal : on peut en fait décider de l'existence d'un unificateur en temps polynomial. Si l'unificateur le plus général existe, on peut en retourner un dans une représentation compacte, elle-même calculable en temps polynomial.

Exercice 3 *On considère l'algorithme de la figure 7. Montrer que, si $|s| \leq m$ et $|t| \leq m$, alors $|s[x := t]| \leq m^2$. En déduire que, partant d'un système E de taille m avec n variables libres, l'algorithme de la figure 7 termine en au plus $O(m^{2^n})$ réécritures.*

$$
\begin{array}{rrcl}
(\mathbf{Dec}) & (E \cup \{f(s_1,\cdots,s_m) \doteq f(t_1,\cdots,t_m)\},\vartheta) & \to & (E \cup \{s_1 \doteq t_1,\cdots,s_m \doteq t_m\},\vartheta) \\
(\mathbf{DecFail}) & (E \cup \{f(s_1,\cdots,s_m) \doteq g(t_1,\cdots,t_n)\},\vartheta) & \to & \mathsf{Fail} \quad \text{si } f \neq g \\
(\mathbf{Triv}) & (E \cup \{x \doteq x\},\vartheta) & \to & (E,\vartheta) \\
(\mathbf{LazyRep}) & (E \cup \{x \doteq t\},\vartheta) & \to & (E \cup \{u \doteq t\},\vartheta) \\
 & & & \text{si } t \neq x, \mathit{bound}(\vartheta,x) = \mathtt{Some}\ u \\
(\mathbf{LazyRep'}) & (E \cup \{t \doteq x\},\vartheta) & \to & (E \cup \{t \doteq u\},\vartheta) \\
 & & & \text{si } t \neq x, \mathit{bound}(\vartheta,x) = \mathtt{Some}\ u \\
(\mathbf{Bind}) & (E \cup \{x \doteq t\},\vartheta) & \to & (E,\vartheta;[x := t]) \\
 & & & \text{si } t \neq x, \mathit{bound}(\vartheta,x) = \mathtt{None} \text{ et non } \mathit{occ}(\vartheta,t,x) \\
(\mathbf{Bind'}) & (E \cup \{t \doteq x\},\vartheta) & \to & (E,\vartheta;[x := t]) \\
 & & & \text{si } t \neq x, \mathit{bound}(\vartheta,x) = \mathtt{None} \text{ et non } \mathit{occ}(\vartheta,t,x) \\
(\mathbf{Check}) & (E \cup \{x \doteq t\},\vartheta) & \to & \mathsf{Fail} \\
 & & & \text{si } t \neq x, \mathit{bound}(\vartheta,x) = \mathtt{None} \text{ et } \mathit{occ}(\vartheta,t,x) \\
(\mathbf{Check'}) & (E \cup \{t \doteq x\},\vartheta) & \to & \mathsf{Fail} \\
 & & & \text{si } t \neq x, \mathit{bound}(\vartheta,x) = \mathtt{None} \text{ et } \mathit{occ}(\vartheta,t,x)
\end{array}
$$

FIGURE 8 – Unification avec formes triangulaires, en temps polynomial

Montrer qu'il termine en un nombre au moins exponentiel (en n) de réécritures, en considérant le système :

$$\alpha_1 := \alpha_2 \to \alpha_2, \alpha_2 := \alpha_3 \to \alpha_3, \cdots, \alpha_{n-1} := \alpha_n \to \alpha_n.$$

Utiliser ce même exemple pour montrer que la taille de $\mathrm{mgu}(E)$ *est, dans le pire des cas, de taille au moins exponentielle en* $|E|$.

Exercice 4 *On va modifier l'algorithme d'unification pour en obtenir un qui tourne en temps polynomial. Ceci implique pour commencer de travailler avec des substitutions θ en* forme triangulaire *(l'appellation vient de la méthode d'élimination de Gauss dans les systèmes linéaires), c'est-à-dire sous forme d'une liste de substitutions élémentaires :*

$$\vartheta = [x_1 := t_1]; [x_2 := t_2]; \cdots; [x_n := t_n]$$

avec x_i non libre dans aucun t_j avec $j \geq i$, et représentant leur composition : mais on ne calcule pas la composition $[\vartheta] = [x_1 := t_1][x_2 := t_2]\cdots[x_n := t_n]$ explicitement, pour éviter le problème de l'exercice 3. La deuxième astuce est que l'on ne remplacera pas effectivement x par t dans les règles (**Bind**) *ou* (**Bind'**).

- *Écrire un programme $\mathit{bound}(\vartheta,x)$ qui décide, en temps polynomial, si $x \in \{x_1,\cdots,x_n\}$, où ϑ est comme ci-dessus ; si $x = x_i$, $\mathit{bound}(\vartheta,x)$ retourne* `Some` t_i*, et si $x \notin \{x_1,\cdots,x_n\}$, $\mathit{bound}(\vartheta,x)$ retourne* `None`.
- *Écrire un programme $\mathit{occ}(\vartheta,t,x)$ qui décide, en temps polynomial, si x est libre dans $t[\vartheta]$. (Noter que ceci implique de ne* pas *calculer la composition $[\vartheta]$ explicitement.)*

— *Montrer que l'algorithme de la figure 8 termine en un nombre d'étapes de réécriture polynomial, comme suit. Montrer que si* $(E, [x_1 := t_1]; \cdots; [x_m := t_m]) \to^* (E', [x_1 := t_1]; \cdots; [x_m := t_m]; \cdots; [x_n := t_n])$*, alors la taille de* E'*, ainsi que celle des termes* t_{m+1}*, …,* t_n*, est majorée par une fonction simple (et polynomiale) de* $|E|$*,* $|t_1|$*, …,* $|t_m|$*. En déduire que l'on ne peut appliquer les règles autres que les 4 dernières qu'un nombre polynomial de fois à la suite. Conclure.*

— *Montrer que l'algorithme de la figure 8 est correct, au sens où, si* $St \to^* St'$*, alors* St' *et* St *ont les mêmes solutions. On dira que les solutions d'un état* (E, ϑ) *sont celles de* $(E, [\vartheta])$.

2.7 Inférence de types, l'algorithme de Damas-Milner

Nous allons voir que nous pouvons aussi inférer les types (pas les jugements de types $\Gamma \vdash u : \tau$ tout entiers, seulement τ) automatiquement pour pureML. Le défi ici est la gestion de la quantification universelle dans les règles $(Let :)$ et $(Letrec :)$. (Voir la figure 1.)

L'algorithme est dû à Damas et Milner [1], et s'exécute en temps exponentiel. C'est incompressible : Mairson a démontré [2], ainsi que Kfoury, Tiuryn et Urzyczyn [3], que la typabilité de programmes (pure)ML était **EXPTIME**-complète.

L'algorithme prend un contexte de typage Γ, une expression pureML u, et retourne un type τ et une substitution θ tels que l'on peut typer $\Gamma\theta \vdash u : \tau\theta$. De plus, τ et θ sont les plus généraux possibles, dans un sens similaire à celui de la relation « est plus générale que » des substitutions.

Il sera plus pratique, même si c'est un peu plus éloigné des implémentations réelles, de raisonner avec des systèmes d'équations de types E plutôt que des substitutions. On va donc retourner un type τ et un système d'équations E tel que $\Gamma\theta \vdash u : \tau$ pour tout unificateur θ de E.

On notera qu'on peut passer de E à son unificateur le plus général $\theta = \mathrm{mgu}(E)$, dès que E a une solution. Réciproquement, si $\theta = [\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]$ est idempotente (voir l'exercice 2), alors on peut la convertir en un système d'équations, $\{\alpha_1 \doteq \tau_1, \cdots, \alpha_n \doteq \tau_n\}$, dont θ est l'unificateur le plus général.

L'algorithme fonctionne à peu près comme celui de Hindley. Pour la règle $(App :)$, par exemple :

$$\frac{\Gamma \vdash u : \sigma \to \tau \quad \Gamma \vdash v : \sigma}{\Gamma \vdash uv : \tau}\ (App :)$$

on va, connaissant Γ, u et v, chercher à typer u : nous obtenons un système d'équations E_1 et un type τ_1 ; puis nous cherchons à typer v, obtenant ainsi un système d'équations E_2 et un type τ_2 ; il ne reste plus qu'à créer une variable de type fraîche α (c'est-à-dire n'apparaissant nulle part dans Γ, E_1, τ_1, E_2, τ_2), et à retourner :

— le système d'équations $E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}$;

— le type α.

(On peut bien sûr optimiser un peu, si τ_1 est déjà un type flèche, pour ne pas avoir à créer de variable de type fraîche.)

Le cas de la règle $(Let :)$ (et de même, de $(Letrec :)$) est plus difficile :

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \forall \vec{\alpha} \cdot \sigma \vdash v : \tau}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau}\ (Let :)$$

où $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$.

