Programmation Fonctionnelle : des concepts aux applications web (NFP119)

Sujet du projet

Traduction d'un mini-Python vers TypeScript/JavaScript et développement d'applications web basées sur React

28 novembre 2024

Le but de ce projet est d'implanter un traducteur pour un fragment fonctionnel de Python (que nous appellerons mini-Python) vers TypeScript/JavaScript (norme ES2020). La documentation sur TypeScript, est une version statiquement typée de JavaScript, est disponible ici :

http://www.typescriptlang.org

Les constructions du langage mini-Python contiennent les fonctions de première classe, la généricité, les listes, les enregistrements, les types récursifs et le filtrage (et éventuellement aussi les exceptions). Un programme mini-Python donné en entrée au traducteur sera supposé syntaxiquement correct et bien typé.

Ce traducteur devra être écrit en Python en complétant les sources fournis (qui contiennent la définition de la syntaxe abstraite et un analyseur syntaxique pour mini-Python). Un exemple d'utilisation du traducteur pour développer une application web, basée sur la bibliothèque React, est aussi demandé. La documentation de React est disponible ici :

https://reactjs.org

Modalités d'évaluation du projet. Une soutenance de 10–15 minutes aura lieu pendant la dernière séance de TP du semestre. Les sources du projet seront rendu avant la veille de cette soutenance. Une démonstration est aussi à préparer, avec des examples originaux et pertinents.

Remarque. Le sujet complet et les sources fournis pour le projet sont disponibles sur la page Moodle de l'UE. Ces sources peuvent être exécutés avec une version récente de Python ($\geqslant 3.12$). Pour ce projet, il faut aussi installer :

- node.js: machine vituelle pour executer du code JavaScript ES2020 (version $\geqslant 12$)
- npm: Node Package Manager (version $\geqslant 6$)

Pour le développement, il est recommandé d'utiliser VS Code Studio avec l'extension Python officielle (le support pour TypeScript est déjà installé par défaut, mais pas le compilateur).

Présentation détaillée

Le langage mini-Python considéré correspond à un fragment simple, purement fonctionnel, du langage Python. Les types de données primitifs autorisés sont int, bool, string, et les listes génériques. L'utilisateur doit de plus pouvoir définir des types enregistrement et des types récursifs (éventuellement génériques aussi).

Un programme mini-Python est composé d'une série de déclarations de types, de valeurs et de fonctions (récursives ou non) qui se termine éventuellement par une expression.

Voici un exemple de programme mini-Python pour les fonctions *size* et *concat* (fichier concat.py) :

```
def size[A](l: list[A]) -> int:
    match l:
        case [_, *t]:
            return 1 + size(t)
        case _:
            return 0
```

```
def concat[A](l1: list[A], l2: list[A]) -> list[A]:
    match l1:
        case [h, *t]:
            return [h, *concat(t, l2)]
        case _:
            return l2


l1 = [5, 4, 3, 2, 1]
l2 = [6, 7, 8, 9, 0]
r = concat(l1, l2)
```

1 Syntaxe concrète

La syntaxe concrète de mini-Python correspond à un fragment de celle de Python. Les exemples fournis doivent suffire à expliciter le sous-ensemble considéré. Les points à noter sont les suivants :

- Toutes les fonctions, récursives ou non, doivent être typées : les types des arguments et le type retour doivent être explicites.
- Toutes les déclarations de types doivent être placées au début du fichier.
- Les définitions de variables peuvent être soit *typées* soit *non typées*.
- Le fichier peut terminer, ou non, par une expression.

2 Syntaxe abstraite

La syntaxe abstraite de mini-Python est donnée en annexe. Les sources de l'analyseur lexical et de l'analyseur syntaxique sont donnés. Pour les utiliser, vous devez simplement appeler une des deux fonctions définies dans le module *parsers* :

- `parse_from_file(filename: str) -> prog`
- `parse_with_regions(filename: str) -> prog`

Cette dernière fonction stocke aussi les *régions* dans l'arbre de syntaxe abstraite. Ces régions permettent de localiser une expression dans le fichier source et donc d'afficher un message d'erreur utile au développeur. Afin d'être reconnue depuis le terminal de VS Code, une région d'un fichier source `test1.py` sera affichée sous la forme suivante :

```
File "test1.py", line 9-11, column 26-7
```

1 Traduction vers Typescript

1.1 Traduction des fonctions

La traduction des fonctions utilise autant que possible les constructions de TypeScript qui correspondent aux constructions de mini-Python.

