Langages de programmation: sémantique et typage
Devoir maison facultatif

François Pottier

24 octobre 2005

Présentation

Les implémentations traditionnelles des langages de la famille ML emploient un glaneur de cellules, ou *garbage collector*, pour libérer automatiquement les blocs de mémoire implicitement alloués par les opérations de construction que propose le langage (λ-abstractions, paires, injections, etc.). Le glaneur de cellules considère la mémoire comme un graphe, dont les racines sont par exemple les variables globales, les registres du processeur, ainsi que les variables allouées sur la pile, et peut, à tout instant, désallouer toute partie de ce graphe qui se révèle inaccessible depuis les racines. Cette technique est sûre mais approchée : certaines parties du graphe, bien qu'accessibles, ne seront peut-être jamais utilisées par le programme. On parle alors de *fuite de mémoire* ou *memory leak*.

Cependant, cette technique n'est pas la seule possible. En effet, la sémantique du langage ML est très abstraite : elle suppose la mémoire infinie et n'indique pas quel mécanisme de libération de mémoire on doit utiliser, en pratique, pour gérer une mémoire finie.

Une seconde technique a été expérimentée dans le compilateur ML Kit. Il s'agit d'un mécanisme d'allocation à base de *régions*. Les régions sont des zones de mémoire : chaque région peut contenir un ensemble de blocs. Lorsqu'on alloue un bloc, on indique dans quelle région on souhaite le créer. De plus, on peut demander la création d'une nouvelle région, initialement vide. On ne peut pas désallouer un bloc individuellement, mais on peut demander explicitement la désallocation d'une région entière, ce qui a pour effet de désallouer tous les blocs qu'elle contient. De plus, les régions sont allouées selon une discipline de pile : on ne peut désallouer que la région la plus récemment allouée. On placera donc typiquement des blocs à courte durée de vie dans les régions les plus récemment créées, et les blocs à longue durée de vie dans des régions plus anciennes. Le langage intermédiaire du compilateur rend explicites la notion de région et les opérations qui s'y rapportent. Le langage source reste ML, sans aucune notion de région. C'est pendant la traduction du langage source vers le langage intermédiaire (qui ne nous intéressera pas ici) que l'on fait apparaître ces opérations, donc que l'on décide quand créer et libérer des régions et dans quelle région allouer chaque bloc.

Ce langage intermédiaire est expressif. Il permet, par exemple, de désallouer une région et tous les blocs qu'elle contient, même si on dispose encore de pointeurs vers certains de ces blocs. On dispose ainsi d'un outil qui peut permettre d'éviter certaines fuites de mémoire, mais qu'il faut manipuler avec précaution, car il faut interdire de déréférencer un pointeur vers un bloc dont la région n'existe plus. Il nous faut donc un système de types sûr pour ce langage intermédiaire. C'est l'objet de cet exercice.

Syntaxe On s'intéresse à un λ-calcul qui représente une version simplifiée du langage intermédiaire du compilateur. Sa syntaxe est la suivante :

$$\begin{array}{lcl} e & ::= & x \mid v \mid \lambda x.e \text{ at } \rho \mid e\ e \mid \text{letregion } \varrho \text{ in } e \\ v & ::= & \langle \lambda x.e \rangle_\rho \\ \rho & ::= & \varrho \mid \bullet \end{array}$$

Ici, ϱ parcourt un ensemble dénombrable de *variables de régions*. Une *région* ρ est soit une variable de régions ϱ, soit la *région invalide* $\bullet$. La construction « letregion ϱ in e » lie la variable de régions ϱ dans l'expression e. Comme toujours, on identifiera les objets manipulés modulo α-conversion des variables et variables de régions liées.

Il s'agit ici d'un λ-calcul pur, où les seules valeurs sont les λ-abstractions. Dans une implémentation réelle, une λ-abstraction est représentée par un bloc en mémoire. Aussi, ici, la valeur $\langle \lambda x.e \rangle_\rho$ est annotée par la région ρ dans laquelle le bloc est alloué. Un appel à cette fonction exigera implicitement un accès en lecture à la région ρ.