Connaissant Γ, on est tenté de :

1. chercher à typer u, obtenant un système d'équations E_1 et un type τ_1 ;
2. calculer $\theta = \mathrm{mgu}(E_1)$ (et échouer si celui-ci n'existe pas), et en déduire l'ensemble $\vec{\alpha}$: $\vec{\alpha} = \mathrm{ftv}(\tau_1\theta) \setminus \mathrm{ftv}(\Gamma\theta)$;
3. chercher à typer v dans le nouveau contexte $\Gamma\theta, x : \forall \vec{\alpha} \cdot \tau_1\theta$, et obtenir un système d'équations E_2 et un type τ_2 ;
4. retourner $E_1 \cup E_2$, et le type τ_2.

Ceci a l'air raisonnable : le type de u cherché est $\sigma = \tau_1\theta$, par définition, et $\vec{\alpha} = \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma\theta)$ est le plus grand ensemble de variables sur lequel on peut généraliser. On n'a plus qu'à typer v dans le contexte $\Gamma, x : \forall \vec{\alpha} \cdot \sigma$.

Ceci est exactement ce que nous allons faire ! Mais la démonstration que ceci est correct se heurte à deux difficultés :

— une mineure : nous devons montrer que choisir pour $\vec{\alpha}$ l'ensemble le plus grand possible, $\mathrm{ftv}(\tau_1\theta) \setminus \mathrm{ftv}(\Gamma\theta)$, est ce que nous pouvons faire de plus général ;

— une majeure : le type σ que nous cherchons pour u n'est *pas* $\tau_1\theta$, mais une instance $\tau_1\theta\theta'$ de $\tau_1\theta$, où θ' sera découvert lors des unifications que nous ferons ultérieurement (pour les super-termes de $\texttt{let}\ x = u\ \texttt{in}\ v$, dont $\texttt{let}\ x = u\ \texttt{in}\ v$ n'est qu'un sous-terme). En principe, il faudrait attendre de finir le typage du programme contenant $\texttt{let}\ x = u\ \texttt{in}\ v$ tout entier pour connaître $\sigma\ldots$ mais on ne pourra le connaître que si on a choisir $\vec{\alpha}$. Le miracle est que le choix de $\vec{\alpha} = \mathrm{ftv}(\tau_1\theta) \setminus \mathrm{ftv}(\Gamma\theta)$ est *effectivement* le plus général que nous puissions faire.

Pour résoudre ces problèmes, nous utiliserons le lemme suivant. Disons qu'un schéma de type $\forall \vec{\alpha} \cdot \sigma$ est *plus général* que $\forall \vec{\alpha}' \cdot \sigma'$ si et seulement si on peut obtenir le second à partir du premier en instanciant le premier en un type $\sigma\theta$, avec $\mathrm{dom}\ \theta \subseteq \vec{\alpha}$, puis en re-quantifiant universellement certaines des variables libres de $\sigma\theta$, du moment qu'elles n'étaient pas libres déjà dans $\forall \vec{\alpha} \cdot \sigma$.

Par exemple, on a aura $\forall \alpha \cdot \alpha \to \beta$ plus général que $\forall \cdot \texttt{int} \to \beta$ (on instancie α par $\texttt{int}$), ou que $\forall \alpha_1, \alpha_2 \cdot (\alpha_1 \to \alpha_2) \to \beta$ (on instancie α par $\alpha_1 \to \alpha_2$, et on quantifie sur α_1 et α_2), ou que $\forall \alpha_1 \cdot (\alpha_1 \to \beta) \to \beta$ (similaire, mais on n'a pas le droit de quantifier sur β, qui était déjà libre dans $\forall \alpha \cdot \alpha \to \beta$). Mais $\forall \alpha \cdot \alpha \to \beta$ n'est pas plus général que $\forall \beta \cdot \beta \to \beta$ par exemple : la variable libre β représente un type que nous ne connaissons pas encore, et pas un type arbitraire sur lequel on pourrait quantifier.

Lemme 10 *Si $\Gamma, x : \forall \vec{\alpha}' \cdot \sigma' \vdash u : \tau$ est dérivable dans le système de types de pureML, et $\forall \vec{\alpha} \cdot \sigma$ est un schéma de type plus général que $\forall \vec{\alpha}' \cdot \sigma'$, alors $\Gamma, x : \forall \vec{\alpha} \cdot \sigma \vdash u : \tau$ est aussi dérivable dans le système de types de pureML, par une dérivation de même taille.*

Démonstration. Par récurrence sur la dérivation donnée de $\Gamma, x : \forall\vec{\alpha}' \cdot \sigma' \vdash u : \tau$. La démonstration, pénible, est reléguée à l'annexe B. □

Il est temps de définir formellement l'algorithme d'inférence de type de pureML. (Cet algorithme s'appelle traditionnellement W, et nous en définissons une variante que nous appelons W_0.) On pose :

$$\mathrm{gen}(\Gamma, \sigma) = \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma).$$

On aura aussi besoin de créer des variables de types fraîches. En pratique, on crée des variables fraîches à l'aide d'un compteur qui s'incrémente de 1 en 1 à chaque création de variable. Pour faciliter les démonstrations, nous supposerons simplement que les variables de types sont des mots finis (ou un ensemble en bijection) sur un alphabet Σ contenant les lettres `l` (left), `r` (right), `f` (fresh). D'autres codages sont possibles.

Nous paramétrerons nos fonctions par un mot p, qui sera le préfixe imposé de toutes les variables (fraîches) que nous créerons. On pose par exemple :

$$\mathrm{inst}_p(\forall\vec{\alpha} \cdot \sigma) = \varrho$$

où ϱ est le renommage de $\vec{\alpha} = \{\alpha_1, \cdots, \alpha_n\}$ qui à chaque α_i associe $p\alpha_i$ (la concaténation du mot p avec le nom de la variable α_i).

Puis on définit $W_p(\Gamma; u)$, par récurrence structurelle sur u, comme un couple (E, τ) ou bien la constante Fail, comme défini dans la figure 9. Finalement, $W(\Gamma; u)$ est défini comme le couple $(\mathrm{mgu}(E), \tau)$ où $(E, \tau) = W_p(\Gamma; u)$, si E a un unificateur, et comme Fail sinon. (Ici p est un mot tel que $\mathrm{ftv}(\Gamma) \cap p\Sigma^* = \emptyset$. Autrement dit, on choisit p de sorte que les variables de types « fraîches » engendrées par inst_p notamment sont réellement fraîches, donc notamment pas libres dans le contexte Γ. En pratique, Γ sera le contexte de typage des primitives du langage, et son ensemble de variables de types libres sera vide. Cette condition ne posera donc aucun problème pratique.)

Si $W(\Gamma; u)$ ne produit pas Fail, alors il produit un couple (θ, τ). Nous verrons plus tard (en proposition 12) que $\tau\theta$ est bien un des types possibles de u. La proposition suivante exprime que ce type est plus général que tous les types de u : si u a le type τ_0 dans le contexte Γ, alors τ_0 est une instance du type trouvé $\tau\theta$.

Proposition 11 (Complétude de W_p) *Soit Γ un contexte de typage pureML, u une expression pureML et τ_0 un type.*

Si l'on peut dériver le jugement de typage $\Gamma \vdash u : \tau_0$, alors pour tout $p \in \Sigma^*$ tel que $\mathrm{ftv}(\Gamma) \cap p\Sigma^* = \emptyset$, $W_p(\Gamma; u)$ est de la forme (E, τ) et il existe une substitution θ qui :

- *est un unificateur de E,*
- *est telle que $\tau_0 = \tau\theta$,*
- *et $\mathrm{dom}\ \theta \subseteq p\Sigma^*$, autrement dit toutes les variables dans $\mathrm{dom}\ \theta$ commencent par le préfixe p.*

En particulier, τ_0 est une instance du type τ $\mathrm{mgu}(E)$.

La dernière condition, $\mathrm{dom}\ \theta \subseteq p\Sigma^*$, est une condition technique dont nous aurons besoin dans la démonstration, mais qui n'a pas d'autre intérêt.

