TP compilation : compilateurs TYP → REC → TPL → VAR

Le langage TYP est un langage typé, qui étend les langages précédents avec deux formes de structures de données : les tableaux et les n-uplets nommés (`struct`).

On va réaliser ici :

— un vérificateur de types pour TYP,
— une traduction de TYP vers REC, un langage identique en tous points mais sans annotations ni vérification de types,
— une traduction de REC vers TPL, un langage qui n'a ni tableaux ni n-uplets nommés mais seulement une notion primitive de n-uplet,
— une traduction de TPL vers FNX, un langage qui est identique au langage VAR du module précédent à ceci près que l'appel de fonction y est une expression ordinaire,
— une fonction d'allocation de mémoire, écrite en FNX.

La traduction de FNX vers VAR est fournie (mais vous en aviez peut-être réalisé une vous-même, en extension du module précédent).

1 Langage TYP

Types Les types manipulés sont :

— entiers et booléens
— pointeurs
— tableaux
— structures
— fonctions

sachant que le booléen "vrai" est représenté par l'entier 1 et que le booléen "faux" est représenté par l'entier 0.

Syntaxe Pour la syntaxe du langage, voir l'analyseur syntaxique fourni.

Vérification de types La fonction `TYPCheck.check_program` vérifie que le programme passé en paramètre est intégralement bien typé. Il se base notamment sur une fonction `TYPCheck.check_instr` vérifiant qu'une instruction est bien typée et une fonction `TYPCheck.type_expr` vérifiant qu'une expression est bien typée et renvoyant son type.

Traduction vers REC La traduction consiste uniquement à effacer toute mention des types. Cette traduction est fournie.

2 Langage REC

Déclaration de structures Le langage REC permet de déclarer des n-uplets nommés, chacun étant caractérisé par un nom et par une liste de noms de champs.

Utilisation de structures de données Le langage contient deux formes d'instructions pour manipuler les structures de données :

— `MkArray(d, s, e)` crée un nouveau tableau de longueur `s`, dont toutes les cases sont initialisées avec l'élement `e`,
— `NewRec(d, id)` crée un nouveau n-uplet nommé de nom `id` dont les champs ne sont pas initialisés,

sachant que dans les deux cas, un pointeur vers la structure de données créée est stocké à l'adresse `d`.

Le langage contient également deux formes d'expressions associées :

— `ArrayAccess(a, i)` accède à la case d'indice `i` dans le tableau `a`, et interrompt le programme si l'accès de fait en dehors des bornes du tableau,
— `RecAccess(r, id)` accède au champ de nom `id` dans le n-uplet nommé `r`.

Notez que `ArrayAccess` et `RecAccess` sont des expressions gauches (donc similaires à `Name`).

Stratégie de compilation

— Un tableau de taille n est compilé comme un $n+1$-uplet TPL, dont le premier champ contient la taille n. La "valeur" du tableau est un pointeur vers sa première case de contenu, c'est-à-dire la deuxième case du $n+1$-uplet sous-jacent. On pourra créer dans le langage TPL deux fonctions `mk_array` et `array_get` respectivement pour la création d'un tableau et pour l'accès à une case.
— Un n-uplet nommé à n champs est compilé en un n-uplet TPL. Chaque accès à un champ est remplacé par l'indice correspondant à la position de ce champ.

3 Langage TPL

Manipulation des n-uplets Le langage TPL fournit une instruction `MkTuple(d, s)` pour créer un n-uplet dont le nombre n de champs est donné par l'expression `s`, ce n-uplet étant stocké à l'adresse `d`, et une expression `TupleAccess(t, i)` qui donne l'adresse du champ d'indice i du n-uplet t, les indices commençant à 0.

Sémantique de l'égalité Deux structures alloués sur le tas seront considérées égales pour l'opérateur == si et seulement si elles ont la même adresse. On parle donc d'*égalité physique*.

Stratégie de compilation Un n-uplet est représenté par n cases consécutives dans le tas, et sa "valeur" est un pointeur vers la première case (celle d'indice 0). L'allocation de l'espace nécessaire sur le tas sera effectué par une fonction `malloc` à définir dans le langage FNX.

Allocation sur le tas La fonction `malloc` fournie pour l'instant est de type `sbrk`. Elle utilise un pointeur `memory_break` désignant la première adresse libre au-dessus du tas, qu'elle incrémente à chaque allocation.

4 Langage FNX

Le langage FNX est identique au langage VAR muni de l'extension permettant de réaliser des appels de fonctions à l'intérieur d'expressions.

5 Extensions

5.1 Fluidifier la création de structures

5.1.1 Expression `new`

Faire de la création d'un n-uplet ou d'un tableau une expression ordinaire.

5.1.2 Structures initialisées

Permettre de créer directement une structure initialisée, par exemple avec l'une des syntaxes suivantes :

— `x := (1, 2, 3)` pour un n-uplet TPL,
— `x := [1; 2; 3]` pour un tableau,

— `x := {a = 1; b = 2}` pour un n-uplet nommé.

Attention, combiner ces extension et faire une expression `[1; 2; 3]` n'est pas immédiat dans l'architecture actuelle du compilateur.

5.2 Égalité structurelle

Introduire un opérateur = d'égalité structurelle, c'est-à-dire pour lequel deux structures sont égales si elles ont des contenus égaux (les contenus étant eux-mêmes comparés à l'aide de cette égalité structurelle).

Indication : les comparaisons qui devront être effectuées vont dépendre du type des données comparées. Cet opérateur va donc être introduit dans le langage TYP, et sera traduit en opérations de comparaison élémentaires lors de la traduction de TYP vers REC. Vous pouvez notamment générer des fonctions REC réalisant les comparaisons.

5.3 Inclusion de code FNX en syntaxe concrète

Pour simplifier les extensions suivantes qui reviennent à inclure des fonctions FNX prédéfinies dans l'AST produit par `TPLtoFNX.translate_program`, inclure dans votre compilateur une fonction auxiliaire permettant d'inclure un fragment de code FNX en syntaxe concrète (donné sous la forme d'une chaîne de caractères).

5.4 Gestion de la mémoire avancée

5.4.1 Allocation et désallocation manuelle

Écrire une fonction `malloc` plus intelligente que la fonction fournie dans le squelette, utilisant par exemple une liste de bloc libres, ainsi qu'une fonction `free` associée.

La fonction actuelle ou une variante peut être utilisée lorsque les blocs déjà présents ne permettent pas de résoudre l'allocation demandée.

5.4.2 Gestion automatique de la mémoire

Écrire des fonctions réalisant une récupération automatique de mémoire de type *Mark & Sweep*. Lorsqu'un appel à `malloc` échoue, faire qu'une telle phase de récupération soit lancée avant d'effectuer une extension du tas.