IMT Atlantique

Compilateur

Langages et Logiques

BARTHELEMY Théo
BEN AYED Khadija

Chargé de TD : BACH Jean-Christophe

Rendu Compilation

April 14, 2024

Exercice 1

Une pile est une structure de données linéaire qui suit le principe de "dernier entré, premier sorti" (LIFO). I.e le dernier élément ajouté à la pile sera le premier à être retiré.

Les opérations les plus courantes que l'on exécute sur une pile sont:

- **Push** : ajouter un élément en haut de la pile.
- **Pop** : retirer l'élément du haut de la pile.
- **Peek** : retourner l'élément du haut de la pile sans le retirer.
- **IsEmpty** : vérifier si la pile est vide ou non.
- **Clear** : enlever tous les éléments de la pile.
- **Size** : retourner le nombre d'éléments dans la pile.

Exercice 2

Etat Initial: La pile est vide
Étape 1 : Push 0
→ Pile: 0

Étape 2 : Push 12
→ Pile : 0, 12

Étape 3 : Push 7
→ Pile : 0, 12, 7

Étape 4 : Swap
⇒ L'échange échange les deux premiers éléments de la pile.
→ Pile après l'échange: 0, 7, 12

Étape 5 : Sub
⇒ Prend les deux premiers éléments (7 et 12) $\Longrightarrow$ les soustrait $\Longrightarrow$ pousse le résultat.
→ Pile après la soustraction: 0, 5 $\Longleftarrow$ **ETAT FINAL**

Résultat final du programme :

0 push 12 push 7 swap sub est 5

Exercice 3

3.1.

Règle 1 :

La régle (1) correspond à la sématique suivante :

$$(1)\quad \frac{i \neq n}{v_1, \ldots, v_n \vdash i, Q \Rightarrow Err}$$

Si i ne correspond pas au nombre de valeurs n sur la pile alors le programme aboutira à une erreur.
⇒ Si la pile ne contient pas exactement le nombre de valeurs attendues, il y aura une erreur.

Règle 2 :

La régle (2) correspond à la sématique suivante :

$$(2)\quad \frac{Q, v1 :: \ldots :: v_n :: \varnothing \rightarrow^* Err}{v_1, \ldots, v_n \vdash n, Q \Rightarrow Err}$$

Si l'exécution de la séquence d'instructions Q avec n arguments attendus sur une pile vide aboutit à une erreur → le programme se finit en erreur.
⇒ Si une tentative d'exécution du programme avec le nombre d'arguments attendus crée une erreur, le programme en entier envoie une erreur.

Règle 3 :

La régle (3) correspond à la sématique suivante :

$$(3)\quad \frac{Q, v1 :: \ldots :: v_n :: \varnothing \rightarrow^* \varnothing, v :: S}{v_1, \ldots, v_n \vdash n, Q \Rightarrow v}$$

Si l'exécution de la séquence d'instructions Q avec n arguments attendus sur une pile contenant les valeurs de v1 à vn mène à une pile intermédiaire v::S après plusieurs étapes (alors le résultat du programme est la valeur v.
⇒ Si après avoir exécuté toutes les instructions il reste une seule valeur sur la pile, alors cette valeur est le résultat du programme.

3.2.

En analysant les 3 règles données, on remarque que les cas où :

- Le nombre d'arguments attendus ne correspond pas au nombre de valeurs sur la pile (règle (1) ; aboutit à une erreur),
- L'exécution de la séquence d'instructions avec le bon nombre d'arguments sur une pile vide (règle (2) ⇒ erreur),
- Et toutes les instructions ont été exécutées avec succès et qu'il ne reste qu'une valeur sur la pile (règle (3) ⇒ v).

Ont été traités (voir 3.1).
Il manque donc une règle pour traiter la situation dans laquelle toutes les instructions sont exécutées avec succès et que la pile en toute fin est vide.

On doit alors se poser la question de savoir si ce scénario doit renvoyer une erreur ou non.

$$\frac{Q,\, v1 :: \ldots :: v_n \;::\; \varnothing \rightarrow^* \varnothing, \varnothing}{v_1, \ldots, v_n \vdash n\,, Q \Rightarrow ??}$$

Or, la plupart du temps, lorsqu'on essaie de réaliser une opération sur une pile vide, une erreur devrait être déclenchée, il n'est en effet souvent pas possible de récupérer ou de manipuler des données à partir d'une pile vide.