— $W_p(\Gamma; x) = (\emptyset, \sigma\varrho)$
s'il y a une hypothèse $x : \forall\vec{\alpha} \cdot \sigma$ dans Γ, où $\varrho = \mathrm{inst}_p(\forall\vec{\alpha} \cdot \sigma)$.
— $W_p(\Gamma; \dot{n}) = (\emptyset, \mathtt{int})$.
— $W_p(\Gamma; uv) = (E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}, \alpha)$,
où $W_{p\mathtt{l}}(\Gamma; u) = (E_1, \tau_1)$,
$W_{p\mathtt{r}}(\Gamma; v) = (E_2, \tau_2)$,
et $\alpha = p\mathtt{f}$ (une variable de type fraîche).
— $W_p(\Gamma; \mathtt{let}\ x = u\ \mathtt{in}\ v) = (E_1 \cup E_2, \tau_2)$
où $W_{p\mathtt{l}}(\Gamma; u) = (E_1, \tau_1)$,
$\theta_1 = \mathrm{mgu}(E_1)$ existe,
$\vec{\alpha}_1 = \mathrm{gen}(\Gamma\theta_1, \tau_1\theta_1)$,
$W_{p\mathtt{r}}(\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1; v) = (E_2, \tau_2)$.
— $W_p(\Gamma; \mathtt{letrec}\ f(x) = u\ \mathtt{in}\ v) = (E_1 \cup \{\tau_1 \doteq \gamma\} \cup E_2, \tau_2)$
où $\beta = p\mathtt{fl}$, $\gamma = p\mathtt{fr}$ sont deux variables de types (fraîches),
$W_{p\mathtt{l}}(\Gamma, f : \forall \cdot \beta \to \gamma, x : \forall \cdot \beta; u) = (E_1, \tau_1)$,
$\theta_1 = \mathrm{mgu}(E_1 \cup \{\tau_1 \doteq \gamma\})$ existe,
$\vec{\alpha}_1 = \mathrm{gen}(\Gamma\theta_1, \beta\theta_1 \to \gamma\theta_1)$,
et $W_{p\mathtt{r}}(\Gamma\theta_1, f : \forall\vec{\alpha}_1 \cdot \beta\theta_1 \to \gamma\theta_1; v) = (E_2, \tau_2)$.
— $W_p(\Gamma; u \dot{+} v) = (E_1 \cup E_2 \cup \{\tau_1 \doteq \mathtt{int}, \tau_2 \doteq \mathtt{int}\}, \mathtt{int})$,
où $W_{p\mathtt{l}}(\Gamma; u) = (E_1, \tau_1)$ et $W_{p\mathtt{r}}(\Gamma; u) = (E_2, \tau_2)$.
— $W_p(\Gamma; \dot{-} u) = (E_1 \cup \{\tau_1 \doteq \mathtt{int}\}, \mathtt{int})$,
où $W_p(\Gamma; u) = (E_1, \tau_1)$.
— $W_p(\Gamma; \mathtt{if}\ u = 0\ \mathtt{then}\ v\ \mathtt{else}\ w) = (E_1 \cup E_2 \cup E_3 \cup \{\tau_1 \doteq \mathtt{int}, \tau_2 \doteq \tau_3\}, \tau_2)$,
où $W_{p\mathtt{l}}(\Gamma; u) = (E_1, \tau_1)$, $W_{p\mathtt{rl}}(\Gamma; u) = (E_2, \tau_2)$, et $W_{p\mathtt{rr}}(\Gamma; w) = (E_3, \tau_3)$.
— Dans tous les autres cas, $W_p(\Gamma; u) = \mathsf{Fail}$.

FIGURE 9 – L'algorithme de Damas-Milner

Démonstration. Par récurrence sur le jugement de typage. Tous les cas sont relativement évidents (et pénibles à vérifier), sauf ceux impliquant la quantification universelle, qui utilise le lemme 10 de façon cruciale. Voir l'annexe C. □

Nous démontrons maintenant une forme de réciproque.

Proposition 12 (Correction de W_p) *Soit Γ un contexte de typage pureML, u une expression pureML, et $p \in \Sigma^*$ tel que $\mathrm{ftv}(\Gamma) \cap p\Sigma^* = \emptyset$. Si $W_p(\Gamma; u) = (E, \tau) \neq \mathsf{Fail}$, alors :*

1. *pour tout unificateur θ de E, on peut dériver le jugement de typage $\Gamma\theta \vdash u : \tau\theta$;*
2. *pour tout unificateur θ le plus général de E, on peut dériver le jugement de typage $\Gamma\theta \vdash u : \tau\theta$;*
3. *si $\mathrm{mgu}(E)$ existe, alors on peut dériver le jugement de typage $\Gamma\ \mathrm{mgu}(E) \vdash u : \tau\ \mathrm{mgu}(E)$.*

Démonstration. Voir l'annexe D. □

Des propositions 12 et 11, on déduit finalement le théorème suivant. On a déjà dit que la restriction $\mathrm{ftv}(\Gamma) = \emptyset$ n'est pas bien exigeante. En ML, Γ va contenir les primitives `nil` : $\forall\alpha \cdot \alpha$ `list`, :: : $\forall\alpha \cdot \alpha * \alpha$ `list` $\to \alpha$ `list`, `size` : `string` $\to$ `int`, etc. : leurs schémas de types sont tous sans aucune variable *libre*.

Théorème 13 (Damas-Milner) *Soit Γ un contexte de typage pureML tel que $\mathrm{ftv}(\Gamma) = \emptyset$, u une expression pureML. Les jugements de typage dérivables de la forme $\Gamma \vdash u : \tau_0$ sont exactement ceux de la forme $\Gamma \vdash u : \tau\theta\theta'$, où $W(\Gamma; u) = (\theta, \tau)$ et θ' est une substitution quelconque.*

Notamment, il existe un jugement de typage dérivable de la forme $\Gamma\theta \vdash u : \tau_0$ si et seulement si $W(\Gamma; u) \neq \mathsf{Fail}$.

Démonstration. Si $W(\Gamma; u) = (\theta, \tau)$, la proposition 12 implique que $\Gamma \vdash u : \tau\theta\theta'$ est dérivable. Réciproquement, si l'on peut dériver $\Gamma \vdash u : \tau_0$, alors la proposition 11 implique que $W_p(\Gamma; u)$, où p est un mot quelconque, est de la forme (E, τ), et qu'il existe un unificateur θ'' de E tel que $\tau_0 = \tau\theta''$. Posons $\theta = \mathrm{mgu}(E)$. Par définition de θ comme unificateur le plus général, $\theta'' = \theta\theta'$ pour une certaine substitution θ', donc $\tau_0 = \tau\theta\theta'$. D'autre part, $W(\Gamma; u) = (\theta, \tau)$, par construction. □

2.8 Remarques supplémentaires

Remarque 1. Nous avons défini l'algorithme de Damas-Milner en figure 9 dans un style applicatif. Ce n'est pas du tout la façon habituelle de le présenter, qui est bien plus impérative.

La façon classique d'implémenter l'algorithme est d'abord d'effectuer une unification *destructive*. Chaque variable de type est non pas une chaîne de caractères mais un *pointeur*. Par exemple, en Caml, on pourrait définir :

```
type ML_type = T_ARR of ML_type * ML_type (* fonctions *)
             | T_INT (* entiers *)
             | T_VAR of ML_type option ref
```

Une variable `T_VAR (ref None)` est une variable non initialisée, alors qu'une variable de la forme `T_VAR (ref (Some` τ`))` est une variable de valeur le type τ.

On peut écrire une procédure d'unification comme suit (attention, je n'ai testé aucun code de cette section ! donnez-moi vos suggestions... après avoir codé `occurs_check` et défini l'exception `ML_unify_failure`) :

```
let rec ML_unify s t =
  match s, t with
  | T_ARR (s1, s2), T_ARR (t1, t2) ->
    (ML_unify s1 t1; ML_unify s2 t2) (* regle (Dec) *)
  | T_INT, T_INT -> ()               (* regle (Dec) *)
  | T_VAR (ref (Some s')), _ -> ML_unify s' t (* (LazyRep) *)
  | _, T_VAR (ref (Some t')) -> ML_unify s t' (* (LazyRep') *)
  | T_VAR (p as ref None), _ ->
    if (match t with T_VAR (q as ref None) -> p==q | false end)
       then ()                       (* regle (Triv) *)
    else if occurs_check p t
       then raise ML_unify_failure   (* regle (Check), (Check') *)
    else p := Some t                 (* regle (Bind), (Bind') *)
  | _, T_VAR _ -> ML_unify t s (* echange, pour economiser du code *)
  | _, _ -> raise ML_unify_failure   (* regle (DecFail) *)
```

Noter que c'est l'algorithme efficace de la figure 8 qui est implémenté ici, en style impératif.

L'algorithme de Damas-Milner, en style impératif (et modulo quelques fonctions auxiliaires à définir, et quelques trous à combler) est alors :

```
let rec W Gamma t =
  match t with
  | VAR x -> let (alpha_list, sigma) = find Gamma x in
             let renamed_alpha_list =
                 List.map (fun alpha -> T_VAR (ref None)) alpha_list
                 (* on cree pour chaque alpha dans la liste alpha_list
                    une variable fraiche *)
             in  (* et on remplace *)
                 subst sigma alpha_list renamed_alpha_list
  | APP (u, v) -> let alpha = T_VAR (ref None) in (* variable fraiche *)
                  let tau1 = W Gamma u in
                  let tau2 = W Gamma v in
                      (unify tau1 (T_ARR (tau2, alpha));
                       alpha)
  | LET (x, u, v) -> let tau1 = unify Gamma u in
                     let alpha_list = gen Gamma tau1 in
                     let Gamma' = add Gamma (alpha_list, tau1) in
                         unify Gamma' v
```

```
  | (* ... a completer ... *)
```

Exercice 5 *Finir le code ci-dessus.*

Remarque 2. Nous avons défini le système de types de pureML, et l'algorithme d'inférence associé, pour un ensemble de types très pauvre : juste `int` et les fonctions.

ML a un système de type plus riche : des types de base `string`, `float`, des constructeurs de type `list`, `option`, et en fait des constructions pour définir de nouveaux types, par exemple :

```
type 'a mylist = NIL
               | CONS of 'a * 'a mylist
```

Ceci ne change rien dans l'esprit, ni au théorème de Milner, ni à l'algorithme d'inférence de types : tout ce qui change est que nous avons maintenant besoin de types écrits sous forme de termes du premier ordre, de $\mathcal{T}(\Sigma, X)$ donc, pour une signature Σ plus riche que la signature $\{\texttt{int}/0, \rightarrow/2\}$ que nous avons utilisée jusqu'à présent.