Voici quelques exemples de fonctions mini-Python et leur traductions en Typescript :

Syntaxe mini-Python (fichier test.py)

```
from typing import Callable


def f1(x: int) -> int:
    return x * 2


def f2(x: int, y: int) -> int:
    if x < y:
        return x
    else:
        return y
```

```
def f3(x: int) -> Callable[[int], int]:
    return (lambda y: y + x)


def f4(f: Callable[[int], bool], g: Callable[[bool], str]) -> Callable[[int], str]:
    return (lambda x: g(f(x)))


def f5[A, B, C](f: Callable[[A], B], g: Callable[[B], C]) -> Callable[[A], C]:
    return (lambda x: g(f(x)))
```

Syntaxe Typescript (fichier test.ts)

```
import * as stdlib from './stdlib.js'


function f1(x: number): number {
    return x * 2
}

function f2(x: number, y: number): number {
    if (x < y) {
        return x
    } else {
        return y
    }
}

function f3(x: number): (_: number) => number {
    return y => (y + x)
}

function f4(f: (_: number) => boolean, g: (_: boolean) => string): (_: number) => string {
    return x => g(f(x))
}

function f5<A, B, C>(f: (_: A) => B, g: (_: B) => C): (_: A) => C {
    return x => g(f(x))
}
```

Remarque. En TypeScript, un type fonction doit mentionner les noms des arguments. Comme ces noms ne sont pas pertinents, nous utiliserons systématiquement la variable « _ ».

Syntaxe Typescript pour les listes.

Les listes de Python sont traduites vers les tableaux de TypeSript, en utilisant la syntaxe « *spread* » pour les primitives. Le fichier concat.py se traduirait par exemple ainsi (fichier concat.ts) :

```
import * as stdlib from './stdlib.js'


function size<A>(l: A[]): number {
    if (l.length > 0) {
        const [, ...t] = l
        return 1 + size(t)
    } else {
        return 0
    }
}
```

```
function concat<A>(l1: A[], l2: A[]): A[] {
    if (l1.length > 0) {
        const [h, ...t] = l1
        return [h, ...concat(t, l2)]
    } else {
        return l2
    }
}
const l1 = [5, 4, 3, 2, 1]
const l2 = [6, 7, 8, 9, 0]
const r = concat(l1, l2)
```

1.2 Traduction des types récursifs

Comme en Python, la traduction des types récursifs est un type *union* de plusieurs types *record* (un type *record* par constructeur du type récursif).

Chacun de ces types *record* sera implanté par une interface qui contiendra en plus un champ appelé *kind*, typé par le nom du constructeur. On génère de plus une fonction pour chaque contructeur, qui prends des arguments du bon type et construit le *record*[1].

Le filtrage est traduit par une instruction *switch* en TypeScript. Ce *switch* est appliqué sur le champ *kind*, et le compilateur TypeScript vérifie que la valeur du *kind* est correcte, et en déduit ainsi la liste des attributs valides. Une déclaration « destructurante » permet ensuite d'y accéder.

Voici l'exemple des arbres binaires en mini-Python :

```
from dataclasses import dataclass

type tree[A, B] = Leaf[A] | LNode[A, B]


@dataclass
class Leaf[A]: value: A


@dataclass
class LNode[A, B]: value: B; left: tree[A, B]; right: tree[A, B]


def max(x: int, y: int) -> int:
    if x > y:
        return x
    else:
        return y


def size(t: tree[int, str]) -> int:
    match t:
        case Leaf(value=_):
            return 1
        case LNode(value=_, left=l, right=r):
            return 1 + max(size(l), size(r))


t = LNode("ok", Leaf(3), Leaf(5))
```