Le cas non traité est donc :

$$\frac{Q,\, v1 :: \ldots :: v_n \;::\; \varnothing \rightarrow^* \varnothing, \varnothing}{v_1, \ldots, v_n \vdash n\,, Q \Rightarrow \; Err}$$

3.3.

1. **Règle de Terminaison** :
   Si la séquence d'instructions est vide, la computation se termine.

   $$\frac{I \;=\; \varnothing}{I.Q, S \;\rightarrow\; Q, S}$$

2. **Push** :
   Si l'instruction est un push, elle pousse la valeur sur le haut sur la pile.
   Runtime error : dans le cas où une des données n'existe pas où si elle débouche sur une divisions par 0 (i.e le second opérande est 0)

   $$\frac{I \;=\; PUSH\; v}{I.Q, S \;\rightarrow\; Q,\, v :: S}$$

3. **Pop** :
   Si l'instruction est une opération pop, la pile est mise à jour avec l'élément retiré.

   $$\frac{I \;=\; POP}{I.Q, \_ :: S \rightarrow\; Q,\, S}$$

   Runtime error : dans le cas où la pile est vide

   $$\frac{I \;=\; POP}{I.Q, \varnothing \;\rightarrow\; Err}$$

4. **Swap** :
   Si l'instruction est une opération de swap, elle échange les deux premiers éléments sur la pile.

   $$\frac{I \;=\; SWAP}{I.Q, v_1 :: v_2 :: S \rightarrow\; Q, v_2 :: v_1 :: S}$$

   Runtime error : dans le cas où la pile contient moins de deux éléments

   $$\frac{I \;=\; SWAP}{I.Q, v_1 :: \varnothing \rightarrow\; Err} \qquad ou \qquad \frac{I \;=\; SWAP}{I.Q, \varnothing \rightarrow\; Err}$$

5. **Règle d'Addition** :
   Si l'instruction est une opération d'addition, le programme doit sommer v1 et v2.

   $$\frac{r \;=\; v1 + v2}{ADD.\, Q, v_1 :: v_2 :: S \rightarrow\; Q, r :: S}$$

   Runtime error : dans le cas où un seul opérande existe

   $$ADD.\, Q, v_1 :: \varnothing \rightarrow\; Err$$

6. **Règle de soustraction** :
   Si l'instruction est une opération d'addition, le programme doit sommer v1 et v2.

$$\frac{r = v1 - v2}{SUB.Q, v_1 :: v_2 :: S \rightarrow Q, r :: S}$$

   Runtime error : dans le cas où un seul opérande existe.

$$SUB.Q, v_1 :: \varnothing \rightarrow Err$$

7. **Règle de multiplication** :
   Si l'instruction est une opération d'addition, le programme doit multiplier v1 et v2.

$$\frac{r = v1 \times v2}{MUL.Q, v_1 :: v_2 :: S \rightarrow Q, r :: S}$$

   Runtime error : dans le cas où un seul opérande existe.

$$MUL.Q, v_1 :: \varnothing \rightarrow Err$$

8. **Règle de division** : Si l'instruction est une opération d'addition, le programme doit diviser v1 par v2.

$$\frac{r = \frac{v_1}{v_2}, \quad v_2 \neq 0}{DIV.Q, v_1 :: v_2 :: S \rightarrow Q, r :: S}$$

   Runtime error : dans le cas où un seul opérande existe OU v2 est égal à 0.

$$DIV.Q, v_1 :: \varnothing \rightarrow Err \qquad ou \qquad DIV.Q, v_1 :: 0 :: S \rightarrow Err$$

Exercice 5

Description formelle des cas :