Remarque 3. En revanche, d'autres extensions classiques de ML (par rapport à pureML), posent problème. C'est le cas des effets de bord : affectations, allocations, et aussi lancement d'exceptions.

Par exemple, si vous utilisez l'algorithme W ci-dessus, et étendu de la façon évidente aux nouvelles constructions du programme ML ci-dessous :

```
let r = ref (fun x -> x) in
    (r := (fun n -> n+1);
     !r "abc")
```

vous obtiendrez le type `string`, et en tout cas pas une erreur de typage. Et pourtant, ce programme va s'évaluer en Wrong (planter, en pratique) : l'appel de la fonction `!r` sur la chaîne `"abc"` va en effet tenter de lui additionner 1.

La démonstration du théorème de Milner (théorème 6) nécessite donc de façon cruciale que la sémantique n'ait pas d'effet de bord.

Concrètement, Caml va refuser le programme ci-dessus. En fait, comme dans l'algorithme W (étendu, comme suggéré plus haut), Caml va donner le type $(\alpha \rightarrow \alpha)$ `ref` à l'expression `ref (fun x -> x)`, mais ne va *pas* généraliser α, contrairement à ce que ferait l'algorithme W. L'affectation `r := (fun n -> n+1)` unifie alors α avec `int`, et l'algorithme d'inférence de Caml échoue avec le message :

```
Error: This expression has type string but an expression was
       expected of type int
```

La correction de l'algorithme W de sorte qu'il s'adapte aux langages ML, dont Caml, a fait couler beaucoup d'encre. La solution de Caml, due à Andrew Wright, est d'une simplicité biblique : la règle de typage de `let` $x = u$ `in` v est modifiée en :

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \forall \vec{\alpha} \cdot \sigma \vdash v : \tau}{\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ v : \tau}\ (Let :)$$

où :

— soit u est une P-valeur (eh oui !), et $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$;

— soit u n'est pas une P-valeur et $\vec{\alpha} = \emptyset$.

(L'unique changement est la dernière ligne. En général, on peut remplacer « P-valeur » par toute expression pureML dont on peut prouver que son évaluation n'entraîne aucune effet de bord.)

Démontrer les propriétés « well-typed programs do not go Wrong » et définir un algorithme d'inférence de types se fait : c'est la thèse de doctorat de Mads Tofte ... et c'est du travail.

Remarque 4. Notre présentation du typage de pureML est plus compliquée que nécessaire.

On aurait par exemple pu éviter la duplication des cas (dans les règles, dans les preuves), par exemple, en définissant $u \dot{+} v$ comme étant l'application d'une primitive (vue comme une simple variable) $\dot{+} : \texttt{int} \to \texttt{int} \to \texttt{int}$.

On aurait pu aussi éviter une certaine redondance entre les constructions `let` et `letrec`. L'astuce est la suivante :

— On peut enrichir le langage d'une construction de fonction non nommée, le `fun` $x \to u$ de Caml, avec la règle de typage :

$$\frac{\Gamma, x : \forall \cdot \sigma \vdash u : \tau}{\Gamma \vdash \texttt{fun}\ x \to u : \sigma \to \tau}$$

— De même, on peut enrichir le langage d'une primitive $\texttt{fix} : \forall \alpha, \beta \cdot ((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$ de calcul de points fixes de fonctionnelles.

Alors `letrec` $f(x) = u$ `in` v a exactement la même règle de typage (et la même sémantique) que `let` $f = \texttt{fix}(\texttt{fun}\ f \to \texttt{fun}\ x \to u)$ `in` v. On aurait ainsi pu se dispenser du `letrec`, et donc des cas à examiner portant sur le `letrec`, en se ramenant au cas du `let`, et d'une seule construction élémentaire supplémentaire (`fun`).

Ces constructions, par ailleurs, étaient définissables dans pureML, et ne sont donc pas véritablement des extensions. On pouvait déjà définir `fun` $x \to u$ comme étant `letrec` $g(x) = u$ `in` g, où g est un nom frais (et notamment non libre dans u). On pouvait aussi poser :

$$\texttt{fix} = \texttt{fun}\ f \to \texttt{letrec}\ g(x) = fgx\ \texttt{in}\ g$$

Remarque 5. Alan Mycroft a remarqué au début des années 1980 que l'on pouvait écrire certains type Caml comme :

```
type 'a mycroft =
  | FIN
  | CONT of 'a * ('a list) mycroft
```

qui est un type de listes formées d'éléments, de listes, de listes de listes, etc., mais que l'algorithme de typage de ML n'arrivait pas à typer des fonctions récursives sur de tels types, par exemple :

```
let rec mycroft_length t =
  match t of
  | FIN -> 0
  | CONT (a, t') -> 1 + mycroft_length t'
```

Pourtant, il semblerait que l'on puisse typer `mycroft_length` de (schéma de) type $\forall\alpha \cdot \alpha$ `mycroft` $\to$ `int`, non ? Pour `t`: α `mycroft`, `t'` sera de type α `list mycroft`, qui est un argument valide pour la fonction `mycroft_lengh`.

La raison pour laquelle l'algorithme W rejette ce programme est que la règle $(Letrec:)$ cherche à typer la fonction `mycroft_length` d'un schéma de types *monomorphe* (avec zéro variable quantifiée) avant de typer son corps, puis de généraliser :

$$\frac{\Gamma, f : \forall \cdot \sigma \to \tau, x : \forall \cdot \sigma \vdash u : \tau \quad \Gamma, f : \forall\vec{\alpha} \cdot \sigma \to \tau \vdash v : \lambda}{\Gamma \vdash \texttt{letrec}\ f(x) = u\ \texttt{in}\ v : \lambda}\ (Letrec:)$$

On voit que les quantifications dans les schémas de types pour f et x sont vides.

Alan Mycroft a ainsi proposé de remplacer cette règle par la suivante :

$$\frac{\Gamma, f : \forall\vec{\alpha} \cdot \sigma \to \tau, x : \forall \cdot \sigma \vdash u : \tau \quad \Gamma, f : \forall\vec{\alpha} \cdot \sigma \to \tau \vdash v : \lambda}{\Gamma \vdash \texttt{letrec}\ f(x) = u\ \texttt{in}\ v : \lambda}\ (Letrec+:)$$

où $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma \to \tau) \setminus \mathrm{ftv}(\Gamma)$.

Le théorème « well-typed programs do not go Wrong » est toujours valide dans cette extension, mais l'inférence de types y est indécidable. Haskell et les versions modernes de Caml autorisent ce genre de polymorphisme à condition de déclarer (en Haskell) ou d'annoter (en Caml) f avec le schéma de types $\forall\vec{\alpha} \cdot \sigma \to \tau$.

Références

[1] L. Damas, R. Milner. Principal Type-Schemes for Functional Programs. 9th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL'82), pages 207–212, ACM Press.

[2] H. G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '90), pages 382–401, ACM Press.

[3] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn ML Typability is DEXPTIME-Complete. CAAP'90, Springer Verlag Lecture Notes in Computer Science 431, pages 206–220.

A Démonstration de la proposition 5

Règle (Var)**.** Nous avons une dérivation de $E \vdash x \Rightarrow E(x)$. Nous disposons d'autre part d'une dérivation de typage de $\Gamma \vdash x : \tau$, nécessairement par la règle $(Var:)$ (par inversion). Écrivons

$x : \forall\alpha_1, \cdots, \alpha_n \cdot \sigma$ l'hypothèse portant sur x dans Γ, de sorte que τ s'écrit $\sigma[\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]$. Par hypothèse, $\triangleright E : \Gamma$, donc $\triangleright E(x) : \forall\alpha_1, \cdots, \alpha_n \cdot \sigma$ est dérivable. Par définition de cette dernière forme de jugement, ceci implique que $\triangleright E(x) : \sigma[\alpha_1 := \tau_1, \cdots, \alpha_n := \tau_n]$ est dérivable.

Règle (Cst)**.** Nous avons une dérivation de $E \vdash \dot{n} \Rightarrow \dot{n}$, et d'une dérivation de typage de $\Gamma \vdash \dot{n} : \tau$. Par inversion, la dernière règle utilisée dans cette dérivation est $(Cst :)$, $\tau = \texttt{int}$, et on a effectivement $\triangleright\dot{n} : \texttt{int}$ par $(Val : \texttt{int})$.

Règle (App)**.** Ce cas, ainsi que les deux suivants, est beaucoup plus intéressant. Nous avons une dérivation de $E \vdash uv \Rightarrow V'$, obtenue par (App) à partir de trois dérivations plus petites (donc sur lesquelles nous pourrons appliquer l'hypothèse de récurrence—y compris pour la numéro 3) de :

(1) $E \vdash u \Rightarrow \langle \texttt{rec}\, f(x) = u', E' \rangle$

(2) $E \vdash v \Rightarrow V$

(3) $E'[f \mapsto \langle \texttt{rec}\, f(x) = u', E' \rangle, x_\sigma \mapsto V] \vdash u' \Rightarrow V'$.

