Chapter 3

Inférence de types

3.1 Introduction à l'inférence de types

Pour un langage statiquement typé, un *typeur* est un algorithme qui prend un programme en entrée et détermine si ce programme est typable ou non. En général, il va aussi déterminer un type τ pour ce programme.

Si plusieurs types τ sont possibles, on essaye de renvoyer comme résultat du typeur un *type principal*, c'est-à-dire un type plus précis que tous les autres types possibles (pour une notion de "plus précis" qui dépend du système de types considéré). Par exemple, en mini-ML, `fun` $x \rightarrow x$ a les types $\tau \rightarrow \tau$ pour n'importe quel type τ, mais le type $\alpha \rightarrow \alpha$ est principal, puisque tous les autres types possibles s'en déduisent par substitution.

La quantité de travail fournie par le typeur est inversement proportionnelle à la quantité de déclarations de types présentes dans le langage source, et donc à la quantité d'information de typage écrite par le programmeur:

Vérification pure: Dans le source, toutes les sous-expressions du programme, ainsi que tous les identificateurs, sont annotés par leur type.

```
fun (x:int) →
  (let (y:int) = (+:int×int→int)((x:int),(1:int):int×int) in (y:int) : int)
```

Le typeur est alors très simple, puisque le programme source contient déjà autant d'informations de typage que la dérivation de typage correspondante. La patience du programmeur est mise à rude épreuve par la quantité d'annotations de types à fournir. Aucun langage réaliste ne suit cette approche.

Déclaration des types des identificateurs et propagation des types: Le programmeur déclare les types des paramètres de fonctions et des variables locales. Le typeur infère les types des sous-expressions en "propageant" les types des feuilles de l'expression vers la racine. Par exemple, sachant que `x` est de type `int`, le typeur peut non seulement vérifier que l'expression `x+1` est bien typée, mais aussi inférer qu'elle a le type `int`. L'exemple ci-dessus devient:

```
fun (x:int) → let (y:int) = +(x,1) in y
```

Le typeur infère le type `int` $\rightarrow$ `int` pour cette expression. Cette approche est suivie par la plupart des langages impératifs: Pascal, C, Java, ... (En fait, ces langages exigent un peu plus d'annotations de types; par exemple, il faut aussi déclarer le type du résultat des fonctions.)

Déclaration des types des paramètres de fonction et propagation des types: La seule différence par-rapport à l'approche précédente est que les variables locales (p.ex. les identificateurs liés par `let`) ne sont pas annotées par leur type, ce dernier étant déterminé par le type de l'expression liée à la variable. Exemple:

```
fun (x:int) → let y = +(x,1) in y
```

Ayant déterminé que `+(x,1)` est de type `int`, le typeur déduit que `y` est de type `int` dans le reste de la fonction.

Inférence complète de types: Le programme source ne contient aucune déclaration de type sur les paramètres de fonctions ni sur les variables locales. Le typeur détermine le type des paramètres de fonctions d'après l'utilisation qui en est faite dans le reste du programme. Exemple:

```
fun x → let y = +(x,1) in y
```

Puisque l'addition `+` n'opère que sur des paires d'entiers, `x` est forcément de type `int`. C'est l'approche suivie dans les langages de la famille ML.

Dans ce cours, nous allons nous concentrer sur la dernière approche (inférence complète), car les autres sont techniquement très simples. On pourra faire l'exercice suivant pour se familiariser avec l'avant-dernière approche.

Exercice de programmation 3.1 *Écrire un typeur pour mini-ML avec typage monomorphe (section 1.3.2) et dans lequel les paramètres de fonctions sont annotés dans le programme source par leur type:*

Expressions: $a ::= \ldots \mid \texttt{fun}\ (x : \tau) \rightarrow a$

Il s'agit donc de la troisième approche dans la liste ci-dessus. On écrira le typeur sous la forme d'une fonction prenant une expression a et un environnement de typage E en arguments, et renvoyant un type τ pour a dans E s'il existe, ou bien échoue (en levant une exception) si a n'est pas typable dans E.

Quels problèmes se posent si l'on veut étendre ce typeur à mini-ML avec typage polymorphe (en gardant les paramètres de fonctions annotés par leurs types)?

3.2 Inférence de types pour mini-ML avec typage monomorphe

Dans cette section, nous considérons le problème de l'inférence de types pour mini-ML muni du système de typage monomorphe de la section 1.3.2. Nous allons procéder en deux temps:

1. À partir du programme source, on construit un système d'équations entre types qui caractérise tous les typages possibles pour ce programme.

2. On résout ensuite ce système d'équations. S'il n'y a pas de solution, le programme est mal typé. Sinon, on détermine une solution principale au système d'équation; cela nous donne le type principal du programme.

En combinant ces deux phases, on obtient un algorithme de typage qui détermine si un programme est bien typé et si oui, calcule son type principal.

3.2.1 Construction du système d'équations

On se donne un programme (une expression close) a_0 dans laquelle tous les identificateurs liés par `fun` ou `let` ont des noms différents. À chaque identificateur x dans a_0, on associe une variable de type α_x. De même, à chaque sous-expression a de a_0, on associe une variable de type α_a.

Le système d'équations $C(a_0)$ associé à a_0 est construit en parcourant l'expression a_0 et en ajoutant des équations pour chaque sous-expression a de a_0, comme suit:

- Si a est une variable x: $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x\}$.
- Si a est une constante c ou un opérateur op: $C(a) = \{\alpha_a \stackrel{?}{=} TC(c)\}$ ou $C(a) = \{\alpha_a \stackrel{?}{=} TC(op)\}$.
- Si a est $\texttt{fun}\ x \to b$: $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x \to \alpha_b\} \cup C(b)$.
- Si a est une application $b\ c$: $C(a) = \{\alpha_b \stackrel{?}{=} \alpha_c \to \alpha_a\} \cup C(b) \cup C(c)$.
- Si a est une paire (b, c): $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_b \times \alpha_c\} \cup C(b) \cup C(c)$.
- Si a est $\texttt{let}\ x = b\ \texttt{in}\ c$: $C(a) = \{\alpha_x \stackrel{?}{=} \alpha_b;\ \alpha_a \stackrel{?}{=} \alpha_c\} \cup C(b) \cup C(c)$.

