Piton, un compilateur de pseudo-Python

Olivier Birot
LIEC - Licence 3 Informatique
Université Paris 8

4 janvier 2018

1 Introduction

Piton est une contrefaçon du langage Python et un compilateur de ce langage, implémenté en Racket. La syntaxe en est la même mais les annotations de type, facultative en Python, sont obligatoires. C'est un langage statiquement et fortement typé à portée lexicale statique. Il n'inclut aucune notion d'héritage ni de structure de classes. Il est compilé vers de l'assembleur MIPS.

Comme le langage Piton est destiné à être compilé, il est bien plus simple qu'il soit statiquement typé afin de ne pas avoir à embarquer un environnement d'exécution à programmer en assembleur. Depuis Python 3.6, il est possible d'annoter les variables, les paramètres des fonctions et leur valeur de retour afin de renseigner leur type. Cette syntaxe est reprise par notre Piton afin que le type exact de chaque symbole puisse être connu à la compilation.

Un programme de test est mis à disposition, il est compilable avec la commande *racket piton.rkt poetry-generator.py* qui fournit en sortie un fichier *out.s* exécutable avec Spim. Un jeu de tests peut être lancé avec le script *run-tests.py* (nécessite Python 3.6) ainsi qu'un jeu de tests d'erreurs avec *run-error-tests.py*.

2 Langage

Contrairement à Python où un type peuvent être désigné par un symbole résolu à l'exécution, Piton impose une liste predéfinie de types dès l'analyse syntaxique. Seul sont

supportés quelques types scalaires : *int*, *bool*, *string*. Les entiers et les booléens sont assignés par copie, les chaîne de caractères par référence, ce qui ne pose pas problème car elles sont immutables. Il n'y a pas de notion d'héritage, un booléen n'hérite pas d'un entier. Les listes et les dictionnaires ne sont pas implémentés.

Les fonctions occupent le même espace de nom que les variables, comme dans Python. Le type *function* est une sorte de type composite des types des paramètres et de retour, mais il est interne au compilateur. Il est pas possible de déclarer un symbole de type *function* autrement que par une définition de fonction qui lui attribue sa valeur. Dans Python, il est possible de redéclarer une fonction, ainsi que de déclarer des fonctions au sein de structures *if*, *else* ou *while*. Ce n'est pas le cas dans Piton.

Dans Python, une assignation ou un appel de fonction se font sur une expression. Ainsi, des instructions telles que *(myvar) = 1* ou *(myfunc)()* sont valides. Dans Piton en revanche, c'est directement à un symbole qu'une valeur est assignée, et c'est un symbole qui est appelé. Il est syntaxiquement valide d'appeler un symbole quelque soit son type, mais l'analyse sémantique vérifiera qu'il s'agit d'une fonction.

La résolution de portée dans Python s'effectue selon la règle *LEGB* : *Local*, *Enclosed*, *Global*, *Built-in*. Les symboles declarés au niveau le plus haut du fichier sont globaux. En Piton, l'espace Global n'existe pas. Cependant, tout le code situé au niveau le plus haut du fichier est contenu dans une fonction main implicite, dont la portée fait en quelque sorte office de Global. Néanmoins, il n'est pas possible de faire référence directement au niveau Global dans passer par les portées intermediaires, comme c'est le cas en Python via le mot-clé *global*.

Autre différence, en Python les déclarations de symboles sont implicites : toute variable assignée est déclarée au niveau local. Pour modifier une variable de portée externe, il faut la déclarer via le mot-clé *non-local*. En Piton, comme la déclaration de variable et de fonctions est explicite, ce mécanisme est moins nécessaire, et il est possible de modifier directement une variable de porte externée. Cependant, il n'est pas possible de modifier une variable externe qui serait masquée par une variable locale.

3 Implémentation

L'indentation en Python est gérée par l'émission de tokens d'indentation et de désindentation à chaque changement de niveau. Il faut donc être capable d'émettre plusieurs tokens par règle, un retour à la ligne pouvant désindenter plusieurs niveaux à la fois. Comme le lexer de Racket ne le permet pas, il nous a fallu intercepter les tokens émis par le lexer, détecter si nous avions enveloppé plusieurs tokens en un seul et mettre en place une queue afin de pouvoir réémettre l'intégralité des tokens au parser.