D'autre part, nous avons une dérivation de typage de $\Gamma \vdash uv : \tau$. Par inversion, c'est nécessairement par la règle $(App :)$, donc nous avons des dérivations (plus petites, mais ceci ne servira pas) de :

(4) $\Gamma \vdash u : \sigma \to \tau$

(5) $\Gamma \vdash v : \sigma$

pour un certain type σ. Par hypothèse de récurrence appliquée à (1) et à (4), on obtient que :

(6) $\triangleright\langle \texttt{rec}\, f(x) = u', E' \rangle : \sigma \to \tau$ est dérivable.

Par hypothèse de récurrence appliquée à (2) et à (5), on obtient que :

(7) $\triangleright V : \sigma$.

Par inversion, (6) ne peut avoir été obtenu que par la règle $(Val : Fun)$. Il existe donc un contexte de typage Γ' tel que :

(8) $\triangleright E' : \Gamma'$

(9) $\Gamma', f : \forall \cdot \sigma \to \tau, x : \forall \cdot \sigma \vdash u' : \tau$ est dérivable.

En combinant (8), (6) et (7), on obtient que :

(10) $\triangleright E'[f \mapsto \langle \texttt{rec}\, f(x) = u', E' \rangle, x_\sigma \mapsto V] : (\Gamma', f : \forall \cdot \sigma \to \tau, x : \forall \cdot \sigma)$.

Par hypothèse de récurrence appliquée à (3) et à (9), sachant (10), on obtient enfin que $\triangleright V' : \tau$ est dérivable.

Règle (Let)**.** Nous avons une dérivation de $E \vdash \texttt{let}\ x = u\ \texttt{in}\ v \Rightarrow V'$ par (Let), donc deux dérivations plus petites de :

(11) $E \vdash u \Rightarrow V$

(12) $E[x \mapsto V] \vdash v \Rightarrow V'$.

Nous avons aussi une dérivation de typage de $\Gamma \vdash \texttt{let}\ x = u\ \texttt{in}\ x : \tau$, nécessairement par $(Let :)$, donc deux dérivations de typage de :

(13) $\Gamma \vdash u : \sigma$

(14) $\Gamma, \forall\vec{\alpha} \cdot \sigma \vdash v : \tau$

où $\vec{\alpha} \subseteq \mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)$. On peut renforcer (13) en utilisant le corollaire 4 : $\Gamma \vdash u : \sigma\theta$ est dérivable pour toute substitution θ de domaine disjoint de $\mathrm{ftv}(\Gamma)$, en particulier pour toute substitution de domaine égal à $\vec{\alpha}$. Par hypothèse de récurrence appliquée à (11) et à ce renforcement de (13), on en déduit que $\triangleright V : \sigma\theta$ est dérivable. Comme θ est arbitraire, $\triangleright V : \forall\vec{\alpha} \cdot \sigma$. Combiné avec le fait que $\triangleright E : \Gamma$, on en déduit que $\triangleright E[x \mapsto V] : (\Gamma, x : \forall\vec{\alpha} \cdot \sigma)$. Par hypothèse de récurrence appliquée à ceci, (12) et (14), $\triangleright V' : \tau$ est dérivable.

Règle $(Letrec)$**.** Nous avons une dérivation de $E \vdash \texttt{letrec}\ f(x) = u\ \texttt{in}\ v \Rightarrow V$, par $(Letrec)$, donc une dérivation plus petite de :

(15) $E[f \mapsto \langle \texttt{rec}\, f(x) = u, E\rangle] \vdash v \Rightarrow V$.

Nous avons aussi une dérivation de typage de $\Gamma \vdash \texttt{letrec}\ f(x) = u\ \texttt{in}\ v : \lambda$, nécessairement par $(Letrec :)$, et donc deux dérivations de typage de :

(16) $\Gamma, f : \forall \cdot \sigma \to \tau, x : \forall \cdot \sigma \vdash u : \tau$

(17) $\Gamma, f : \forall\vec{\alpha} \cdot \sigma \to \tau \vdash v : \lambda$

et nous souhaitons montrer que $\triangleright V : \lambda$ est dérivable. Pour ceci, comme plus haut nous renforçons d'abord (16) en :

(18) $\Gamma, f : \forall \cdot \sigma\theta \to \theta, x : \forall \cdot \sigma\theta \vdash u : \tau\theta$

pour toute substitution θ de domaine inclus dans $\vec{\alpha}$. On utilise ici le Lemma 3, plus le fait que $\Gamma\theta = \Gamma\theta$, puisque $\mathrm{dom}\ \theta \subseteq \vec{\alpha}$ est disjoint de $\mathrm{ftv}(\Gamma)$. On utilise maintenant la règle $(Val : Fun)$ comme suit. D'abord, on a $\triangleright E : \Gamma$ par hypothèse ; l'autre prémisse est (18) : on en déduit une dérivation de $\triangleright\langle \texttt{rec}\, f(x) = u, E\rangle : \sigma\theta \to \tau\theta$, pour toute substitution θ de domaine inclus dans $\vec{\alpha}$. Par définition, $\triangleright\langle \texttt{rec}\, f(x) = u, E\rangle : \forall\vec{\alpha} \cdot \sigma \to \tau$.

Combiné avec $\triangleright E : \Gamma$, ceci implique $\triangleright E[f \mapsto \langle \texttt{rec}\, f(x) = u, E\rangle] : (\Gamma, f : \forall\vec{\alpha} \cdot \sigma \to \tau)$. Par (15), (17), et l'hypothèse de récurrence, on obtient ainsi que $\triangleright V : \lambda$ est dérivable.

Nous ne traitons pas des autres règles, qui ne posent pas de problème spécial. Par exemple, pour la règle $(\dot{+})$, l'hypothèse de récurrence nous donne que $\triangleright\dot{n}_1 : \texttt{int}$ et $\triangleright\dot{n}_2 : \texttt{int}$ sont dérivables, mais nous le savions déjà ; nous devons conclure que $\triangleright\dot{n} : \texttt{int}$ est dérivable pour $n = n_1 + n_2$, mais ceci découle directement de la règle $(Val : \texttt{int})$. □

B Démonstration du lemme 10

Rappelons l'énoncé à prouver : si $\Gamma, x : \forall\vec{\alpha}' \cdot \sigma' \vdash u : \tau$ est dérivable dans le système de types de pureML, et $\forall\vec{\alpha} \cdot \sigma$ est un schéma de type plus général que $\forall\vec{\alpha}' \cdot \sigma'$, alors $\Gamma, x : \forall\vec{\alpha} \cdot \sigma \vdash u : \tau$ est aussi dérivable dans le système de types de pureML, par une dérivation de même taille.

Par construction, on peut écrire $\sigma' = \sigma\theta$ pour une certaine substitution θ de domaine $\vec{\alpha}$, et $\vec{\alpha}' \subseteq \mathrm{ftv}(\sigma\theta) \setminus \mathrm{ftv}(\forall\vec{\alpha}\cdot\sigma)$. (Le point important que nous utiliserons est que $\vec{\alpha}'$ n'intersecte pas $\mathrm{ftv}(\forall\vec{\alpha}\cdot\sigma)$.) Les seuls cas intéressants sont ceux des règles $(Var\ :)$, $(Let\ :)$, et $(Letrec\ :)$.

Règle $(Var\ :)$. Ici, le seul cas intéressant où $u = x$, et τ est de la forme $\sigma'\theta'$ pour une certaine substitution θ' de domaine $\vec{\alpha}'$. Le type τ est donc aussi égal à $\sigma(\theta\theta')$ par associativité de la composition. On réécrit ceci sous la forme $\sigma\theta''$, pour une certaine substitution θ'', de sorte que $\mathrm{dom}\ \theta'' \subseteq \vec{\alpha}$, pour permettre de déduire $\Gamma, \forall\vec{\alpha}\cdot\sigma \vdash x : \sigma\theta''$ par $(Var\ :)$. Il est clair que l'on doit poser $\theta''(\alpha) = \alpha\theta\theta'$ pour chaque $\alpha \in \vec{\alpha}$. Ceci permet d'appliquer $(Var\ :)$, mais nous devons encore vérifier que $\sigma\theta'' = \sigma(\theta\theta')$, autrement dit que $\alpha\theta'' = \alpha\theta\theta'$ pour toute $\alpha \in \mathrm{ftv}(\sigma)$. Or les variables α de $\mathrm{ftv}(\sigma)$ se séparent en celles qui sont dans $\vec{\alpha}$, pour lesquelles c'est la définition de θ'', et celles qui sont dans $\mathrm{ftv}(\forall\vec{\alpha}\cdot\sigma)$. Pour ces dernières, $\alpha\theta'' = \alpha$ d'une part, et d'autre part $\alpha\theta = \alpha$ (puisque $\mathrm{dom}\ \theta \subseteq \vec{\alpha}$ et $\alpha \notin \vec{\alpha}$), donc $\alpha\theta\theta' = \alpha\theta' = \alpha$ (puisque $\mathrm{dom}\ \theta' \subseteq \vec{\alpha}'$ n'intersecte pas $\mathrm{ftv}(\forall\vec{\alpha}\cdot\sigma)$, auquel appartient α).

