Second projet : un compilateur *PCF* vers une machine abstraite

Le but de ce projet est de réaliser un compilateur du langage *PCF* vers une machine abstraite que l'on munira d'une sémantique opérationnelle. Nous munirons dans un second temps *PCF* d'un système de types.

Les sources peuvent être trouvées à l'adresse suivante : http://www.irisa.fr/celtique/zakowski/compilo.tar.gz. Similairement à ces sources, trois programmes distincts doivent être rendus : un compilateur pour *PCF* non typé (Sections 1 et 2), une adaptation de ce compilateur à *PCF* typé (Section 3) et enfin un compilateur pour *PCF* typé sans annotation de type (Section 4).

1 Une machine abstraite à pile

Plutôt que nous intéresser à un langage machine concret, propre à une architecture et fort complexe, nous allons considérer une machine abstraite ! Cette approche, suivie notamment par Java et Caml, permet entre autres avantages une grande portabilité du code compilé. Le but de cette première étape du projet est de simuler le comportement de cette machine imaginaire : nous allons la doter d'un interpréteur.

Notre machine abstraite va manipuler trois registres. Le premier, l'accumulateur, est une valeur représentant le résultat courant de notre programme. La pile vient pallier l'absence de définitions locales en agissant comme une sorte de mémoire vive où l'on stocke des valeurs intermédiaires. Enfin, l'environnement associant des valeurs à des variables.

Naturellement, cette machine opère sur des programmes. Mais au niveau de ce langage, un programme est simplement une liste d'instructions élémentaires : un quatrième registre contiendra le code.

Voyons d'un peu plus près comment cette machine opère. Nous disposons d'opérations élémentaires de calcul :

— **Ldi** n : met l'entier n dans l'accumulateur ;
— **Push** : met la valeur de l'accumulateur sur le sommet de la pile ;
— **Add** : ajoute le sommet de la pile et l'accumulateur, met le résultat dans l'accumulateur et retire le sommet de la pile.

Ainsi, le terme ((1 + 2) + 3) s'écrit-il dans ce langage **Ldi** *3* ; **Push** ; **Ldi** *2* ; **Push** ; **Ldi** *1* ; **Add** ; **Add**.

S'ajoutent à cela les instructions **Sub**, **Mult** et **Div**, aux sémantiques évidentes, ainsi que **Test** (i) (j) exécutant le code i si l'accumulateur vaut 0, j sinon. Notez bien que i et j sont eux-mêmes du code, c'est à dire une liste d'instructions : ce test est une instruction structurée.

Vient ensuite la gestion de l'environnement, une liste de couples de variables et de valeurs. Notre langage contient donc deux instructions permettant respectivement de lier une variable x à la valeur contenue dans l'accumulateur - **Extend** x - et de mettre dans l'accumulateur la valeur associée dans l'environnement à une variable x - **Search** x.

Lors de l'application d'une fonction, celle-ci va s'exécuter dans l'environnement dans lequel elle avait été définie. Il faut donc avoir un mécanisme pour stocker l'environnement courant, et être capable de le restaurer à la fin de l'appel de fonction. Le langage contient donc à cet effet deux instructions **Pushenv** et **Popenv** permettant de mettre l'environnement courant au dessus de la pile et, inversement, de le récupérer.

Vient enfin la gestion de ce qui tient lieu de fonctions. Une valeur représentant une fonction est composée d'une fermeture : la donnée de deux variables (les noms de la fonction et de son argument), d'une liste d'instructions, le corps de la fonction, et de l'environnement dans lequel la fonction a été définie, et doit donc être exécutée. Viennent donc les deux dernières instructions de notre langage : **MkClose** f x (i) et **Apply**. La première permet de définir une fonction en mettant dans l'accumulateur la fermeture correspondante, l'environnement stocké étant l'environnement courant. La seconde permet l'application d'une fonction : prenant une fermeture stockée dans l'accumulateur, le code de cet fermeture est concaténé en tête du code à exécuter ; en outre, l'environnement doit être modifié en associant à f sa fermeture et à x la valeur située sur le dessus de la pile.

Le jeu d'instructions complet de la machine abstraite est donc : **Add**, **Sub**, **Mult**, **Div**, **Ldi** , **Push**, **Test** (i) (j), **Extend** x, **Search** x, **Pushenv**, **Popenv**, **MkClose** f x (i) et **Apply**.

Consigne de rendu. Réaliser un interpréteur simulant l'exécution de cette machine. La sémantique sera petit pas : le cœur de votre interpréteur sera une fonction *step* de signature
(accumulateur * pile * env * code) → (accumulateur * pile * env * code).

Notez que vous n'aurez pas besoin de Lexer et Parser pour ce langage, celui-ci étant notre langage cible : nous ne souhaitons pas écrire de programmes dans ce langage.