- `generate(Const n)` : Convertit la constante n en Pfx en plaçant simplement n sur la pile en appelant Push.
  [`PUSH n`]
- `generate(Binop (Badd, e1, e2))` : Pour une addition, traite d'abord les sous-expressions e1 et e2, puis ajoute leurs résultats ensemble sur la pile.
  [`generate(e2) ; generate(e1) ; ADD`]
- `generate(Binop (Bsub, e1, e2))` : Pour une soustraction, procède de manière similaire à l'addition, mais soustrait e2 de e1.
  [`generate(e2) ; generate(e1) ; SUB`]
- `generate(Binop (Bmul, e1, e2))` : Pour une multiplication, procède de manière similaire à l'addition, mais multiplie e1 par e2.
  [`generate(e2) ; generate(e1) ; MUL`]
- `generate(Binop (Bdiv, e1, e2))` : Pour une division, procède de manière similaire à l'addition, mais divise e1 par e2.
  [`generate(e2) ; generate(e1) ; DIV`]
- `generate(Binop (Bmod, e1, e2))` : Pour un modulo, procède de manière similaire à l'addition, mais prend le reste de la division de e1 par e2.
  [`generate(e2) ; generate(e1) ; REM`]
- `generate(Uminus e)`: Pour une négation, empile d'abord 0 sur la pile, puis soustrait e de 0, simulant ainsi l'effet de la négation.
  [`generate(e) ; SUB`]

Exercice 9

9.1.

Les règles pour les constructions déjà définies, telles que Const, Var, Binop et Uminus, restent inchangées car l'introduction de fonctions et d'applications ne modifie pas leur fonctionnement. Ces constructions continuent à être traitées selon les règles existantes dans le langage.
Les nouvelles constructions, comme App et Fun n'affectent pas le traitement des constructions existantes.

Conclusion

⇒ Les règles déjà établies pour **Const**, **Var**, **Binop** et **Uminus** sont toujours valides. **Elles ne nécessitent pas de modification.**

9.2.

1. Séquence exécutable (Q) :
   Lorsqu'une séquence exécutable est rencontrée, elle est placée sur le dessus de la pile.

$$Q.\,I\,.\,J, S \rightarrow\ I\,.\,J, (Q) :: S$$

2. Exec :
   Lorsque l'instruction exec est exécutée, elle dépile le dessus de la pile et l'exécute en l'ajoutant au début de la séquence en cours d'exécution.

$$\frac{I\ = EXEC}{I.\,J\,, (Q) :: S \rightarrow\ Q\,.I.J, S}$$

   Dans le cas où le haut de la pile n'est pas une séquence exécutable :

$$\frac{I\ = EXEC, \quad J \neq (Q)}{I\,.\,J, v_1 :: S \rightarrow Err}$$

3. Get :
   Lorsque l'instruction get est exécutée avec un indice valide i, elle copie la i-ème valeur de la pile au dessus de la pile.

$$\frac{I\ = GET, \quad v_1\ = n}{I.\,J\,, v_1 :: v_2 :: \ldots :: v_n :: S \rightarrow\ v_n :: v_{n-1} :: \ldots :: v_2 :: S}$$

   Si la pile ne contient pas suffisamment d'éléments pour effectuer l'opération get, une erreur est générée.

$$\frac{I\ = EXEC, \quad \#S < v_1}{I.\,J\,, v_1 :: S \rightarrow\ Err}$$

Exercice 10

10.1.

La version compilée de l'expression (λx.x + 1) 2 en Pfx est :

push 2 seq_start push 0 get push 1.

Explication : On commence par push 2 sur la pile.
Au deuxième état, on a donc une pile [2].
Ensuite, une nouvelle séquence exécutable est poussée en haut de la pile. Elle sera par la suite executée en récupérant l'élément d'indice 0 sur la pile (ici, 2) puis pousse 1 et éxecute l'addition.
On obtient alors 3 qui est placé sur la pile.

10.2.

e → *Exec_Seq*[`push v; push` [*t*]`; exec`] , Avec :

- v est une valeur
- t une expression transformée en une séquence exécutable et empilée sur la pile.

10.4.

La version compilée est :

push 12 push 8 *seq_start* push 0 get push 0 get swap sub *seq_end* exec

Explication :

Dans cette traduction Pfx, nous commençons par mettre les nombres 12 et 8 sur la pile avec les instructions push 12 et push 8.
Ensuite, nous traduisons l'expression (λx.λy.(x - y)) en une séquence exécutable.
Pour cela, nous utilisons get pour accéder à nos arguments et effectuer la soustraction.
Le problème est que pour accéder à nos arguments, nous devons chaque fois pousser leur position.

Exercice 11

11.2.