Règle $(Let\ :)$. Nous avons une dérivation de $\Gamma, x : \forall\vec{\alpha}'\cdot\sigma' \vdash \mathtt{let}\ x_1 = u_1\ \mathtt{in}\ v_1 : \tau$ par $(Let\ :)$, donc deux dérivations plus courtes de :

(1) $\Gamma, x : \forall\vec{\alpha}'\cdot\sigma' \vdash u_1 : \sigma_1$ et

(2) $\Gamma, x : \forall\vec{\alpha}'\cdot\sigma', x_1 : \forall\vec{\alpha}_1\cdot\sigma_1 \vdash v_1 : \tau$,

où $\vec{\alpha}_1 \subseteq \mathrm{ftv}(\sigma_1) \setminus \mathrm{ftv}(\Gamma, x : \forall\vec{\alpha}'\cdot\sigma')$. Par hypothèse de récurrence et (1), resp. (2), on a aussi deux dérivations de :

(3) $\Gamma, x : \forall\vec{\alpha}\cdot\sigma \vdash u_1 : \sigma_1$, et

(4) $\Gamma, x : \forall\vec{\alpha}\cdot\sigma, x_1 : \forall\vec{\alpha}_1\cdot\sigma_1 \vdash v_1 : \tau$.

On remarque maintenant que $\mathrm{ftv}(\forall\vec{\alpha}\cdot\sigma)$ est inclus dans $\mathrm{ftv}(\forall\vec{\alpha}'\cdot\sigma')$: en remplaçant (1) par (3), on n'a pas augmenté l'ensemble des variables de types libres dans le contexte. En effet, si α est libre dans $\forall\vec{\alpha}\cdot\sigma$, comme $\vec{\alpha}'$ n'intersecte pas $\mathrm{ftv}(\forall\vec{\alpha}\cdot\sigma)$, α n'est pas dans $\vec{\alpha}'$. D'autre part, α est libre dans $\sigma' = \sigma\theta$, puisque $\alpha \notin \vec{\alpha}$ et $\mathrm{dom}\ \theta \subseteq \vec{\alpha}$. Donc α est dans $\mathrm{ftv}(\sigma') \setminus \vec{\alpha}' = \mathrm{ftv}(\forall\vec{\alpha}'\cdot\sigma')$.

Cette remarque nous permet d'affirmer que $\mathrm{ftv}(\Gamma, x : \forall\vec{\alpha}\cdot\sigma)$ est inclus dans $\mathrm{ftv}(\Gamma, x : \forall\vec{\alpha}'\cdot\sigma')$. Comme $\vec{\alpha}_1 \subseteq \mathrm{ftv}(\sigma_1) \setminus \mathrm{ftv}(\Gamma, x : \forall\vec{\alpha}'\cdot\sigma')$, on a donc aussi $\vec{\alpha}_1 \subseteq \mathrm{ftv}(\sigma_1) \setminus \mathrm{ftv}(\Gamma, x : \forall\vec{\alpha}\cdot\sigma)$.

On peut donc appliquer $(Let\ :)$ aux prémisses (3) et (4) et en obtenir une dérivation de $\Gamma, x : \forall\vec{\alpha}\cdot\sigma \vdash \mathtt{let}\ x_1 = u_1\ \mathtt{in}\ v_1 : \tau$. Le cas de la règle $(Letrec\ :)$ se traite de façon similaire. □

C Démonstration de la proposition 11

Nous commençons par démontrer deux lemmes auxiliaires, dont nous aurons par ailleurs encore besoin plus tard.

On notera $\mathrm{ftv}(E)$ l'ensemble des variables de types libres du système d'équations de types E. Par extension, on notera $\mathrm{ftv}(\theta)$ pour une substitution θ l'union $\mathrm{dom}\ \theta \cup \bigcup_{\alpha\in\mathrm{dom}\ \theta'} \mathrm{ftv}(\alpha\theta)$: c'est l'ensemble des variables libres du système d'équations canonique $\{\alpha \doteq \alpha\theta \mid \alpha \in \mathrm{dom}\ \theta\}$ associé à θ.

Lemme 14 *Soit E un système d'équations de types. Si $\theta = \mathrm{mgu}(E)$ existe, alors* $\mathrm{ftv}(\theta) \subseteq \mathrm{ftv}(E)$.

Démonstration. Il suffit d'observer que si $(E, \theta) \to (E', \theta')$ dans la procédure de la figure 7, alors $\mathrm{ftv}(E') \cup \mathrm{ftv}(\theta') \subseteq \mathrm{ftv}(E) \cup \mathrm{ftv}(\theta)$. Pour finir, on utilise ceci dans une récurrence sur le nombre de réécritures de $(E, [])$ à $(\emptyset, \theta)$. □

Lemme 15 *Soit Γ un contexte de typage pureML, u une expression pureML. Si $W_p(\Gamma; u) = (E, \tau)$, alors les variables de types libres dans E et dans τ appartiennent à $\mathrm{ftv}(\Gamma) \cup p\Sigma^*$.*

Démonstration. Par récurrence structurelle sur u. Dans le cas des `let`, qui est l'un des deux qui ne sont pas complètement triviaux, rappelons que $W_{p\mathtt{1}}(\Gamma; u) = (E_1, \tau_1)$, $\theta_1 = \mathrm{mgu}(E_1)$ existe, $\vec{\alpha}_1 = \mathrm{gen}(\Gamma\theta_1, \tau_1\theta_1)$, $W_{p\mathtt{r}}(\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1; v) = (E_2, \tau_2)$. Par hypothèse de récurrence, $\mathrm{ftv}(E_1)$ et $\mathrm{ftv}(\tau_1)$ sont inclus dans $\mathrm{ftv}(\Gamma)$. Par le lemme 14, $\mathrm{ftv}(\theta_1) \subseteq \mathrm{ftv}(E_1) \subseteq \mathrm{ftv}(\Gamma)$. Il s'ensuit que toutes les variables libres dans $\Gamma\theta_1$ et dans $\tau_1\theta_1$, donc dans $\forall\vec{\alpha}_1 \cdot \tau_1\theta_1$, sont dans $\mathrm{ftv}(\Gamma)$. Par hypothèse de récurrence, on en déduit que $\mathrm{ftv}(E_2)$ et $\mathrm{ftv}(\tau_2)$ sont inclus dans $\mathrm{ftv}(\Gamma)$. Donc $\mathrm{ftv}(E_1 \cup E_2)$ aussi. L'autre cas non trivial, celui de `letrec`, est similaire. □

Rappelons l'énoncé à prouver. Soit Γ un contexte de typage pureML, u une expression pureML et τ_0 un type. Si l'on peut dériver le jugement de typage $\Gamma \vdash u : \tau_0$, alors pour tout $p \in \Sigma^*$ tel que $\mathrm{ftv}(\Gamma) \cap p\Sigma^* = \emptyset$, $W_p(\Gamma; u)$ est de la forme (E, τ) et il existe une substitution θ qui :

— est un unificateur de E,
— est telle que $\tau_0 = \tau\theta$,
— et $\mathrm{dom}\ \theta \subseteq p\Sigma^*$, autrement dit toutes les variables dans $\mathrm{dom}\ \theta$ commencent par le préfixe p.

En particulier, τ_0 est une instance du type $\tau\ \mathrm{mgu}(E)$.

La démonstration est par récurrence sur le jugement de typage. Les seuls cas intéressants sont les règles $(Var :)$, $(Let :)$ et $(Letrec :)$. Nous traitons aussi de $(App :)$, qui nous permettra d'apprécier la simplification apportée par le codage des variables fraîches via l'opération de concaténation avec le préfixe p.

Règle $(Var :)$**.** On peut dériver $\Gamma \vdash x : \tau_0$ par $(Var :)$, donc on a une hypothèse $x : \forall\vec{\alpha} \cdot \sigma$ dans Γ, et une substitution θ_1 de domaine $\vec{\alpha}$ telle que $\tau_0 = \sigma\theta_1$. Alors $W_0(\Gamma; x) = (\emptyset, \sigma\varrho)$ où $\varrho = \mathrm{inst}_p(\forall\vec{\alpha} \cdot \sigma)$. On pose $\theta = \varrho^{-1}\theta_1$.