Conseil : on pourra si on le souhaite se restreindre dans un premier temps aux instructions **Ldi** i, **Push** et **Add** avant d'étendre son interpréteur au reste du langage. Concevez au fur et à mesure une batterie de jeux de tests pour vous aider à tester, mettre au point et débugger votre interpréteur. Une fois confiant, vérifiez que le programme suivant renvoie bien 720 :

Pushenv ; **MkClose** f x (**Search** x ; **Test** (**Ldi** 1) (**Pushenv** ; **Ldi** 1 ; **Push** ; **Search** x ; **Sub** ; **Push** ; **Search** f ; **Apply** ; **Popenv** ; **Push** ; **Search** x ; **Mult**)) ; **Extend** f ; **Pushenv** ; **Ldi** 6 ; **Push** ; **Search** f ; **Apply** ; **Popenv** ; **Popenv**

2 Un compilateur de *PCF* vers la machine abstraite

Nous considérons à présent le langage *PCF* tel qu'étudié en cours. Rappelons-en la syntaxe :

$$t ::= n \mid x \mid t + t \mid t - t \mid t * t \mid t/t \mid fun\ x \to t \mid t\ t \mid fixfun\ f\ x \to t \mid$$
$$let\ x := t\ in\ t \mid ifz\ t\ then\ t\ else\ t$$

Vous trouverez un parser et un lexer pour ce langage dans les sources fournies.

Nous souhaitons à présent réaliser un compilateur de *PCF* vers le langage de notre machine abstraite.

Consigne de rendu. Définir une fonction *compil* prenant pour argument un terme *PCF* et renvoyant une liste d'instructions dans le langage de notre machine abstraite ayant le même comportement attendu. [1]

Écrire un *top-level* permettant la génération d'un exécutable prenant en argument un fichier contenant un programme *PCF*, imprimant ce programme *PCF*, le programme compilé et finalement la valeur obtenue en interprétant ce programme compilé.

Tester votre compilateur sur une batterie de tests élémentaires, puis sur le programme *fact.pcf* fourni dans les sources.

3 *PCF* typé

Tout comme le λ-calcul, *PCF* dans son habitat naturel est un animal sauvage : il autorise d'étrange comportements tels que le terme $(fun\ x\ \rightarrow\ x)(fun\ x\ \rightarrow\ x)$ se réduisant en lui-même *ad vitam eternam*. Nous allons domestiquer notre langage en lui affectant des types.

En effet, une explication à ces comportements indésirables est l'absence de domaine de définition de nos fonctions : une fonction accepte des objets de n'importe quelle nature en tant qu'argument. Nous voulons modifier la syntaxe de notre langage pour que le type soit explicitement déclaré.

Plus précisement, nous considérons le langage de types suivant : un type est soit N*at*, le type des entiers naturels, soit $\tau_1 \rightarrow \tau_2$ où τ_1 et τ_2 sont des types, représentant naturellement le type des fonctions prenant un argument de type τ_1 et renvoyant un résultat de type τ_2.

La syntaxe de *PCF* est alors modifiée comme suit :

$$
\begin{aligned}
\tau &::= \mathrm{N}at \mid \tau \ \rightarrow\ \tau \\
t &::= n \mid x \mid t + t \mid t - t \mid t * t \mid t/t \mid fun\ (x :\ \tau)\ \rightarrow\ t \mid t\ t \mid fixfun\ (f :\ \tau)\ x\ t \mid \\
&\qquad let\ x := t\ in\ t \mid ifz\ t\ then\ t\ else\ t
\end{aligned}
$$

Ces nouvelles annotations de type n'ont aucune incidence sur la sémantique de notre programme, et donc en particulier sur notre compilateur. Vous pouvez donc aisément adapter votre compilateur à cette nouvelle syntaxe.

Mais elles vont par contre nous être précieuses pour refuser des fonctions à la nature indésirable. Nous souhaitons en effet écrire une fonction de relation de typage : étant donné un terme t et un environnement de typage Γ (une fonction associant un type aux variables), la relation de typage doit retourner le type de t dans Γ si ce terme est bien typé, et lever une erreur dans le cas contraire. En particulier, si le terme que l'on souhaite compiler est bien typé à partir de l'environnement de typage vide, la compilation peut alors avoir lieu.

Consigne de rendu. Adapter votre compilateur à la nouvelle syntaxe de *PCF* et implémenter l'algorithme de typage pour n'accepter la compilation que des termes

1. C'est à dire la même sémantique dans un sens qu'il nous faudrait préciser. Nous resterons ici informel sur ce point.

bien typés. On renverra un message d'erreur aussi informatif que possible dans le cas contraire afin d'aider l'utilisateur à débugger son programme.

4 Pour aller plus loin : Inférence de type

Le typage explicite demande un travail additionnel au programmeur. Nous voudrions pouvoir éviter cela en réalisant de l'inférence de type : c'est le compilateur qui, en l'absence d'annotation, parvient à déterminer si le terme est bien typé. Pour ce faire nous revenons donc à la syntaxe initiale (Section 2) de *PCF*.