'let x = e1 in e2' permet de créer une nouvelle variable nommée x, d'assigner son résultat à l'expression e1, et d'utiliser cette variable dans l'expression e2.

let $x = e1$ in $e2$ peut être formulée dans le calcul lambda comme :

$(\lambda x \cdot e2) \cdot e1$

Exercice 12

Expr donnée : $(((\lambda x \cdot \lambda y \cdot (x - y))\, 12)\, 8)$

Étape 1 - Évaluation de la fonction externe $(\lambda yx \cdot \lambda yy \cdot (x - y))\, 12$:
App : L'expression externe est une application de fonction.
$\lambda x \cdot \lambda y \cdot (x - y)$ **évalue à** $\{E', x, \lambda y \cdot (x - y)\}$ où E' est l'environnement étendu avec x = 12.
12 évalue à 12.
App : Appliquer la fonction à l'argument 12.
12 est lié à x, donc x = 12.
Nous obtenons {E", y, 12 - y} où E" est l'environnement étendu avec y = 12.

Étape 2 - Évaluation de l'expression interne $(\lambda y \cdot (x - y))\, 8$:
App : L'expression interne est une application de fonction.
$\lambda y \cdot (x - y)$ **évalue à {E", y, x - y}** où E" est l'environnement étendu avec y = 12.
8 évalue à 8.
App : Appliquer la fonction à l'argument 8.
8 est lié à y, donc y = 8.
Nous obtenons {E"', x, 12 - 8} où E"' est l'environnement étendu avec x = 12 et y = 8.

Étape 3 - Évaluation de (x - y) : 12-8 évalue à 4.

Résultat final :

La valeur de l'expression $(((\lambda x.\lambda y \cdot (x - y))\, 12)\, 8)$ est 4.

Exercice 13

0.1 13.1.

Pour traduire le langage Expr en Pfx, nous devons tenir compte de la manière dont Expr traite les fonctions et les variables, et adapter cela à la façon dont Pfx fonctionne.La traduction de Expr en Pfx implique de représenter les fonctions en tant que séquences d'instructions et de gérer les variables libres en les plaçant sur la pile avant d'exécuter les fonctions. Nous nous penchons sur 3 aspects qu'il nous faudra traiter pour assurer la traduction :

1. **Fonctions** : Expr permet la définition de fonctions à l'aide de l'opérateur lambda (λ). En Pfx, nous ne disposons pas directement de fonctions. Au lieu de cela, nous utilisons des séquences d'instructions qui représentent le comportement de la fonction. Donc, pour chaque fonction définie en Expr, nous devons traduire son corps en une séquence d'instructions Pfx.

2. **Variables** : Les variables en Expr peuvent être liées (définies dans le contexte de la fonction) ou libres (définies en dehors de la fonction). En Pfx, nous pouvons utiliser des variables pour stocker des valeurs sur la pile, mais elles ne sont pas liées au contexte d'une fonction. Pour gérer les variables libres dans Expr lors de la traduction en Pfx, nous devons trouver un moyen de stocker ces valeurs sur la pile avant d'exécuter la fonction.

3. **Gestion des Fonctions** : Lorsque nous exécutons une fonction en Pfx, nous devons garantir que toutes les variables libres nécessaires à son exécution sont disponibles sur la pile au moment de son exécution.

13.2.

L'opération 'append' est utilisée pour modifier une séquence exécutable en ajoutant une valeur à son début. Elle prend deux arguments : une valeur v et une séquence exécutable Q. Elle modifie la séquence exécutable Q en plaçant la valeur v au début de la séquence.

L'opération 'append' se définit par :

$$\texttt{append} : (\, V \times Q \,) \to Q$$

où V est l'ensemble des valeurs possibles et Q est l'ensemble des séquences exécutables.

La sémantique formelle de l'opération 'append' est définie par la règle de transition suivante :

$$\texttt{append}\,(v, Q) \Rightarrow Q'$$

où v est la valeur à ajouter, Q est la séquence exécutable initiale, et Q' est la nouvelle séquence exécutable obtenue en ajoutant la valeur v au début de la séquence Q.