L'analyse sémantique repère les déclarations de variables et définitions de fonctions et elle référence leur nom et type dans des table de symboles, une par fonction. Ces tables nous servent à vérifier la validité de toutes les instructions du point de vue du typage, et elles sont construites au fur et a mesure de l'analyse sémantique. Il n'est ainsi pas possible d'assigner une valeur à une variable ou d'appeler des fonctions avant leur déclaration.

Lors de cette passe, les opérations sont transformées en appel de fonction afin de simplifier la compilation, après avoir vérifié qu'il existait une fonction pour l'opérateur dont les types des paramètres correspondent aux types des opérandes. Les accès par subscript aux chaînes de caractères sont également transformés en appels de fonction.

Enfin, une seconde passe vient désimbriquer les appels, c'est à dire ajouter des déclarations et assignations pour les arguments d'un appel qui sont eux-mêmes des appels, ce récursivement. Cela permet ensuite de simplifier la phase de compilation proprement dite.

A la compilation, certains registres sont réservés à certaines opérations. Le registre temporaire *$t9* sert au stockage des résultats des conditions de test des *if* et *while*, ainsi qu'à des opérations nécessitant un registre temporaire immédiatement libéré (mouvement d'une valeur de mémoire à mémoire, par exemple). Les registres *$t8* et *$t7* sont dédiés à la récuperation de variables de portée englobante. Le registre *$sp* nous sert en réalité de frame pointer, il n'est jamais modifié au sein d'une même fonction. C'est le compilateur qui mémorise le nombre de variables présentes sur la stack dans la fonction.

Dans une première étape, les tables de symboles de chaque fonction sont à nouveau reconstruites, cette fois-ci en y mémorisant non pas le type mais l'emplacement, c'est-à-dire l'adresse relative au frame pointer pour une variable, et le label pour une fonction. On mémorise également le niveau d'imbrication de la fonction dans laquelle est déclarée le symbole, une information qui nous sera utile pour la résolution de portée.

Lors des appels de fonctions, l'*activation link* est placé sur la stack après la *return adress*. Il s'agit du frame pointer de la fonction englobante présente en mémoire étant la plus proche de la fonction appelée. Ensuite, les arguments sont ajoutés à la stack plutôt qu'aux registres *$a0* à *$a3*. Certaines fonctions de la bibliothèque standard, en particulier celles qui ont été substituées aux opérations, sont inlinées : elles ne sont pas appelées mais leur code est inséré directement dans les instructions de la fonction appelante. Cela permet d'éviter la lourdeur d'un véritable appel de fonction pour des opérations pouvant souvent se faire en une seule instruction.

A la compilation, la résolution de symboles se fait en consultant les tables de symboles, en partant de la table locale. Si le symbole est trouvée dans une table englobante, les informations disponibles sont son adresse par rapport au frame pointer de la fonction où il est declaré, et le niveau d'imbrication de cette fonction dans le code (0 pour built-ins, 1 pour main, etc). Ainsi, en calculant la différence de niveau entre la fonction locale et la fonction englobante concernée, il est possible d'accéder à sa frame en remontant la chaîne

des *activation links*. La variable est ensuite obtenue grâce à son adresse relative au frame pointer.

4 Limites

Le principal point faible du langage Piton est qu'il est possible d'accéder à des variables non initialisées. Tout d'abord, comme une variable peut être déclarée au sein d'un *if-else*, il est possible d'y accéder même si à l'exécution la branche du *if-else* où elle a été déclarée n'a pas été visitée.

De plus, si le typage des valeurs retournées dans une fonction est bien vérifié, il n'est en revanche pas vérifié qu'une fonction retourne effectivement quelque chose. A nouveau, Piton ne sait pas à la compilation si une branche d'un *if-else* contenant un *return* sera visitée à l'exécution. En conséquence, il se peut que le *return* ne soit jamais appelée et que la fonction n'initialise jamais le contenu de *$v0*, le registre contenant la valeur retournée.

Il serait possible de mettre en place plusieurs mécanismes pour détecter ces situations : valeur par défaut pour les variables non initialisées, instruction d'erreur ajoutée en fin des fonctions supposées retourner une valeur.