Règle $(App :)$**.** On peut dériver $\Gamma \vdash uv : \tau_0$, à partir de deux dérivations plus courtes, l'une de $\Gamma \vdash u : \sigma \to \tau_0$, l'autre de $\Gamma \vdash v : \sigma$. Par hypothèse de récurrence, $W_{p\mathtt{1}}(\Gamma; u)$ est de la forme (E_1, τ_1), et il existe une substitution θ_1 de domaine inclus dans $p\mathtt{1}\Sigma^*$ qui unifie E_1 et telle que $\sigma \to \tau_0 = \tau_1\theta_1$. De même, $W_{p\mathtt{r}}(\Gamma; v)$ est de la forme (E_1, τ_2), et il existe une substitution θ_2 de domaine inclus dans $p\mathtt{r}\Sigma^*$ qui unifie E_2 et telle que $\sigma = \tau_2\theta_2$. Comme $\mathrm{dom}\ \theta_1 \subseteq p\mathtt{1}\Sigma^*$ et $\mathrm{dom}\ \theta_2 \subseteq p\mathtt{r}\Sigma^*$, les domaines de θ_1 et de θ_2 sont disjoints, et ne contiennent pas $p\mathtt{f}$, donc il est sensé de définir la substitution $\theta = \theta_1 \uplus \theta_2 \uplus [p\mathtt{f} := \tau_0]$ qui à tout $\beta \in \mathrm{dom}\ \theta_1$ associe $\beta\theta_1$, à tout $\beta \in \mathrm{dom}\ \theta_2$ associe $\beta\theta_2$, et à $p\mathtt{f}$ associe τ_0. Son domaine est inclus dans $p\mathtt{1}\Sigma^* \cup p\mathtt{r}\Sigma^* \cup \{p\mathtt{f}\} \subseteq p\Sigma^*$, θ unifie à la fois E_1, E_2 et $\tau_1 \doteq \tau_2 \to \alpha$ (car $\tau_1\theta = \tau_1\theta_1 = \sigma \to \tau_0$, $\tau_2\theta = \tau_2\theta_2 = \sigma$, et $\alpha\theta = p\mathtt{f}\theta = \tau_0$; on utilise le lemme 15 pour démontrer $\tau_1\theta = \tau_1\theta_1$ et $\tau_2\theta = \tau_2\theta_2$). Finalement, $\alpha\theta$ est bien égal à τ_0, comme souhaité.

Règle $(Let :)$**.** On peut dériver $\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ v : \tau_0$, à partir de deux dérivations plus courtes, l'une de $\Gamma \vdash u : \sigma$, l'autre de $\Gamma, x : \forall\vec{\alpha} \cdot \sigma \vdash v : \tau_0$, où $\vec{\alpha} \subseteq \mathrm{gen}(\Gamma, \sigma)$. Par hypothèse de récurrence, $W_{p\mathtt{1}}(\Gamma; u)$ est de la forme (E_1, τ_1), et il existe une substitution θ'_1 de domaine

inclus dans $p\mathtt{l}\Sigma^*$ qui unifie E_1 et telle que $\tau_1\theta_1' = \sigma$. En particulier, E_1 a un unificateur, donc un unificateur le plus général $\theta_1 = \mathrm{mgu}(E_1)$. Posons, comme dans la définition de W_p sur le cas des `let`, $\vec{\alpha}_1 = \mathrm{gen}(\Gamma\theta_1, \tau_1\theta_1)$.

Comme θ_1 est l'unificateur le plus général de E_1 et θ_1' est un autre unificateur, θ_1' est une instance de θ_1, c'est-à-dire que l'on peut écrire θ_1' comme $\theta_1\theta_1''$ pour une certaine substitution θ_1''. On en déduit que $\mathrm{dom}\ \theta_1 \subseteq \mathrm{dom}\ \theta_1' \subseteq p\mathtt{l}\Sigma^*$. L'hypothèse $\mathrm{ftv}(\Gamma) \cap p\Sigma^*$ implique alors que : (a) $\Gamma\theta_1 = \Gamma$.

Nous prétendons que : (b) $\forall\vec{\alpha}_1 \cdot \tau_1\theta_1$ est un schéma de type plus général que $\forall\vec{\alpha} \cdot \sigma$, c'est-à-dire que $\forall\vec{\alpha} \cdot \tau_1\theta_1'$. Une partie est facile : $\sigma = \tau_1\theta_1' = \tau_1\theta_1\theta_1''$. Il reste à vérifier qu'aucune variable de $\vec{\alpha}$ n'est libre dans $\forall\vec{\alpha}_1 \cdot \tau_1\theta_1$. Pour ceci, on remarque que $\vec{\alpha}_1 \supseteq \mathrm{ftv}(\sigma) \cap \vec{\alpha}_1 = \mathrm{ftv}(\sigma) \cap \mathrm{gen}(\Gamma\theta_1, \tau_1\theta_1) = \mathrm{ftv}(\sigma) \cap (\mathrm{ftv}(\tau_1\theta_1) \setminus \mathrm{ftv}(\Gamma\theta_1)) = \mathrm{ftv}(\sigma) \cap (\mathrm{ftv}(\tau_1\theta_1) \setminus \mathrm{ftv}(\Gamma))$ (par (a)) $= (\mathrm{ftv}(\sigma) \setminus \mathrm{ftv}(\Gamma)) \cap \mathrm{ftv}(\tau_1\theta_1) = \mathrm{gen}(\Gamma, \sigma) \cap \mathrm{ftv}(\tau_1\theta_1) \supseteq \vec{\alpha} \cap \mathrm{ftv}(\tau_1\theta_1)$. En résumé, $\vec{\alpha} \cap \mathrm{ftv}(\tau_1\theta_1) \subseteq \vec{\alpha}_1$. Si α est une variable de $\vec{\alpha}$ qui est libre dans $\forall\vec{\alpha}_1 \cdot \tau_1\theta_1$, elle est libre dans $\tau_1\theta_1$ et hors de l'ensemble $\vec{\alpha}_1$. Mais, étant à la fois dans $\vec{\alpha}$ et dans $\mathrm{ftv}(\tau_1\theta_1)$, nous venons de démontrer qu'elle serait dans $\vec{\alpha}_1$, contradiction.

C'est ici que, de façon cruciale, **nous utilisons le lemme 10.** Avec (b) et le fait que nous avons une dérivation de $\Gamma, x : \forall\vec{\alpha} \cdot \sigma \vdash v : \tau_0$, ceci nous donne une dérivation de $\Gamma, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1 \vdash v : \tau_0$ de même taille, donc de nouveau plus courte que la dérivation originale de $\Gamma \vdash \mathtt{let}\ x = u\ \mathtt{in}\ v : \tau_0$. Par (a), c'est aussi une dérivation de $\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1 \vdash v : \tau_0$.

En appliquant l'hypothèse de récurrence à cette dernière, on obtient que $W_{p\mathtt{r}}(\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1; v)$ est de la forme (E_2, τ_2), et qu'il existe un unificateur θ_2 de E_2 de domaine inclus dans $p\mathtt{r}\Sigma^*$ tel que $\tau_0 = \tau_2\theta_2$.

Par un raisonnement similaire à celui que nous avons effectué dans le cas $(App :)$, la substitution $\theta = \theta_1 \uplus \theta_2$ est alors un unificateur de $E_1 \cup E_2$ de domaine inclus dans $p\Sigma^*$, et $\tau_2\theta = \tau_0$.

Le cas de la règle $(Letrec :)$ est similaire.

Finalement, la dernière affirmation de la proposition, que τ_0 est une instance du type $\tau\ \mathrm{mgu}(E)$, se démontre à partir des premières comme suit. Soit θ un unificateur de E tel que $\tau_0 = \tau\theta$. Comme $\mathrm{mgu}(E)$ est l'unificateur le plus général de E, on peut écrire θ sous la forme $\mathrm{mgu}(E)\theta'$ pour une certaine substitution θ'. Donc $\tau_0 = \tau\ \mathrm{mgu}(E)\theta'$. □

D Démonstration de la proposition 12

Nous avons besoin d'un lemme auxiliaire de plus, dont nous n'utiliserons en fait que la moitié, à savoir que $\theta_1\theta_2$ est un unificateur de $E_1 \cup E_2$; on n'aura pas besoin de savoir qu'il est plus général.

Lemme 16 *Soient E_1, E_2 deux systèmes d'équations de types, supposons que $\theta_1 = \mathrm{mgu}(E_1)$ existe, et que $\mathrm{ftv}(E_2) \cap \mathrm{dom}\ \theta_1 = \emptyset$. Si $\theta_2 = \mathrm{mgu}(E_2)$ existe, alors $\theta_1\theta_2$ est un unificateur le plus général de $E_1 \cup E_2$.*

Démonstration. D'abord, c'est un unificateur : pour chaque équation $\sigma \doteq \tau$ de E_1, on a $\sigma\theta_1 = \tau\theta_1$, donc $\sigma\theta_1\theta_2 = \tau\theta_1\theta_2$; pour chaque équation $\sigma \doteq \tau$ de E_2, on a $\sigma\theta_1 = \sigma$ et $\tau\theta_1 = \tau$ puisque $\mathrm{ftv}(E_2) \cap \mathrm{dom}\ \theta_1 = \emptyset$, donc $\sigma\theta_1\theta_2 = \sigma\theta_2 = \tau\theta_2 = \tau\theta_1\theta_2$.