Exemple: on considère le programme

$$a = \underbrace{(\texttt{fun}\ x \to \underbrace{\texttt{fun}\ y \to \underbrace{1}_{d}}_{c})}_{b}\ \underbrace{\texttt{true}}_{e}$$

On a:

$$\begin{aligned} C(a) = \{\ & \alpha_b \stackrel{?}{=} \alpha_e \to \alpha_a; \\ & \alpha_b \stackrel{?}{=} \alpha_x \to \alpha_c; \\ & \alpha_c \stackrel{?}{=} \alpha_y \to \alpha_d; \\ & \alpha_d \stackrel{?}{=} \texttt{int}; \\ & \alpha_e \stackrel{?}{=} \texttt{bool}\} \end{aligned}$$

Exercice de programmation 3.2 *Écrire une fonction qui prend une expression a en entrée et construit son ensemble d'équations $C(a)$. Pour associer les variables α_b, α_x aux sous-expressions b et aux identificateurs x, on pourra ou bien utiliser une table d'association globale (table de hachage ou autre), ou bien écrire une première passe sur a qui annote les identificateurs et les sous-expressions par des variables de types.*

3.2.2 Lien entre typages et solutions des équations

Une *solution* de l'ensemble d'équations $C(a)$ est une substitution φ telle que pour toute équation $\tau_1 \stackrel{?}{=} \tau_2$ dans $C(a)$, on ait $\varphi(\tau_1) = \varphi(\tau_2)$. Autrement dit, une solution est un unificateur de l'ensemble d'équations $C(a)$.

Les deux propositions suivantes montrent que les solutions de $C(a)$ caractérisent exactement les typages possibles pour a.

Proposition 3.1 (Correction des solutions vis-à-vis du typage) *Si φ est une solution de $C(a)$, alors $E \vdash a : \varphi(\alpha_a)$ où E est l'environnement de typage $\{x : \varphi(\alpha_x) \mid x \text{ libre dans } a\}$.*

Démonstration: par récurrence structurelle sur a. Les cas de base $a = x$, $a = c$ et $a = op$ sont immédiats.

Pour le cas $a = \texttt{fun}\ x \to b$, on a $C(a) = \{\alpha_a \stackrel{?}{=} \alpha_x \to \alpha_b\} \cup C(b)$. Comme φ est une solution de $C(a)$, c'est aussi une solution de $C(b)$, et de plus $\varphi(\alpha_a) = \varphi(\alpha_x) \to \varphi(\alpha_b)$. Appliquant l'hypothèse de récurrence à b, il vient $E + \{x : \varphi(\alpha_x)\} \vdash b : \varphi(\alpha_b)$. On peut donc construire la dérivation suivante:

$$\frac{E + \{x : \varphi(\alpha_x)\} \vdash b : \varphi(\alpha_b)}{E \vdash \texttt{fun}\ x \to b : \varphi(\alpha_x) \to \varphi(\alpha_b)}$$

c'est-à-dire $E \vdash a : \varphi(\alpha_a)$, comme attendu.

Pour le cas $a = b\ c$, on a $C(a) = \{\alpha_b \stackrel{?}{=} \alpha_c \to \alpha_a\} \cup C(b) \cup C(c)$. Donc, φ est une solution de $C(b)$ et de $C(c)$. Appliquant deux fois l'hypothèse de récurrence, il vient $E \vdash b : \varphi(\alpha_b)$ et $E \vdash c : \varphi(\alpha_c)$. Comme $\varphi(\alpha_b) = \varphi(\alpha_c) \to \varphi(\alpha_a)$, on a la dérivation suivante:

$$\frac{E \vdash b : \varphi(\alpha_c) \to \varphi(\alpha_a) \qquad E \vdash c : \varphi(\alpha_c)}{E \vdash b\ c : \varphi(\alpha_a)}$$

C'est le résultat attendu. Les cas restants sont tout aussi simples. □

Proposition 3.2 (Complétude des solutions vis-à-vis du typage) *Soit a une expression. S'il existe un environnement E et un type τ tels que $E \vdash a : \tau$, alors le système d'équations $C(a)$ admet une solution.*

Démonstration: on construit une solution φ par récurrence sur la dérivation de $E \vdash a : \tau$ et par cas sur la dernière règle de typage utilisée. Cette solution φ vérifie de plus les propriétés suivantes:

1. $\varphi(\alpha_a) = \tau$
2. $\varphi(\alpha_x) = E(x)$ pour tout $x \in \mathrm{Dom}(E)$
3. le domaine de φ est α_x pour $x \in \mathrm{Dom}(E)$ et α_b pour toute sous-expression b de a.

On montre deux cas représentatifs de la preuve; les autres sont similaires.

Cas la dérivation se termine par la règle (fun).

$$\frac{E + \{x : \tau_1\} \vdash b : \tau_2}{E \vdash \texttt{fun}\ x \to b : \tau_1 \to \tau_2}$$

On a donc $a = \texttt{fun}\ x \to b$ et $\tau = \tau_1 \to \tau_2$. Par application de l'hypothèse de récurrence, il existe une solution φ' de $C(b)$ vérifiant de plus (1), (2) et (3). On prend $\varphi = \varphi' + [\alpha_a \leftarrow \tau]$. Il est facile de voir que φ est une solution de $C(a) = C(b) \cup \{\alpha_a \stackrel{?}{=} \alpha_x \to \alpha_b\}$. En effet,

$$\varphi(\alpha_a) = \tau = \tau_1 \to \tau_2 = \varphi(\alpha_x) \to \varphi(\alpha_b)$$

et d'autre part φ est solution de $C(b)$ puisque φ prolonge φ'. Enfin, les propriétés (1), (2) et (3) sont vérifiées pour φ.