La construction $\lambda x.e$ at ρ n'est pas une valeur : il s'agit d'une instruction demandant la création d'un bloc représentant la fonction $\lambda x.e$ dans la région ρ. Dans la mesure où l'on souhaite raisonner à propos de la gestion de la mémoire, nous ne pouvons pas identifier l'instruction $\lambda x.e$ at ρ, qui demande une allocation, et la valeur $\langle \lambda x.e \rangle_\rho$, qui est le résultat de cette allocation.

La construction letregion ϱ in e alloue une nouvelle région, qu'elle lie à la variable de régions ϱ, puis évalue l'expression e, et enfin désalloue la région ϱ. Son résultat est celui de e.

Évaluation Les explications informelles ci-dessus sont formalisées par la donnée d'une sémantique opérationnelle à petits pas. Les règles d'évaluation élémentaires sont les suivantes :

$$
\begin{array}{rcll}
\lambda x.e \text{ at } \varrho & \to & \langle \lambda x.e \rangle_\varrho & (1) \\
\langle \lambda x.e \rangle_\varrho\ v & \to & [x \mapsto v]e & (2) \\
\text{letregion } \varrho \text{ in } v & \to & [\varrho \mapsto \bullet]v & (3)
\end{array}
$$

La première modélise l'allocation d'une λ-abstraction. La seconde est la règle de β-réduction habituelle, en appel par valeur, avec accès à la région ϱ – c'est-à-dire déréférencement d'un pointeur vers un bloc situé dans la région ϱ – pour accéder au bloc qui représente la fonction. On prendra garde de noter que ces deux premières règles font apparaître une variable de régions ϱ, et non une région arbitraire ρ. La troisième règle modélise de façon très abstraite la désallocation d'une région. L'expression située dans la portée de la construction letregion est ici une valeur v, c'est-à-dire que le calcul qui devait être effectué pendant la durée de vie de la région ϱ est terminé. La région ϱ disparaît alors, ce que l'on reflète en remplaçant toute référence à ϱ – s'il en existe – dans v par une référence à la région invalide $\bullet$.

On ajoute aux règles ci-dessus les règles d'évaluation sous contexte :

$$
\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \quad (4)
$$

$$
\frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \quad (5)
$$

$$
\frac{e \to e'}{\text{letregion } \varrho \text{ in } e \to \text{letregion } \varrho \text{ in } e'} \quad (6)
$$

Les deux premières modélisent une évaluation de gauche à droite à l'application de fonction. La dernière permet l'évaluation pendant la durée de vie d'une région ϱ.

Question 1 *Évaluez pas à pas le terme*

$$
(\lambda f.(\lambda y.y \text{ at } \varrho_0) \text{ at } \varrho_0)\ (\text{letregion } \varrho_1 \text{ in } (\lambda x.x \text{ at } \varrho_1))
$$

Que représente, en termes d'implémentation, le sous-terme $\langle \lambda x.x \rangle_\bullet$ que l'on voit apparaître de façon temporaire ? ◇

Question 2 *Caractérisez les termes bloqués en donnant leur grammaire. (On s'intéressera uniquement aux termes clos, c'est-à-dire sans variables x libres.) Que représentent ces termes vis-à-vis d'une implémentation ?* ◇

Typage On se donne maintenant un système de types pour ce calcul. L'objectif est d'interdire toute erreur d'exécution, c'est-à-dire toute utilisation d'une région déjà désallouée. La définition du système est donnée ; l'exercice est d'en démontrer la correction par la méthode syntaxique (via auto-réduction et progrès).

Un *effet* φ est un ensemble fini de régions ρ. Le système de types attribue à toute expression un effet indiquant quelles régions seront potentiellement utilisées (donc doivent être actives) lorsque l'expression sera évaluée. De même, il attribue à toute fonction un effet indiquant quelles régions seront potentiellement utilisées lorsque le corps de la fonction sera évalué.

