

TME 7 – Coroutines

Antoine Miné

1 Présentation

Les *coroutines* sont une forme étendue d'appels de fonctions, permettant un flot de contrôle plus complexe. Lors d'un appel de fonction classique, l'exécution de l'appelant est suspendue jusqu'à la terminaison complète de la fonction appelée. Un appel de coroutine permet, au contraire, à la coroutine appelée de s'interrompre temporairement, avant sa fin normale, et de revenir à l'appelant. L'appelant pourra alors continuer l'exécution de son code un moment, puis revenir à la coroutine pour reprendre son exécution à l'endroit où celle-ci s'était interrompue, jusqu'à sa prochaine suspension, et ainsi de suite. Voici un exemple de coroutine :

```
1  function f()
2  (
3    print("b1 ");
4    yield();
5    print("b2 ");
6    yield();
7    print("b3 ")
8  );
9
10
11
12  let x = costart f() in (
13    print("a1 ");
14    resume(x);
15    print("a2 ");
16    resume(x);
17    print("a3 ");
18    resume(x);
19    print("a4 ");
20    resume(x);
21    print("a5 ")
22  )
```

Le résultat affiché sera : a1 b1 a2 b2 a3 b3 a4 a5.

Nous notons les points suivants :

- Une coroutine `f` est définie comme une fonction globale.
- L'appel à `f()` est remplacé par `let x = costart f()`, qui démarre une nouvelle instance de la coroutine `f` et stocke l'instance dans `x`, sans commencer l'exécution de `f`.
- `f` commence réellement à s'exécuter au premier `resume(x)`, donc après l'affichage de `a1`.
- L'exécution se poursuit dans `f`, jusqu'au premier `yield()`. Elle affiche donc `b1`.
- Après un `yield()`, le flot d'exécution reprend dans la fonction appelante, immédiatement après l'instruction `resume(x)`. L'exécution affiche donc `a2`.
- L'exécution se poursuit dans l'appelant jusqu'au deuxième `resume(x)`. À ce moment, `f` reprend la main au deuxième `yield()`.
- L'exécution continue de sauter de l'appelant à `f`, au gré des `yield()` et des `resume(x)`.
- Après la fin de `f` atteinte, les instructions `resume(x)` n'ont plus aucun effet, donc l'affichage de `a4` est immédiatement suivi de `a5`.

L'exemple suivant illustre deux points supplémentaires : le passage d'arguments à une coroutine et l'exécution concurrente de plusieurs instances de la même coroutine (chacune ayant ses propres variables locales et son flot d'instructions) :

```
1  function f(nb, msg)
2  (
3    let i = 0 in
4    while i < nb do
5      (
6        print(msg);
7        i = i + 1;
8        yield()
9      )
10 );
```

```

11                                     18  (
12                                     19      print(i);
13  let c1 = costart f(5,"A") in      20      resume(c1);
14  let c2 = costart f(3,"B") in      21      resume(c2);
15  let c3 = costart f(7,"C") in      22      resume(c3);
16  let i = 0 in                      23      i = i + 1
17  while i < 10 do                   24  )

```

La notion de coroutine existe en Python, à travers les générateurs. Par contre, de nombreux langages populaires (C, Java, C++) n'incluent pas nativement les coroutines.

Objectif. Ajouter à ILP3 le support pour les coroutines.

Buts :

- revoir l'ajout d'une construction dans ILP, de la syntaxe à l'interprète et au compilateur ;
- comprendre l'appel dans ILP aux fonctions globales ;
- comprendre les limitations inhérentes au modèle d'exécution à pile unique, et comment y remédier ;
- revoir ou s'initier aux *Threads* Java.

Version d'ILP. Nous nous baserons sur le code ILP3. Ce code est disponible sur le GitLab du cours (<https://stl.algo-prog.info>), dans le projet ILP3 du groupe du semestre en cours (DLP-2020oct pour 2020–2021). Comme pour ILP1 et ILP2, vous devrez commencer par en faire un *fork* personnel, puis par importer ce nouveau projet dans Eclipse. Vous ajouterez le code de ce TME dans votre *fork* d'ILP3 (vous n'avez pas besoin d'intégrer les TME précédents dans ILP3).

Nous travaillerons dans le *package* `com.paracamplus.ilp3.ilp3tme7`.