Cas la dérivation se termine par la règle (app).

$$\frac{E \vdash b : \tau' \to \tau \qquad E \vdash c : \tau'}{E \vdash b\ c : \tau}$$

On a donc $a = b\ c$. En appliquant deux fois l'hypothèse de récurrence à b et c, il vient des solutions φ_1 et φ_2 de $C(b)$ et $C(c)$ qui satisfont les propriétés (1)–(3). Par la propriété (2), il vient $\varphi_1(\alpha_x) = \varphi_2(\alpha_x)$ pour tout $x \in \mathrm{Dom}(E)$. On peut donc prendre $\varphi = \varphi_1 + \varphi_2 + [\alpha_a \leftarrow \tau]$. C'est une substitution qui prolonge φ_1 et φ_2. Donc, φ est solution de $C(b)$ et $C(c)$. Enfin,

$$\varphi(\alpha_b) = \varphi_1(\alpha_b) = \tau' \to \tau = \varphi_2(\alpha_c) \to \varphi(\alpha_a) = \varphi(\alpha_c) \to \varphi(\alpha_a).$$

Donc, φ est solution de $C(a)$, et de plus vérifie (1)–(3). CQFD. □

3.2.3 Résolution des équations

L'ensemble $C(a)$ peut être vu comme un problème d'unification du premier ordre: c'est un ensemble d'équations entre équations de types, qui sont des termes du premier ordre construits sur la signature $T \cup \{\to, \times\}$. Il existe donc un algorithme `mgu` qui, étant donné un ensemble d'équations C, a l'un des deux comportements suivants:

- $\texttt{mgu}(C)$ échoue, signifiant que C n'a pas de solution.
- $\texttt{mgu}(C)$ renvoie une substitution φ (un unificateur principal de C) qui est une solution de C, et de plus telle que toute autre solution ψ de C peut s'écrire $\psi = \theta \circ \varphi$ pour une certaine substitution θ.

(Voir le cours de J.-P. Jouannaud pour plus de détails.) Un algorithme qui convient est l'algorithme d'unification de Robinson, que nous rappelons ci-dessous.

$$\begin{array}{rcl}
\texttt{mgu}(\emptyset) & = & id \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \alpha\} \cup C) & = & \texttt{mgu}(C) \\
\texttt{mgu}(\{\alpha \stackrel{?}{=} \tau\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\
\texttt{mgu}(\{\tau \stackrel{?}{=} \alpha\} \cup C) & = & \texttt{mgu}(C[\alpha \leftarrow \tau]) \circ [\alpha \leftarrow \tau] \text{ si } \alpha \text{ n'est pas libre dans } \tau \\
\texttt{mgu}(\{\tau_1 \to \tau_2 \stackrel{?}{=} \tau_1' \to \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C) \\
\texttt{mgu}(\{\tau_1 \times \tau_2 \stackrel{?}{=} \tau_1' \times \tau_2'\} \cup C) & = & \texttt{mgu}(\{\tau_1 \stackrel{?}{=} \tau_1'; \tau_2 \stackrel{?}{=} \tau_2'\} \cup C)
\end{array}$$

Dans tous les autres cas, $\texttt{mgu}(C)$ échoue et C n'a pas de solutions.

Exemple: on considère le jeu d'équations obtenus dans l'exemple de la section 3.2.1. La solution principale est

$$\begin{array}{ll} \alpha_x \leftarrow \texttt{bool} & \alpha_e \leftarrow \texttt{bool} \\ \alpha_a \leftarrow \alpha_y \rightarrow \texttt{int} & \alpha_c \leftarrow \alpha_y \rightarrow \texttt{int} \\ \alpha_d \leftarrow \texttt{int} & \end{array}$$

Les autres solutions s'en déduisent en remplaçant α_y par un type quelconque.

Exercice de programmation 3.3 *Implémenter la fonction* `mgu`.

3.2.4 L'algorithme d'inférence

En recollant les morceaux, on obtient l'algorithme d'inférence de types $I(a)$ suivant:
Entrée: une expression close a.
Sortie: ou bien `err`, ou bien un type τ.
Calcul: calculer $\varphi = \texttt{mgu}(C(a))$. Si `mgu` échoue, renvoyer `err`. Sinon, renvoyer $\varphi(\alpha_a)$.

Proposition 3.3 (Propriétés de l'algorithme I)

1. *Correction: si $I(a)$ est un type τ, alors $\emptyset \vdash a : \tau$.*
2. *Complétude: si $I(a)$ est* `err`*, alors a n'est pas typable dans $\emptyset$.*
3. *Principalité du type inféré: s'il existe un type τ' tel que $\emptyset \vdash a : \tau'$, alors $I(a)$ n'est pas* `err`*; au contraire, c'est un type τ, et de plus il existe une substitution θ telle que $\tau' = \theta(\tau)$.*

Démonstration: (1) est conséquence du lemme 3.1 et de la correction de l'algorithme `mgu`: le résultat de $\texttt{mgu}(C(a))$, s'il existe, est une solution de $C(a)$.

Pour (3), supposons $\emptyset \vdash a : \tau'$. Par le lemme 3.2, cela signifie qu'il existe une solution φ' de $C(a)$ avec de plus $\tau' = \varphi'(\alpha_a)$. Puisque $C(a)$ admet une solution, $\varphi = \texttt{mgu}(C(a))$ existe, et de plus $\varphi' = \theta \circ \varphi$ pour une certaine substitution θ (par principalité de l'unificateur calculé par `mgu`). Donc, $\tau' = \varphi'(\alpha_a) = \theta(\varphi(\alpha_a)) = \theta(\tau')$.

Pour (2), la contraposée de (2), "si a est typable, alors $I(a) \neq \texttt{err}$", est un corollaire immédiat de (3). □

Exercice de programmation 3.4 *Implémenter l'algorithme I.*

3.3 Inférence de types pour mini-ML avec typage polymorphe

L'approche de la section 3.2 ne s'étend pas facilement au typage polymorphe de la section 1.3.4. En effet, les contraintes de typages $C(a)$ devraient contenir non seulement des équations d'unification entre types, mais aussi des contraintes d'instanciation (pour les règles (var-inst), (const-inst) et (op-inst)) et de généralisation (pour la règle (let-gen)):

- Si a est une variable x: $C(a) = \{\alpha_a \leq \alpha_x\}$.
- Si a est `let` $x = b$ `in` c: $C(a) = \{\alpha_x \stackrel{?}{=} \texttt{Gen}(\alpha_b, E);\ \alpha_a \stackrel{?}{=} \alpha_c\} \cup C(b) \cup C(c)$
  où E est l'environnement $\{y : \alpha_y\}$ pour tout y lié à l'endroit où apparaît l'expression `let`.