La syntaxe des *types nus* τ et des *types alloués* μ est la suivante :

$$\begin{array}{rcl} \tau & ::= & \mu \xrightarrow{\varphi} \mu \\ \mu & ::= & (\tau, \rho) \end{array}$$

Le type nu $\mu_1 \xrightarrow{\varphi} \mu_2$ décrit une fonction dont l'argument doit avoir le type μ_1, dont le résultat a le type μ_2, et dont l'effet est φ. Un type alloué (τ, ρ) décrit une valeur dont le type nu est τ et qui a été allouée dans la région ρ, de sorte que toute utilisation de cette valeur exige de déréférencer un pointeur vers un bloc de la région ρ.

Un environnement de typage Γ attribue des types alloués μ aux variables x.

Les règles de typage sont les suivantes :

$$\textsc{var}\quad \Gamma \vdash x : \Gamma(x), \varnothing \qquad \textsc{vabs}\quad \frac{\Gamma; x : \mu_1 \vdash e : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{\Gamma \vdash \langle \lambda x.e \rangle_\rho : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \varnothing} \qquad \textsc{abs}\quad \frac{\Gamma; x : \mu_1 \vdash e : \mu_2, \varphi \qquad \varphi \subseteq \varphi'}{\Gamma \vdash \lambda x.e \text{ at } \rho : (\mu_1 \xrightarrow{\varphi'} \mu_2, \rho), \{\rho\}}$$

$$\textsc{app}\quad \frac{\Gamma \vdash e_1 : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho), \varphi_1 \qquad \Gamma \vdash e_2 : \mu_1, \varphi_2}{\Gamma \vdash e_1\ e_2 : \mu_2, \varphi_1 \cup \varphi_2 \cup \{\rho\} \cup \varphi} \qquad \textsc{letregion}\quad \frac{\Gamma \vdash e : \mu, \varphi \qquad \varrho \notin \mathrm{frv}(\Gamma, \mu)}{\Gamma \vdash \text{letregion } \varrho \text{ in } e : \mu, \varphi \setminus \{\varrho\}}$$

En résumé, la règle ABS reflète la définition informelle de l'effet d'une fonction : il s'agit de l'effet associé au corps de la fonction. De plus, elle reflète le fait qu'une λ-abstraction est allouée dans la région ρ, ce qui exige que ρ soit active. Elle diffère en cela de la règle VABS, qui n'exige pas que ρ soit active. La règle APP indique que les régions auxquelles on accède pendant l'exécution de $e_1\ e_2$ sont (dans l'ordre chronologique) celles auxquelles on accède pendant l'exécution de e_1, puis celles auxquelles on accède pendant l'exécution de e_2, puis la région ρ où cette fonction a été allouée, puis enfin celles auxquelles on accède pendant l'exécution du corps de la fonction. Enfin, la règle LETREGION exige $\varrho \notin \mathrm{frv}(\Gamma, \mu)$, ce qui revient à vérifier que le nom de la région créée pour l'exécution de e n'apparaît pas dans la description de la valeur *finale* de l'expression e, même s'il peut apparaître dans son effet φ. En résumé, e peut utiliser ϱ pendant son calcul, mais ϱ ne doit plus apparaître dans (le type de) son résultat, puisque cette région est désallouée à l'issue de l'évaluation de e. Si ϱ apparaît dans l'effet φ, donc si e utilise ϱ, cela n'est pas observable de l'extérieur, donc l'effet global est $\varphi \setminus \{\varrho\}$.