2 Travail à réaliser

2.1 Grammaire et AST

L'ajout des coroutines nécessitera l'ajout d'une construction dans la grammaire du langage permettant de créer une instance d'une coroutine. La syntaxe de cette construction est similaire à celle d'un appel de fonction, préfixé du nom-clé `costart` ; ainsi, le fragment de programme suivant est une expression valide :

```
costart f(x,y)
```

Les instructions `yield` et `resume` seront par contre représentées par des primitives (comme `print`), ce qui évite des modifications supplémentaires de la grammaire et de l'AST.

Travail à réaliser : écrivez une grammaire ANTLR `ILPMLgrammar3tme7` qui étend la grammaire d'ILP3 avec la construction `costart` ; étendez également l'AST avec un nœud `ASTcostart`, inspiré de `ASTinvocation`.

2.2 Interprétation

2.2.1 Threads

La notion de coroutine n'existant pas en Java, nous allons l'implanter à l'aide de *threads*. Rappelons que les threads sont des processus légers qui permettent en Java de créer plusieurs unités d'exécution indépendantes, partageant les mêmes objets.

Les instances de coroutines seront donc représentées par une classe `CoroutineInstance` que vous devrez créer, dérivant de `java.lang.Thread`.

2.2.2 Démarrage

L'interprétation de l'instruction `costart f(x,y)` sera de créer un nouvel objet de classe `CoroutineInstance`, de lancer l'exécution de la thread correspondante (par la méthode `start` héritée de `Thread`) et de renvoyer cet objet pour qu'il soit stocké dans une variable du programme. Rappelons qu'un appel à `start` sur une instance de classe dérivant de `Thread` créera une nouvelle unité d'exécution qui exécutera la méthode `run` de cette classe sans interrompre l'appelant.

Travail à réaliser : programmez une classe `CoroutineInstance` dérivant de `Thread` et dont la méthode `run()` lancera l'interprétation de la fonction `f(x,y)`, passée en argument de `costart`.

2.2.3 Flot de contrôle avec des sémaphores

Par défaut, les threads sont exécutées en concurrence, indépendamment les unes des autres. Cependant, les coroutines obéissent à des règles strictes de flot de contrôle : un `resume` met en attente la thread de l'appelant et autorise l'exécution de la thread coroutine ; un `yield` met en attente la thread coroutine et autorise l'exécution de la thread ayant appelé `resume`. Nous devons donc ajouter un mécanisme d'attente entre threads.

Nous suggérons pour cela l'emploi d'objets `java.util.concurrent.Semaphore`. Si un sémaphore est créé avec une valeur initiale nulle, alors un premier appel à `acquire` permettra à une thread de se mettre en attente, jusqu'à ce qu'une autre thread la délivre avec la méthode `release` sur le même sémaphore. Une coroutine aura besoin de deux sémaphores, le premier pour mettre en attente une coroutine jusqu'au prochain `resume`, et le deuxième pour mettre en attente la thread ayant appelé `resume` jusqu'au prochain `yield`.

Travail à réaliser : implantez dans `CoroutineInstance` des méthodes `yieldCoroutine()` et `resumeCoroutine()` modélisant le passage de contrôle de la coroutine à l'appelant, et de l'appelant à la coroutine ; modifiez `run` pour qu'elle attende le premier `resumeCoroutine` avant d'appeler la fonction coroutine. Prenez garde au cas où un `resumeCoroutine` est appelé sur une coroutine ayant terminé son exécution.

2.2.4 Liaison avec l'interpréteur

Il reste à connecter l'interpréteur à la classe `CoroutineInstance`. Lors d'un appel à `resume`, l'instance de coroutine à réveiller est passée en argument (il faudra tout de même vérifier le type de l'objet). Lors d'un appel à `yield`, l'instance de coroutine qui s'interrompt n'est pas passée en argument : c'est implicitement l'instance en cours d'exécution. Pour retrouver la coroutine courante, la méthode statique `Thread.currentThread` pourra être utile...

Travail à réaliser : implantez des classes `Yield` et `Resume` pour les primitives correspondantes ; enrichissez les classes `Interpreter` et `GlobalVariableStuff` pour gérer l'instruction `costart` et les nouvelles primitives.

Comme toujours, vous testerez votre implantation en développant une banque de programmes de test ayant des coroutines, et vous ajouterez vos tests dans l'intégration continue du serveur GitLab du cours (fichier `.gitlab-ci.yml`).