Remarquons que les α_x sont maintenant des schémas et non plus des types simples.

La résolution des contraintes $C(a)$ est maintenant beaucoup plus difficile qu'un problème d'unification du premier ordre. En particulier, on ne peut plus résoudre les contraintes d'unification et les contraintes de généralisation dans n'importe quel ordre.

Exemple: on considère

$$a = \texttt{let}\ x = \underbrace{\texttt{fun}\ y \to \underbrace{y}_{c}}_{b}\ \texttt{in}\ x\ 1$$

On obtient les contraintes suivantes (entre autres):

$$\begin{aligned} \alpha_x &= \texttt{Gen}(\alpha_b, \emptyset) \\ \alpha_b &= \alpha_y \to \alpha_c \\ \alpha_c &= \alpha_y \end{aligned}$$

Si l'on résout la première immédiatement, on obtient $\alpha_x = \forall \alpha_b.\alpha_b$, ce qui n'est clairement pas correct puisque b a forcément un type fonctionnel. Il faut donc avoir résolu au préalable les contraintes sur α_b et α_c avant de calculer α_x.

Pour résoudre ce problème, nous allons voir un algorithme qui entremêle construction de contraintes et résolution de ces contraintes par unification, en une seule passe sur le programme d'entrée.

3.3.1 L'algorithme W de Damas-Milner-Tofte

On commence par définir la notion d'instance triviale $\texttt{Inst}(\sigma, V)$ d'un schéma de types σ par-rapport à un ensemble de variables "nouvelles" V:

$$\texttt{Inst}(\forall \alpha_1, \ldots, \alpha_n.\tau,\ V) = (\tau[\alpha_i \leftarrow \beta_i],\ V \setminus \{\beta_1 \ldots \beta_n\})$$

où $\beta_1, \ldots, \beta_n$ sont n variables distinctes choisies dans V.

L'algorithme W est alors le suivant:

Entrée: un environnement de typage E, une expression a, et un ensemble de variables V.

Sortie: une erreur ou bien un triplet (τ, φ, V').
τ est le type inféré pour l'expression a.
φ est la substitution à effectuer sur les variables libres de E afin que a soit typable dans E.
V' est V privé des variables "nouvelles" que W a utilisées.

Calcul:

- Si a est une variable x avec $x \in \mathrm{Dom}(E)$:
  prendre $(\tau, V') = \texttt{Inst}(E(x), V)$ et $\varphi = id$.

- Si a est une constante c ou un opérateur op:
  prendre $(\tau, V') = \texttt{Inst}(TC(a), V)$ et $\varphi = id$.

- Si a est $\texttt{fun } x \to a_1$:
soit α une nouvelle variable prise dans V
soit $(\tau_1, \varphi_1, V_1) = W(E + \{x : \alpha\}, a_1, V \setminus \{\alpha\})$
prendre $\tau = \varphi_1(\alpha) \to \tau_1$ et $\varphi = \varphi_1$ et $V' = V_1$.

- Si a est une application $a_1\ a_2$:
soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
soit α une nouvelle variable prise dans V_2
soit $\mu = \texttt{mgu}\{\varphi_2(\tau_1) \stackrel{?}{=} \tau_2 \to \alpha\}$
prendre $\tau = \mu(\alpha)$ et $\varphi = \mu \circ \varphi_2 \circ \varphi_1$ et $V' = V_2 \setminus \{\alpha\}$.

- Si a est une paire (a_1, a_2):
soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E), a_2, V_1)$
prendre $\tau = \varphi_2(\tau_1) \times \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$.

- Si a est $\texttt{let } x = b \texttt{ in } c$:
soit $(\tau_1, \varphi_1, V_1) = W(E, a_1, V)$
soit $(\tau_2, \varphi_2, V_2) = W(\varphi_1(E) + \{x : \texttt{Gen}(\tau_1, \varphi_1(E))\}, a_2, V_1)$
prendre $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$.

- Dans tous les cas non couverts par les cas ci-dessus, et en particulier si l'un des appels à `mgu` échoue, ou si V est vide lorsqu'on veut prendre une nouvelle variable dedans, on prend $W(E, a, V) = \texttt{err}$.

Exemple on considère $a = \texttt{fun } x \to +(x, 1)$. Le déroulement de $W(\emptyset, a, V)$ est le suivant:

$$\begin{array}{rcl}
W(\{x : \alpha\}, +, V \setminus \{\alpha\}) & = & (\texttt{int} \times \texttt{int} \to \texttt{int}, id, V \setminus \{\alpha\}) \\
W(\{x : \alpha\}, x, V \setminus \{\alpha\}) & = & (\alpha, id, V \setminus \{\alpha\}) \\
W(\{x : \alpha\}, 1, V \setminus \{\alpha\}) & = & (\texttt{int}, id, V \setminus \{\alpha\}) \\
W(\{x : \alpha\}, (x, 1), V \setminus \{\alpha\}) & = & (\alpha \times \texttt{int}, id, V \setminus \{\alpha\}) \\
W(\{x : \alpha\}, +(x, 1), V \setminus \{\alpha\}) & = & (\texttt{int}, [\alpha \leftarrow \texttt{int}, \beta \leftarrow \texttt{int}], V \setminus \{\alpha, \beta\}) \\
W(\emptyset, a, V) & = & (\texttt{int} \to \texttt{int}, [\alpha \leftarrow \texttt{int}, \beta \leftarrow \texttt{int}], V \setminus \{\alpha, \beta\})
\end{array}$$

Exercice de programmation 3.5 *Implémenter l'algorithme W et le faire tourner sur des exemples. (On pourra se dispenser du paramètre V et du résultat V', et à la place générer des variables "nouvelles" en utilisant un compteur.)*

3.3.2 Propriétés de l'algorithme W

Théorème 3.1 (Correction de l'algorithme W) *Si $W(E, a, V) = (\tau, \varphi, V')$, alors on peut dériver $\varphi(E) \vdash a : \tau$.*

La preuve utilise le lemme technique suivant:

Proposition 3.4 (Commutation entre Gen et substitution) *On dit qu'une variable α est hors de portée d'une substitution φ si $\varphi(\alpha) = \alpha$ et pour tout $\beta \neq \alpha$, α n'est pas libre dans $\varphi(\beta)$. Soit alors un environnement E, un type τ et une substitution φ tels que les variables généralisables $\mathcal{L}(\tau) \setminus \mathcal{L}(E)$ sont toutes hors de portée de φ. Alors, $\mathtt{Gen}(\varphi(\tau), \varphi(E)) = \varphi(\mathtt{Gen}(\tau, E))$.*

Démonstration: on remarque qu'une variable α hors de portée de φ est libre dans un type $\varphi(\tau)$ si et seulement si elle est libre dans le type d'origine τ. Il s'ensuit $\mathcal{L}(\varphi(\tau)) \setminus \mathcal{L}(\varphi(E)) = \mathcal{L}(\tau) \setminus \mathcal{L}(E)$, puis le résultat annoncé. □

Démonstration: (du théorème 3.1) par récurrence structurelle sur a. La preuve utilise de manière essentielle la stabilité du typage par substitution (proposition 1.2). On donne un cas de base, et deux cas qui utilisent l'hypothèse de récurrence; les autres cas sont similaires. On reprend les notations de l'algorithme.

Cas $a = x$ avec $x \in \mathrm{Dom}(E)$. On a $(\tau, V') = \mathtt{Inst}(E(x), V)$ et $\varphi = id$. Par définition de `Inst`, on a $\tau \leq E(x)$. On peut donc bien dériver $E \vdash x : \tau$ par la règle (inst-var).

Cas $a = a_1\ a_2$. Appliquant l'hypothèse de récurrence aux deux appels récursifs de `W`, on obtient des dérivations de

$$\varphi_1(E) \vdash a_1 : \tau_1 \qquad \text{et} \qquad \varphi_2(\varphi_1(E)) \vdash a_2 : \tau_2.$$

On applique la substitution $\mu \circ \varphi_2$ à la dérivation de gauche, et μ à celle de droite. Par la proposition 1.2, il vient:

$$\varphi(E) \vdash a_1 : \mu(\varphi_2(\tau_1)) \qquad \text{et} \qquad \varphi(E) \vdash a_2 : \mu(\tau_2).$$

Comme μ est un unificateur de $\{\varphi_2(\tau_1) \stackrel{?}{=} \tau_2 \to \alpha\}$, on a $\mu(\varphi_2(\tau_1)) = \mu(\tau_2) \to \mu(\alpha)$. On peut donc dériver par la règle (app)

$$\varphi(E) \vdash a_1\ a_2 : \mu(\alpha)$$

C'est le résultat attendu.

Cas $a = \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$. On applique l'hypothèse de récurrence aux deux appels récursifs de W. Il vient des preuves de

$$\varphi_1(E) \vdash a_1 : \tau_1 \qquad \text{et} \qquad \varphi_2(\varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\} \vdash a_2 : \tau_2.$$

Si nécessaire, on renomme dans la dérivation de gauche les variables généralisées pour qu'elles soient hors de portée de φ_2. On a alors par le lemme 3.4

$$\mathtt{Gen}(\varphi_2(\tau_1), \varphi_2(\varphi_1(E))) = \varphi_2(\mathtt{Gen}(\tau_1, \varphi_1(E)))$$

Notant $\varphi = \varphi_2 \circ \varphi_1$, on a donc des preuves de:

$$\varphi(E) \vdash a_1 : \varphi_2(\tau_1) \qquad \text{et} \qquad \varphi(E) + \{x : \mathtt{Gen}(\varphi_2(\tau_1), \varphi(E))\} \vdash a_2 : \tau_2.$$

On conclut, par la règle (let-gen),

$$\varphi(E) \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau_2.$$

C'est le résultat annoncé. □

Définition: étant données deux substitutions φ et ψ et un ensemble de variables V, on dit que $\varphi = \psi$ *hors de* V si $\varphi(\alpha) = \psi(\alpha)$ pour toute variable $\alpha \notin V$. On voit facilement que si $\mathcal{L}(\tau) \cap V = \emptyset$ et si $\varphi = \psi$ hors de V, alors $\varphi(\tau) = \psi(\tau)$.

Théorème 3.2 (Complétude et principalité de l'algorithme W) *Soit V un ensemble de variables infini et tel que $V \cap \mathcal{L}(E) = \emptyset$. S'il existe un type τ' et une substitution φ' tels que $\varphi'(E) \vdash a : \tau'$, alors $W(E, a, V)$ n'est pas* `err`*; au contraire, il existe τ, φ, V' et une substitution θ tels que*

$$W(E,a,V) = (\tau, \varphi, V') \qquad \textit{et} \qquad \tau' = \theta(\tau) \qquad \textit{et} \qquad \varphi' = \theta \circ \varphi \textit{ hors de } V.$$

Démonstration: on commence par remarquer que, avec les hypothèses de la proposition, si $(\tau, \varphi, V') = W(a, E, V)$ est défini, alors $V' \subseteq V$, V' est infini, et les variables de V' ne sont pas libres dans τ et sont hors de portée de φ. En conséquence, $V' \cap \mathcal{L}(\varphi(E)) = \emptyset$.

La preuve du théorème 3.2 est par récurrence structurelle sur a. On donne un cas de base et trois cas de récurrence; les autres cas sont similaires.