Cependant, plusieurs cas d'erreur peuvent être retrouvés :

1. **Pile vide ou nombre incorrect d'éléments** :
   Si la pile est vide ou ne contient pas le nombre requis d'éléments pour l'opération 'append', cela peut être exprimé comme suit :
   $$\texttt{append}(v, \varnothing) \Rightarrow \text{Error, } \varnothing \text{ représente une pile vide.}$$

2. **Type incompatible de la valeur en haut de la pile** :
   Si le type de la valeur en haut de la pile n'est pas compatible avec l'opération 'append', cela pourrait être écrit comme :
   $$\texttt{append}(v, Q) \Rightarrow \text{Error,} Q \text{ est la séquence exécutable initiale.}$$

3. **Exécution de la séquence exécutable terminée prématurément** :
   Si la séquence exécutable Q est terminée avant l'ajout de la valeur sur la pile, cela peut être représenté comme :
   $\texttt{append}(v, \texttt{Finished}) \Rightarrow$ Error, `Finished` représente une séquence exécutable qui est terminée..

4. **Erreur lors de l'ajout de la valeur à la séquence exécutable** :
   Si une erreur se produit lors de l'ajout de la valeur sur la séquence exécutable, cela pourrait être écrit comme :
   $$\texttt{append}(v, Q) \Rightarrow \text{Error, } Q \text{ est la séquence exécutable initiale.}$$

13.4.

Les règles de Traduction de Expr vers Pfx pour le Support des Fermetures sont :

1. **Traduction des Constantes** :
   la constante n est ajoutée à la pile de manière à ce qu'elle soit disponible pour les opérations ultérieures de la machine Pfx.
   $\texttt{generate}\,(n) \rightarrow \text{PUSH } n$

2. **Traduction des Variables** : Accès à la valeur de la variable x dans l'environnement et placer sa valeur sur la pile.
   $E(x) = v,\ \texttt{generate}\ (x) \rightarrow \text{PUSH } v$

3. **Traduction de l'Opération unaire Moins** :
   Évaluer l'expression e puis soustraire son résultat de 0.
   $\texttt{generate}\,(e) \rightarrow \texttt{generate}\,(e); \text{SUB}$

4. **Traduction des Opérations Binaires** :
   Évaluer les sous-expressions e_1 et e_2 , puis effectuer l'opération binaire correspondante.

   - Addition : $\texttt{generate}\,(\mathit{Binop}(\mathit{Badd}, e_1, e_2)) \rightarrow \texttt{generate}\,(e_2); \texttt{generate}\,(e_1); \text{ADD}$
   - Soustraction : $\texttt{generate}\,(\mathit{Binop}(\mathit{Bsub}, e_1, e_2)) \rightarrow \texttt{generate}\,(e_2); \texttt{generate}\,(e_1); \text{SUB}$
   - Multiplication : $\texttt{generate}\,(\mathit{Binop}(\mathit{Bmul}, e_1, e_2)) \rightarrow \texttt{generate}\,(e_2); \texttt{generate}\,(e_1); \text{MUL}$
   - Division : $\texttt{generate}\,(\mathit{Binop}(\mathit{Bdiv}, e_1, e_2)) \rightarrow \texttt{generate}\,(e_2); \texttt{generate}\,(e_1); \text{DIV}$
   - Modulo : $\texttt{generate}\,(\mathit{Binop}(\mathit{Bmod}, e_1, e_2)) \rightarrow \texttt{generate}\,(e_2); \texttt{generate}\,(e_1); \text{REM}$

5. **Traduction de l'Abstraction de Fonction (Création de Fermeture)** :
   Traduire le corps de la fonction e en une séquence exécutable, capturant les variables libres dans l'environnement.
   Soit $FV(e)$ l'ensemble des variables libres dans e.
   $\texttt{generate}\,(\mathit{Fun}(x, e)) \rightarrow \{E, x, \texttt{generate}\,(e)\}$, E est l'environnement étendu avec les variables libres actuelles.

13.6.

1. Push la valeur 12 dans la pile : **Push 12**
2. Début de la fermeture en plaçant une valeur sur la pile : **PushClosure 1**
3. Push la valeur 12 dans la pile : **Push 8**
4. Echange des 2 valeurs : **Swap**
5. Soustraction : **Sub** ce qui génère la valeur 4
6. Exécution de la séquence d'instructions stockée dans la fermeture : **EndClosure**

Cette traduction Pfx est plus efficace car elle évite les calculs inutiles et utilise efficacement la pile.