2.3 Compilation

2.3.1 Contextes

Une implantation des coroutines en C est possible via les fonctions `getcontext`, `setcontext`, `makecontext` et `swapcontext` de la bibliothèque standard POSIX, disponible en particulier sous Linux et MacOS X (`ucontext.h`). Ces fonctions manipulent le contexte d'exécution, qui correspond non seulement à la position dans le programme mais aussi à la pile complète. Elles généralisent donc les fonctions `setjmp` et `longjmp` vues en cours. Ces dernières permettent uniquement de « sauter » à un point antérieur de la pile (i.e., un appelant), tandis que les fonctions de contexte permettent de « sauter » entre plusieurs piles, simulant ainsi plusieurs unités d'exécution indépendantes avec leur pile d'appel, leurs variables locales, etc.

La création d'un nouveau contexte nécessite plusieurs étapes :

- allouer un objet `c` de type `ucontext_t` ;
- initialiser `c` avec `getcontext` ;
- allouer un bloc mémoire pour la pile du contexte, et renseigner les champs `uc_stack.ss_sp` et `us_stack.ss_size` ;
- définir la fonction exécutée quand le contexte est actif, avec `makecontext`.

Une fois le contexte créé, on peut le rendre actif avec `setcontext`. La fonction `swapcontext` sera plus utile : elle permet en un seul appel de sauter au contexte donné et de stocker le contexte courant, ce qui permet de revenir plus tard au point d'appel `swapcontext` par un autre `swapcontext` ou un `setcontext`.

2.3.2 Bibliothèque d'exécution

Il est nécessaire d'enrichir la bibliothèque d'exécution C avec la notion d'instance de coroutine et les fonctions pour y accéder.

Travail à réaliser :

- Enrichir le type `ILP_Object` dans `ilp.h` avec un champ coroutine. Celui-ci contiendra, entres autres, des contextes permettant de savoir où sauter après un `yield` ou un `resume`.
- Ajouter à `ilp.c` des primitives `ILP_yield()`, `ILP_resume(coroutine)`.
- Ajouter à `ilp.c` une primitive `ILP_costart(f, argc, ...)` en s'inspirant de `ILP_invoke`. La primitive crée un nouveau contexte appelant la fonction `f` avec les arguments spécifiés et renvoie un nouvel `ILP_Object` correspondant à l'instance de coroutine créée.

Attention à ce qui se passe quand la fonction coroutine se termine. Le comportement par défaut de `swapcontext` est de terminer le programme, ce qui n'est pas souhaitable, mais le champ `uc_link` du contexte permet de changer ce comportement...

2.3.3 Normalisation

Le compilateur effectue une spécialisation de certains nœuds de l'AST ; en particulier, il classe les d'appels `ASTInvocation` en plusieurs types, selon leur complexité : `ASTCcomputedInvocation`, `ASTCglobalInvocation`, etc. Nous supposons ici que l'instruction `costart` est toujours utilisée avec, comme argument, une fonction globale. L'étape de normalisation va naturellement transformer nos `ASTcostart` en nouveaux nœuds `ASTCglobalCostart`, inspirés de `ASTCglobalInvocation`. Nous ignorerons les autres types d'invocation.

Travail à réaliser : ajouter la classe `ASTCglobalCostart` et adapter la classe `Normalizer` pour traiter `ASTcostart`.

2.3.4 Compilation

Finaliser l'adaptation du compilateur est maintenant assez simple.

Travail à réaliser :

- Ajouter la gestion des primitives `yield` et `resume` à `GlobalVariableStuff` (elles se traduisent respectivement en `ILP_yield` et `ILP_resume`).
- Ajouter la gestion du nœud `ASTCglobalCostart` à la classe `Compiler`. Il est possible de s'inspirer de la gestion de `IASTCglobalInvocation` de `Compiler`, sauf que l'appel à `ILP_invoke` est remplacé par un appel à `ILP_costart`.

2.4 Rendu

Vous effectuerez un rendu en vous assurant que tout le code développé a été envoyé sur le serveur GitLab (*push*) dans votre *fork* d'ILP3. Vous vous assurerez que les tests d'intégration continue développés ont bien été configurés et fonctionnent sur le serveur. Vous ajouterez un tag « rendu-initial-tme7 » en fin de séance, puis un tag « rendu-final-tme7 » quand le TME est finalisé.