Cas $a = x$. Puisque $\varphi(E) \vdash x : \tau$, on a $x \in \mathrm{Dom}(\varphi(E))$ et $\tau \leq \varphi(E)(x)$. Ceci entraîne $x \in \mathrm{Dom}(E)$. Écrivons $E(x) = \forall \alpha_1 \ldots \alpha_n . \tau_x$, avec les α_i choisies dans V' et hors de portée de φ'. Nous avons donc que $W(E, x, V)$ est défini et renvoie

$$\tau = \tau_x[\alpha_1 \leftarrow \beta_1, \ldots, \alpha_n \leftarrow \beta_n] \qquad \text{et} \qquad \varphi = id \qquad \text{et} \qquad V' = V \setminus \{\beta_1 \ldots \beta_n\}$$

pour certaines variables $\beta_1 \ldots \beta_n \in V$. Par choix des α_i, nous avons $\varphi'(E(x)) = \forall \alpha_1 \ldots \alpha_n . \varphi'(\tau_x)$. On note ρ la substitution sur les α_i telle que $\tau' = \rho(\varphi'(\tau_x))$. On prend

$$\psi = \rho \circ \varphi' \circ [\beta_1 \leftarrow \alpha_1, \ldots, \beta_n \leftarrow \alpha_n].$$

On a $\psi(\tau) = \rho(\varphi'(\tau_x)) = \tau'$. D'autre part, toute variable $\alpha \notin V$ n'est ni une des α_i, ni une des β_i, d'où $\psi(\alpha) = \rho(\varphi'(\alpha)) = \varphi'(\alpha)$. C'est le résultat annoncé, puisque $\varphi = id$ ici.

Cas $a = \texttt{fun}\ x \to a_1$. La dérivation initiale se termine par

$$\frac{\varphi'(E) + \{x \leftarrow \tau_2'\} \vdash a_1 : \tau_1'}{\varphi'(E) \vdash \texttt{fun}\ x \to a_1 : \tau_2' \to \tau_1'}$$

Prenons α dans V, comme dans l'algorithme. On définit l'environnement E_1 et la substitution φ_1' par

$$E_1 = E + \{x \leftarrow \alpha\} \qquad \text{et} \qquad \varphi_1' = \varphi' + \{\alpha \leftarrow \tau_2'\}.$$

On a $\varphi_1'(E_1) = \varphi'(E) + \{x \leftarrow \tau_2'\}$. On applique l'hypothèse de récurrence à a_1, E_1, $V \setminus \{\alpha\}$, φ_1' et τ_2'. Il vient

$$(\tau_1, \varphi_1, V_1) = W(a_1, E_1, V \setminus \{\alpha\}) \qquad \text{et} \qquad \tau_1' = \psi_1(\tau_1) \qquad \text{et} \qquad \varphi_1' = \psi_1 \circ \varphi_1 \text{ hors de } V \setminus \{\alpha\}.$$

Il s'ensuit que $W(E, a, V)$ est bien défini. On prend alors $\psi = \psi_1$. Montrons que cette substitution ψ convient. On a:

$$\begin{array}{rcll}
\psi(\tau) & = & \psi(\varphi_1(\alpha) \to \tau_1) & \text{par définition de } \tau \text{ dans l'algorithme} \\
& = & \psi_1(\varphi_1(\alpha) \to \tau_1) & \text{par définition de } \psi \\
& = & \varphi_1'(\alpha) \to \psi_1(\tau_1) & \text{parce que } \alpha \notin V \setminus \{\alpha\} \\
& = & \tau_2' \to \psi_1(\tau_1) & \text{par construction de } \varphi_1' \\
& = & \tau_2' \to \tau_1' & \text{par construction de } \psi_1
\end{array}$$

D'autre part, pour toute variable γ hors de V,

$$\begin{array}{rcll} \psi(\varphi(\gamma)) & = & \psi_1(\varphi_1(\gamma)) & \text{par définition de } \psi \text{ et } \varphi \\ & = & \varphi_1'(\gamma) & \text{puisque } \gamma \notin V \\ & = & \varphi_1(\gamma) & \text{puisque } \gamma \notin V \text{ implique } \gamma \neq \alpha \end{array}$$

D'où le résultat annoncé.

Cas $a = a_1(a_2)$. La dérivation initiale est de la forme

$$\frac{\varphi'(E) \vdash a_1 : \tau'' \to \tau' \qquad \varphi'(E) \vdash a_2 : \tau''}{\varphi'(E) \vdash a_1(a_2) : \tau'}$$

On applique l'hypothèse de récurrence à a_1, E, V, $\tau'' \to \tau'$ et φ'. Il vient

$$(\tau_1, \varphi_1, V_1) = W(a_1, E, V) \qquad \text{et} \qquad \tau'' \to \tau' = \psi_1(\tau_1) \qquad \text{et} \qquad \varphi' = \psi_1 \circ \varphi_1 \text{ hors de } V.$$

En particulier, $\varphi'(E) = \psi_1(\varphi_1(E))$. On applique l'hypothèse de récurrence à a_2, $\varphi_1(E)$, V_1, τ et ψ_1. On a bien $\mathcal{L}(\varphi_1(E)) \cap V_1 = \emptyset$ par la remarque du début de la preuve. Il vient:

$$(\tau_2, \varphi_2, V_2) = W(a_2, \varphi_1(E), V_1) \qquad \text{et} \qquad \tau'' = \psi_2(\tau_2) \qquad \text{et} \qquad \psi_1 = \psi_2 \circ \varphi_2 \text{ hors de } V_1.$$

On a $\mathcal{L}(\tau_1) \cap V_1 = \emptyset$, d'où $\psi_1(\tau_1) = \psi_2(\varphi_2(\tau_1))$. Posons $\psi_3 = \psi_2 + \{\alpha \leftarrow \tau'\}$. (La variable α, choisie dans V_2, est hors de portée de ψ_2, et donc ψ_3 prolonge ψ_2.) On a:

$$\begin{array}{rcccccc} \psi_3(\varphi_2(\tau_1)) & = & \psi_2(\varphi_2(\tau_1)) & = & \psi_1(\tau_1) & = & \tau'' \to \tau' \\ \psi_3(\tau_2 \to \alpha) & = & \psi_2(\tau_2) \to \tau'' & = & \tau'' \to \tau' & & \end{array}$$