Si θ est un unificateur de $E_1 \cup E_2$, il unifie E_1, et s'écrit donc $\theta_1\theta'_1$ pour une certaine substitution θ'_1, puisque θ_1 est l'unificateur de E_1 le plus général. Mais, pour chaque équation $\sigma \doteq \tau$ de E_2, on a, comme plus haut $\sigma\theta_1 = \sigma$ et $\tau\theta_1 = \tau$, donc $\sigma\theta_1\theta'_1 = \sigma\theta'_1 = \tau\theta'_1 = \tau\theta_1\theta'_1$. Donc θ'_1 unifie E_2. Il s'ensuit que θ'_1 s'écrit $\theta_2\theta'_2$ pour une certaine substitution θ'_2. Donc $\theta = \theta_1\theta_2\theta'_2$ est moins général que $\theta_1\theta_2$. □

Rappelons l'énoncé. Soit Γ un contexte de typage pureML, u une expression pureML, et $p \in \Sigma^*$ tel que $\mathrm{ftv}(\Gamma) \cap p\Sigma^* = \emptyset$. Si $W_p(\Gamma; u) = (E, \tau) \neq \mathsf{Fail}$, alors :

1. pour tout unificateur θ de E, on peut dériver le jugement de typage $\Gamma\theta \vdash u : \tau\theta$;
2. pour tout unificateur θ le plus général de E, on peut dériver le jugement de typage $\Gamma\theta \vdash u : \tau\theta$;
3. si $\mathrm{mgu}(E)$ existe, alors on peut dériver le jugement de typage $\Gamma\ \mathrm{mgu}(E) \vdash u : \tau\ \mathrm{mgu}(E)$.

Avant de passer à la démonstration proprement dite, on remarque que les trois affirmations 1, 2, 3 sont équivalentes : 1 implique 2, 2 implique 3, et finalement si 3 est vrai, alors tout unificateur θ de E est de la forme $\mathrm{mgu}(E)\theta'$ pour une certaine substitution θ', et on en déduit 1 par le lemme 3.

On démontre maintenant 1, ou 2, ou 3, de façon équivalente, par récurrence sur la structure de u. Nous ne regarderons que les cas des variables, des applications, et des `let`, qui sont les plus importants ou les plus typiques.

Variables. Soit x une variable. Si $W_p(\Gamma; x) = (E, \tau)$, c'est que Γ contient une hypothèse de la forme $x : \forall\vec{\alpha} \cdot \sigma$, $E = \emptyset$, et $\tau = \sigma\varrho$ où $\varrho = \mathrm{inst}_p(\forall\vec{\alpha} \cdot \sigma)$. On démontre 1. On a $\mathrm{mgu}(E) = []$ (la substitution vide), et l'on obtient une dérivation de typage de $\Gamma \vdash x : \tau$ par $(Var :)$.

Applications. Supposons que $W_p(\Gamma; uv) = (E, \tau)$. Par définition, $W_{p\mathtt{1}}(\Gamma; u) = (E_1, \tau_1)$, $W_{p\mathtt{r}}(\Gamma; v) = (E_2, \tau_2)$, $E = E_1 \cup E_2 \cup \{\tau_1 \doteq \tau_2 \to \alpha\}$ et $\tau = \alpha$, où $\alpha = p\mathtt{f}$. On démontre 2. Soit θ un unificateur de E. Comme θ unifie en particulier E_1, on obtient par hypothèse de récurrence une dérivation de $\Gamma\theta \vdash u : \tau_1\theta$. Comme θ unifie $\tau_1 \doteq \tau_2 \to \alpha\}$ et que $\tau = \alpha$, c'est aussi une dérivation de $\Gamma\theta \vdash u : \tau_2\theta \to \tau\theta$. Comme θ unifie E_2, l'hypothèse de récurrence nous donne une dérivation de $\Gamma\theta \vdash v : \tau_2\theta$. De ces deux dernières dérivations, on obtient une dérivation de $\Gamma\theta \vdash uv : \tau\theta$.

Expressions `let`. C'est un peu plus subtil. On aura besoin des lemmes précédents, et de réaliser que le mgu retourné par les règles de la figure 7 est **idempotent** (voir l'exercice 2).

Supposons que $W_p(\Gamma; \mathtt{let}\ x = u\ \mathtt{in}\ v) = (E, \tau)$. Par définition, $E = E_1 \cup E_2$, $\tau = \tau_2$, où $W_{p\mathtt{1}}(\Gamma; u) = (E_1, \tau_1)$, $\theta_1 = \mathrm{mgu}(E_1)$ existe, $\vec{\alpha}_1 = \mathrm{gen}(\Gamma\theta_1, \tau_1\theta_1)$, $W_{p\mathtt{r}}(\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1; v) = (E_2, \tau_2)$.

Soit $\theta_2 = \mathrm{mgu}(E_2)$, qui existe puisque $E_1 \cup E_2$, donc E_2, a un unificateur. Aucune variable de $\mathrm{dom}\ \theta_1$ n'est libre dans $\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1$, puisque θ_1 est idempotent. (L'application de la substitution a remplacé toutes les variables de $\mathrm{dom}\ \theta_1$ par des variables qui ne sont pas dans $\mathrm{dom}\ \theta_1$.) Par le lemme 15, $\mathrm{ftv}(E_2)$ est inclus dans $\mathrm{ftv}(\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1) \cup p\mathtt{r}\Sigma^*$, et est donc d'intersection vide avec $\mathrm{dom}\ \theta_1$. On peut donc appliquer le lemme 14 et conclure que $\theta_1\theta_2$ est un unificateur le plus général de $E_1 \cup E_2$. (Qu'il soit le plus général n'est pas très important.)

Posons $\theta = \theta_1\theta_2$. On démontre 2. Par hypothèse de récurrence (variante 1), on a une dérivation de $\Gamma\theta \vdash u : \tau_1\theta$. De même, on a une dérivation de $\Gamma\theta_1\theta_2, x : (\forall\vec{\alpha}_1 \cdot \tau_1\theta_1)\theta_2 \vdash v : \tau_2\theta_2$. Ce n'est

pas exactement la forme que l'on souhaite : pour appliquer $(Let :)$, il nous faudrait une dérivation de $\Gamma\theta, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta \vdash v : \tau_2\theta$. Or :

— $\Gamma\theta_1\theta_2 = \Gamma\theta$, par définition de θ.

— $\mathrm{ftv}(\theta_2) \subseteq \mathrm{ftv}(E_2)$ par le lemme 14. Ceci implique que $\vec{\alpha}_1$ est d'intersection vide avec $\mathrm{ftv}(\theta_2)$. En effet, une variable α dans l'intersection serait dans $\mathrm{ftv}(E_2)$, donc dans $\mathrm{ftv}(\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1) \cup p\mathtt{r}\Sigma^*$. Mais elle ne peut pas être libre dans $\Gamma\theta_1$ puisque $\vec{\alpha} = \mathrm{ftv}(\tau_1\theta_1) \setminus \mathrm{ftv}(\Gamma\theta_1)$, elle ne peut pas être libre dans $\forall\vec{\alpha}_1 \cdot \tau_1\theta_1$ qui précisément est quantifiée sur $\vec{\alpha}_1$. Il ne reste qu'à montrer qu'elle ne peut pas être dans $p\mathtt{r}\Sigma^*$: c'est parce que $\alpha \in \vec{\alpha}_1$ est dans $\mathrm{ftv}(\tau_1\theta_1) \subseteq \mathrm{ftv}(\Gamma) \cup p\mathtt{l}\Sigma^*$ par le lemme 14 et le lemme 15, et par hypothèse $\mathrm{ftv}(\Gamma)$ n'intersecte pas $p\Sigma^*$, donc pas non plus $p\mathtt{r}\Sigma^*$.
Comme $\vec{\alpha}_1$ est d'intersection vide avec $\mathrm{ftv}(\theta_2)$, on obtient immédiatement que $(\forall\vec{\alpha}_1 \cdot \tau_1\theta_1)\theta_2 = \forall\vec{\alpha}_1 \cdot \tau_1\theta_1\theta_2 = \forall\vec{\alpha}_1 \cdot \tau_1\theta$.

— Par le lemme 15, les variables de types libres dans τ_2 sont dans $\mathrm{ftv}\Gamma\theta_1, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta_1) \cup p\mathtt{r}\Sigma^*$, qui par des arguments similaires n'intersecte pas $\mathrm{dom}\ \theta_1$. Donc $\tau_2\theta = \tau_2\theta_1\theta_2 = \tau_2\theta_2$.

Notre dérivation de $\Gamma\theta_1\theta_2, x : (\forall\vec{\alpha}_1 \cdot \tau_1\theta_1)\theta_2 \vdash v : \tau_2\theta_2$ est donc *exactement* la dérivation souhaitée de $\Gamma\theta, x : \forall\vec{\alpha}_1 \cdot \tau_1\theta \vdash v : \tau_2\theta$.

De plus, $\vec{\alpha}_1 \subseteq \mathrm{ftv}(\tau_1\theta) \setminus \mathrm{ftv}(\Gamma\theta)$, puisque les variables de $\vec{\alpha}_1$ sont libres dans $\tau_1\theta_1$, et restent libres dans $\tau_1\theta_1\theta_2 = \tau_1\theta$ (rappelons que $\vec{\alpha}_1 \cap \mathrm{ftv}(\theta_2) = \emptyset$, en particulier aucune variable de $\vec{\alpha}_1$ n'est dans $\mathrm{dom}\ \theta_2$) ; et parce que si une variable de $\vec{\alpha}_1$ était libre dans $\Gamma\theta = \Gamma\theta_1\theta_2$, comme elle n'est pas dans $\mathrm{ftv}(\theta_2)$, elle serait libre dans $\Gamma\theta_1$.

Donc la règle $(Let :)$ s'applique, et l'on peut dériver $\Gamma\theta \vdash \mathtt{let}\ x = u\ \mathtt{in}\ v : \tau_2\theta$. □