Question 3 *Quels sont les types et effets possibles pour le terme* $\lambda x.x$ at ρ *dans l'environnement vide ?* ◇

Question 4 *Le terme que vous avez évalué lors de la question 1 est-il bien typé ? Justifiez.* ◇

Question 5 *Expliquez comment on étendrait la syntaxe, la sémantique, et le système de types pour introduire la construction de paire* (e, e) at ρ*, les projections* $\pi_1\ e$ *et* $\pi_2\ e$*, et une construction « let » simplement typée (sans généralisation de variables de types).* ◇

Question 6 *Dans le système ainsi étendu, quel est le type du terme*

$$\text{letregion } \varrho_1 \text{ in let } f = (\lambda y.y \text{ at } \varrho_1) \text{ in } \lambda z.\pi_1\ ((() \text{ at } \varrho_0, f) \text{ at } \varrho_0) \text{ at } \varrho_0 \quad ?$$

Évaluez ce terme et vérifiez que sa valeur mentionne la région invalide $\bullet$. ◇

La question 6 montre que, contrairement à ce que la question 4 aurait pu laisser supposer, certains termes contenant des pointeurs vers des régions déjà désallouées peuvent néanmoins être bien typés. Ceci montre que ce calcul permet, au moins dans certains cas, de désallouer des blocs mémoires (ou régions) pourtant accessibles. La preuve de correction du système de types ne reposera donc pas sur la propriété que tout bloc (ou région) accessible est vivant.

Correction du système de types Nous allons maintenant démontrer la correction du système de types selon l'approche syntaxique étudiée en cours, et due à Wright et Felleisen [3].

On note S une substitution des variables de régions vers les régions, c'est-à-dire une fonction totale, égale à l'identité presque partout, qui à toute variable de régions ϱ associe une région ρ. On étend S en une fonction des régions vers les régions en posant $S(\bullet) = \bullet$.

Question 7 *Démontrez le lemme de substitution suivant : si le jugement $\Gamma \vdash e : \mu, \varphi$ est dérivable, alors le jugement $S(\Gamma) \vdash S(e) : S(\mu), S(\varphi)$ l'est également. On identifiera et on se concentrera sur le ou les cas non triviaux de la preuve.* ◇

On admettra par ailleurs le lemme de substitution classique suivant : si les jugements $\Gamma; x : \mu \vdash e : \mu', \varphi'$ et $\Gamma \vdash v : \mu, \varnothing$ sont dérivables, alors $\Gamma \vdash [x \mapsto v]e : \mu', \varphi'$ l'est également.

Question 8 *Démontrez la propriété d'auto-réduction : si $\Gamma \vdash e : \mu, \varphi$ et $e \rightarrow e'$ sont dérivables, alors il existe un effet φ' tel que $\Gamma \vdash e' : \mu, \varphi'$ est dérivable et φ' est un sous-ensemble de φ, c.à.d. $\varphi' \subseteq \varphi$.* ◇

Question 9 *Expliquez pourquoi la propriété de progrès, ainsi énoncée, est fausse : si $\vdash e : \mu, \varphi$ est dérivable, alors soit e se réduit, soit e est une valeur. Ajoutez une hypothèse appropriée sur φ de façon à corriger l'énoncé, et démontrez que l'énoncé ainsi corrigé est vrai.* ◇

Question 10 *Déduisez des deux questions précédentes un énoncé dit de correction du système de types.* ◇

Question subsidiaire *L'intuition indique qu'une expression ou fonction qui a « peu » d'effets peut être fournie dans tout contexte où une expression ou fonction autorisée à effectuer « plus » d'effets est attendue. Comment formaliser cette intuition en termes de règles de sous-typage ? Étendre la définition du système de types ainsi que sa preuve de correction.* ◇

Pour aller encore plus loin, on peut ajouter au système une notion de polymorphisme de régions, ainsi qu'un algorithme d'inférence de types et de régions. Le mieux est de consulter la littérature à ce sujet.

Références

[1] Cristiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Information and Computation*, 173(2) :199–221, 2002.

[2] Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. In *International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, volume 41 of *Electronic Notes in Theoretical Computer Science*, pages 1–19, 2000.

[3] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1) :38–94, November 1994.