La substitution ψ_3 est donc un unificateur de $\varphi_2(\tau_1)$ et $\tau_2 \to \alpha$. L'unificateur principal de ces deux types, μ, existe donc, et $W(a_1(a_2), E, V)$ est bien défini. De plus, on a $\psi_3 = \psi_4 \circ \mu$ pour une certaine substitution ψ_4. On montre maintenant que $\psi = \psi_4$ convient. Avec les notations de l'algorithme, on a bien

$$\psi(\tau) = \psi_4(\mu(\alpha)))) = \psi_3(\alpha) = \tau',$$

d'une part, et d'autre part pour tout $\beta \notin V$ (et donc a fortiori $\beta \notin V_1$, $\beta \notin V_2$, $\beta \neq \alpha$):

$$\begin{array}{rcll} \psi(\varphi(\beta)) & = & \psi_4(\mu(\varphi_2(\varphi_1(\beta)))) & \text{par définition de } \varphi \\ & = & \psi_3(\varphi_2(\varphi_1(\beta))) & \text{par définition de } \psi_4 \\ & = & \psi_2(\varphi_2(\varphi_1(\beta))) & \text{parce que } \beta \neq \alpha \text{ et } \alpha \text{ hors de portée de } \varphi_1 \text{ et de } \varphi_2 \\ & = & \psi_1(\varphi_1(\beta)) & \text{parce que } \varphi_1(\beta) \notin V_1 \\ & = & \varphi'(\beta) & \text{parce que } \beta \notin V. \end{array}$$

C'est le résultat annoncé.

Cas $a = (\texttt{let } x = a_1 \texttt{ in } a_2)$. La dérivation initiale se termine par

$$\frac{\varphi'(E) \vdash a_1 : \tau' \qquad \varphi'(E) + \{x \leftarrow \texttt{Gen}(\tau', \varphi'(E))\} \vdash a_2 : \tau''}{\varphi'(E) \vdash \texttt{let } x = a_1 \texttt{ in } a_2 : \tau''}$$

On applique l'hypothèse de récurrence à a_1, E, V, τ' et φ'. Il vient

$$(\tau_1, \varphi_1, V_1) = W(a_1, E, V) \qquad \text{et} \qquad \tau' = \psi_1(\tau_1) \qquad \text{et} \qquad \varphi' = \psi_1 \circ \varphi_1 \text{ hors de } V.$$

En particulier, $\varphi'(E) = \psi_1(\varphi_1(E))$. On vérifie facilement que $\psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E)))$ est plus général que $\mathtt{Gen}(\psi_1(\tau_1), \psi_1(\varphi_1(E)))$, c'est-à-dire que $\mathtt{Gen}(\tau', \varphi'(E))$. Puisqu'on peut prouver

$$\varphi'(E) + \{x \leftarrow \mathtt{Gen}(\tau', \varphi'(E))\} \vdash a_2 : \tau'',$$

le lemme 1.4 dit qu'on peut a fortiori prouver

$$\varphi'(E) + \{x \leftarrow \psi_1(\mathtt{Gen}(\tau_1, \varphi_1(E)))\} \vdash a_2 : \tau'',$$

c'est-à-dire

$$\psi_1(\varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\}) \vdash a_2 : \tau''.$$

On applique l'hypothèse de récurrence à a_2, dans l'environnement $\varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\}$, avec les variables V_1, le type τ'' et la substitution ψ_1. Il vient

$$(\tau_2, \varphi_2, V_2) = W(a_2,\ \varphi_1(E) + \{x \leftarrow \mathtt{Gen}(\tau_1, \varphi_1(E))\},\ V_1)$$

et $\tau'' = \psi_2(\tau_2)$ et $\psi_1 = \psi_2 \circ \varphi_2$ hors de V_1. L'algorithme prend $\tau = \tau_2$ et $\varphi = \varphi_2 \circ \varphi_1$ et $V' = V_2$. Montrons que $\psi = \psi_2$ convient. On a bien $\psi(\tau) = \tau''$. Et si $\alpha \notin V$, a fortiori $\alpha \notin V_1$, et donc:

$$\begin{array}{rcll} \psi(\varphi(\alpha)) & = & \psi_2(\varphi_2(\varphi_1(\alpha)))) & \text{par définition de } \varphi \\ & = & \psi_1(\varphi_1(\alpha)) & \text{parce que } \varphi_1(\alpha) \notin V_1\text{, puisque } \alpha \text{ hors de portée de } \varphi_1 \\ & = & \varphi'(\alpha) & \text{parce que } \alpha \notin V. \end{array}$$

D'où $\varphi' = \psi \circ \varphi$ hors de V, comme annoncé. □

3.3.3 Typage polymorphe de ML par expansion des `let`

Une autre approche du typage de ML avec polymorphisme provient de la remarque suivante. Supposons que $\mathtt{let}\ x = a_1\ \mathtt{in}\ a_2$ est bien typée, de type τ. Nous avons donc:

$$\frac{E \vdash a_1 : \tau_1 \qquad E + \{x : \mathtt{Gen}(\tau_1, E)\} \vdash a_2 : \tau}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}\ \text{(let-gen)}$$

Le lemme de substitution 2.2 du cours précédent nous dit qu'alors $E \vdash a_2[x \leftarrow a_1] : \tau$. Autrement dit, au lieu de généraliser le type de a_1 et de typer a_2 sous l'hypothèse $x : \mathtt{Gen}(\tau_1, E)$, nous pourrions simplement substituer x par a_1 partout dans a_2, et typer l'expression $a_2[x \leftarrow a_1]$. Plus formellement, cela revient à remplacer la règle (let-gen) ci-dessus par la règle (let-subst) suivante:

$$\frac{E \vdash a_1 : \tau_1 \qquad E \vdash a_2[x \leftarrow a_1] : \tau}{E \vdash \mathtt{let}\ x = a_1\ \mathtt{in}\ a_2 : \tau}\ \text{(let-subst)}$$

La raison pour laquelle il faut quand même typer a_1, et non pas uniquement $a_2[x \leftarrow a_1]$, est que sinon nous laisserions passer des expressions de la forme

$$\mathtt{let}\ x = \text{(expression mal typée)}\ \mathtt{in}\ \text{(expression n'utilisant pas } x\text{)}$$

comme par exemple `let x = 1 2 in 3`.

Exemple: avec la règle (let-subst), on a

$$\emptyset \vdash \texttt{let f} = \texttt{fun}\ x \rightarrow x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}$$

parce que $\emptyset \vdash \texttt{fun}\ x \rightarrow x : \texttt{string} \rightarrow \texttt{string}$ (ou tout autre type à la place de `string`), d'une part, et de l'autre $\emptyset \vdash ((\texttt{fun}\ x \rightarrow x)\ 1, (\texttt{fun}\ x \rightarrow x)\ \texttt{true}) : \texttt{int} \times \texttt{bool}$.