Nous nous proposons ici de réaliser l'inférence de types à l'aide de l'algorithme de Hindley. Celui-ci se décompose en deux phases : la génération de contraintes, puis leur résolution.

Puisque nous manquons a priori d'information, celle qui était préalablement fournie par l'utilisateur, nous introduisons des variables de type : un type peut à présent être N*at*, $\tau \to \tau$ ou une variable. L'objet de la première phase est alors de suivre un schéma similaire à l'algorithme de typage utilisé en Section 3 pour collecter des contraintes sur ces variables inconnues : l'algorithme renvoie à présent un type et un ensemble de contraintes sous la forme d'une liste d'égalités entre types.

Ainsi, lors du typage d'une fonction $fun\ x \to t$ par exemple, nous typons le terme t dans un environnement de type où la variable x est associée à une nouvelle [2] variable de type X. En contrepartie, en supposant que u est récursivement typé avec un type τ_1 et des contraintes A, v un type τ_2 et des contraintes B, le terme $u\ v$ est alors typé par une variable fraîche X et renvoie les contraintes $A \cup B \cup \{\tau_1 = \tau_2 \to X\}$.

Vient alors la passe de résolution des contraintes. L'idée est de procéder par simplifications successives du système jusqu'à ce que celui-ci soit stable par les règles de transformation. Pour ce faire, on regarde les contraintes une à une et l'on essaie de leur appliquer un ensemble de règles défini comme suit. Lorsque la contrainte considérée admet pour forme :

- $\tau_1 \to \tau_2 = \tau_3 \to \tau_4$, la remplacer dans le système par les deux équations $\tau_1 = \tau_3$ et $\tau_2 = \tau_4$;
- N*at* = N*at*, la supprimer du système ;
- X = X, la supprimer du système ;
- N*at* = $\tau_1 \to \tau_2$ ou $\tau_1 \to \tau_2$ = N*at*, lever une erreur (le système n'a pas de solution, le terme est mal typé) ;
- X = τ ou τ = X, et si X apparait dans τ, lever une erreur (le système n'a pas de solution, le terme est mal typé) ;
- X = τ ou τ = X, et si X n'apparait pas dans τ, la supprimer et substituer X par τ dans toutes les autres équations du système.

On peut (mais cela n'est pas demandé ici) prouver que cet algorithme est correct et termine : t est bien typé si et seulement si le système atteint un état où aucune des contraintes ne se trouveront modifiées par ces règles. Sinon, une erreur sera levée.

2. Vous trouverez dans les sources une fonction *fresh_var* permettant de générer des variables fraîches.

Consigne de rendu. Implémenter cet algorithme d'inférence de types et l'interfacer avec votre compilateur pour n'accepter à la compilation que les termes bien typés. On renverra un message d'erreur aussi informatif que possible dans le cas contraire afin d'aider l'utilisateur à débugger son programme.

5 Consignes générales

Les sources fournies sont volontairement minimales. Libre à vous d'organiser et décomposer votre code comme bon vous semble. Les seules contraintes demandées sont l'utilisation des fichiers fournis et pour les rares fonctions dont la signature est fournie, la conservation de leur nom.

Code source Le code de votre projet est à envoyer à yannick.zakowski@irisa.fr avant mercredi 03 décembre, 20h (pénalité d'un point par heure de retard).

Vous mettrez ce code dans une archive nommée `Nom1-Nom2` (avec vos deux noms). Celle-ci contiendra trois dossiers, présentant votre compilateur respectivement pour *PCF* sans types, explicitement typé et avec inférence de type. Vous fournirez, outre le code source et des programmes *PCF* de test, des instructions en français pour le compiler et l'exécuter sur vos tests. En l'absence d'un fichier `README` correctement écrit, votre code ne sera pas examiné.

En vue des soutenances (voir ci-dessous), votre archive contiendra également un fichier Makefile, fournissant impérativement les deux cibles suivantes :

— `make comp` pour compiler un exécutable du compilateur, et permettant de prendre des noms de fichiers (contenant des programmes *PCF*) en argument

— `make demo` pour compiler un exécutable spécial pour la réalisation de la démonstration. Cet exécutable ne prendra aucun argument, la commande `./demo` doit lancer la démo.

Vos archives reçues seront compilées, et testées par les encadrants. Un makefile absent ou ne répondant pas à ces contraintes sera pénalisé.

Soutenances Jeudi 04 décembre, de 8h à 10h. Chaque binôme présentera son travail en 10 minutes et répondra à des questions pendant 5 minutes. Durant la soutenance, chaque binôme devra réaliser une démonstration express (vu le temps imparti) de son compilateur.

Rapport Votre rapport, nommé `Nom1-Nom2.pdf`, est à envoyer à la même adresse, au format PDF, avant vendredi 05 décembre, 23h59 (pénalité d'un point par heure de retard). Celui-ci peut être au choix en français ou en anglais.