Une fois (let-gen) remplacée par (let-subst), l'environnement de typage E ne contient plus que les identificateurs liés par `fun` (ceux liés par `let` ne sont jamais ajoutés à E). Donc, nous n'avons plus besoin de schémas de types: il suffit de dire que E fait correspondre des types simples τ aux identificateurs x, comme dans le système de types monomorphe. De même, la règle (var-inst) n'est plus nécessaire, et nous pouvons la remplacer par l'axiome (var) du système monomorphe:

$$E \vdash x : E(x) \quad \text{(var)}$$

Nous sommes donc ramenés à un système de types essentiellement monomorphe (le système de la section 1.3.2 plus la règle (let-subst)), auquel nous pouvons appliquer les techniques de la section 3.2: génération d'équations entre types simples et résolution par unification. En particulier, les équations à générer pour une construction `let` sont les suivantes:

$$\text{si } a = \texttt{let}\ x = b\ \texttt{in}\ c,\ C(a) = C(b) \cup C(c[x \leftarrow b]) \cup \{\alpha_a \stackrel{?}{=} \alpha_{c[x \leftarrow b]}\}$$

(Remarque: ceci n'est pas tout à fait exact, car nous avions supposé pour définir $C(a)$ que tous les identificateurs liés dans a avaient des noms différents, et ce n'est certainement pas le cas pour $a = c[x \leftarrow b]$. Par exemple, si $b = \texttt{fun}\ y \rightarrow y$ et $c = x\ x$, nous avons $a = (\texttt{fun}\ y \rightarrow y)\ (\texttt{fun}\ y \rightarrow y)$, dans laquelle y est liée deux fois. Pour être tout à fait correct, il faut considérer les expressions de mini-ML à α-conversion près, en s'autorisant à renommer les identificateurs liés par `let` et `fun` comme dans le λ-calcul. Puis il faut interpréter $c[x \leftarrow b]$ dans la formule ci-dessus comme "l'expression c dans laquelle chaque occurrence de x est remplacée par une copie de b dans laquelle on a renommé tous les identificateurs liés par de nouveaux identificateurs". Dans l'exemple, cela donnerait $c[x \leftarrow b] = (\texttt{fun}\ y' \rightarrow y')\ (\texttt{fun}\ y'' \rightarrow y'')$.)

Pour justifier complètement l'approche décrite ci-dessus, il faut encore montrer que le système de type ML monomorphe + la règle (let-subst) type exactement les mêmes programmes que ML polymorphe (tout programme bien typé dans l'un des systèmes est bien typé avec le même type dans l'autre). C'est une conséquence du théorème suivant:

Théorème 3.3 (Expansion du `let`) *Dans ML polymorphe,* $E \vdash \texttt{let}\ x = a_1\ \texttt{in}\ a_2 : \tau$ *si et seulement s'il existe* τ_1 *tel que* $E \vdash a_1 : \tau_1$ *et* $E \vdash a_2[x \leftarrow a_1] : \tau$.

Démonstration: (esquissée). La partie "seulement si" est une conséquence du lemme 2.2. Pour la partie "si", on montre à l'aide du théorème de principalité de W (théorème 3.2) qu'il existe un type τ_0 qui est principal pour a_1 dans E: c'est-à-dire, $E \vdash a : \tau_0$, et de plus pour tout type τ' tel que $E \vdash a : \tau'$, il existe une substitution ψ telle que $\tau' = \psi(\tau_0)$ et $\mathrm{Dom}(\psi) \subseteq \mathcal{L}(\tau) \setminus \mathcal{L}(E)$. On prend alors $\sigma = \texttt{Gen}(\tau_0, E)$, et on montre par récurrence structurelle sur a que si $E + E' \vdash a_2[x \leftarrow a_1] : \tau$ avec $\mathrm{Dom}(E) \cap \mathrm{Dom}(E') = \emptyset$, alors $E + \{x : \sigma\} + E' \vdash a_2 : \tau$. □

Il faut noter que l'algorithme d'inférence à base de (let-subst) décrit dans cette section, bien que produisant les mêmes résultats que l'algorithme W, est cependant beaucoup moins efficace: sur `let` $x = a_1$ `in` a_2, l'algorithme W type a_1 une seule fois, alors que l'algorithme avec (let-subst) re-type a_1 autant de fois que x est utilisé dans a_2. L'algorithme W est également préférable en pratique car il fournit des messages d'erreurs qui sont directement reliés au source du programme.

3.3.4 Complexité du typage polymorphe de ML

Pour une étude détaillée de l'efficacité de l'algorithme W et de la complexité du problème de typabilité de ML (décider si un programme est typable), on se reportera à Kanellakis, P.C., Mairson, H.G. and Mitchell, J.C., *Unification and ML type reconstruction.* In *Computational Logic: Essays in Honor of Alan Robinson*, ed. J.-L. Lassez and G.D. Plotkin, MIT Press, 1991, pages 444–478. `ftp://theory.stanford.edu/pub/jcm/papers/complexity-type-inf.dvi.Z`. Les principaux résultats sont les suivants:

- Si n est la taille du programme d'entrée, l'algorithme W tourne en temps $O(2^{2^n})$ si le type résultat est représenté par un arbre, et en temps $O(2^n)$ si on représente le type par un graphe acyclique (pour partager des sous-types identiques).
- Le problème de typabilité de ML est NP-dur et DEXPTIME-complet.

Voici un exemple simple de programme de taille $O(n)$ dont le type principal est de taille $O(2^n)$ si représenté par un arbre:

$$\texttt{let } f_0 = \texttt{fun } x \to x \texttt{ in let } f_1 = (f_0, f_0) \texttt{ in } \ldots \texttt{let } f_n = (f_{n-1}, f_{n-1}) \texttt{ in } f_n$$

Malgré cette complexité extrêmement élevée, l'algorithme W est très efficace en pratique (empiriquement, quasi-linéaire en la taille du programme source). Ceci est dû au fait que les programmes réalistes ne ressemblent pas à l'exemple ci-dessus.