1. Une alternative (en annexe) consiste à remplacer l'interface et la fonction par une classe avec un constructeur.

Et voici le code TypeScript correspondant :

```
import * as stdlib from './stdlib.js'


type tree<A, B> = Leaf<A> | LNode<A, B>

interface Leaf<A> {
    kind: 'Leaf'
    value: A
}

function Leaf<A>(value: A): Leaf<A> {
    return { kind: 'Leaf', value: value }
}

interface LNode<A, B> {
    kind: 'LNode'
    value: B
    left: tree<A, B>
    right: tree<A, B>
}

function LNode<A, B>(value: B, left: tree<A, B>, right: tree<A, B>): LNode<A, B> {
    return { kind: 'LNode', value: value, left: left, right: right }
}

function max(x: number, y: number): number {
    if (x > y) {
        return x
    } else {
        return y
    }
}

function size(t: tree<number, string>): number {
    switch (t.kind) {
        case 'Leaf':
            return 1
        case 'LNode':
            const { left: l, right: r } = t
            return 1 + max(size(l), size(r))
    }
}
const t = LNode("ok", Leaf(3), Leaf(5))
```

2 Extensions

En plus des constructions décrites ci-dessus, et en particulier après avoir traité les difficultés mentionnées dans les remarques, on pourra considérer les améliorations et extensions suivantes :

1. Etendre mini-Python avec le traitement des exceptions (**raise** et **try**/**except**).

2. Etendre la bibliothèque standard de mini-Python (pour autoriser des fonctions prédéfinies).

3 Compilation et exécution des exemples TypeScript

Un programme exemple concat.ts peut être compilé ainsi (l'option --strict active un maximum de contrôles et l'option --target détermine la version JavaScript du code généré) :

```
tsc --strict --target ES2020 concat.ts
```

Le programme concat.js ainsi généré peut ensuite être exécuté par node :

```
node concat.js
```

TypeScript utilise toutefois le système de module du standard EcmaScript, qui n'est pas celui utilisé par défaut par node. Pour accéder à un module, comme stdlib.js, il faut ajouter dans le répertoire un fichier package.json de configuration du projet contenant la ligne avec "type": "module", par exemple :

```
{
  "name": "test",
  "version": "1.0.0",
  "type": "module"
}
```

4 Développement d'une application web basée sur React

Le développement d'une application web basée sur la bibliothèque React suppose que l'état global de l'application soit immutable et que l'ensemble de ses mises à jour se fasse en utilisant uniquement du code purement fonctionnel. Le tutoriel qui décrit cette méthode se trouve là :

> https://react.dev/reference/react/hooks

L'objectif de cette partie du projet est d'illustrer l'utilisation du traducteur pour développer une application qui utilise le *hook* useReducer. En effet, tout le code purement fonctionnel utilisé par useReducer peut être écrit en mini-Python, puis traduit automatiquement en TypeScript. Pour simplifier, nous partirons de l'application « Counter » décrite dans le même tutoriel :

> https://react.dev/learn/typescript#typing-usereducer

Annexe A. Syntaxe abstraite de mini-Python

```
from typing import Union
from dataclasses import dataclass
from error import region

type ident = str

type typ = Union[
    AnyType, # pass
    NoneType, # pass
    IntType, # pass
    BoolType, # pass
    StrType, # pass
    ListType, # ty: typ
    TupleType, # tys: list[typ]
    UnionType, # tys: list[typ]
    FunType, # argtypes: list[typ]; returns: typ
    ParamType, # id: ident; tys: list[typ]
    TypeName, # id: ident
]

type exp = Union[
    NoneCst, # pass
    IntCst, # value: int
    StrCst, # value: str
    BoolCst, # value: bool
    Var, # id: ident
    Plus, # left: exp; right: exp
    Times, # left: exp; right: exp
    Minus, # left: exp; right: exp
    Is, # left: exp; right: exp
    Equal, # left: exp; right: exp
    Less, # left: exp; right: exp
    Greater, # left: exp; right: exp
    And, # left: exp; right: exp
    Or, # left: exp; right: exp
    Not, # operand: exp
    Call, # func: ident; args: list[exp]
    CallExt, # module: ident; func: ident; args: list[exp]
    Lambda, # args: list[ident]; body: exp
    Tuple, # exps: list[exp]
    List, # exps: list[exp]
    Spread, # operand: exp
    DataCons, # id: ident; exps: list[exp]
    Field, # id: ident; attr: ident
    ExpRegion, # contents: exp; reg: region
]

type binding = list[tuple[ident, ident]]
```

```
type comm = Union[
    IfThenElse,   # test: exp; body: block; orelse: block
    MatchList,   # subject: exp; ifempty: block; hd: ident; tl: ident; orelse: block
    MatchData,   # subject: exp; cases: list[tuple[ident, binding, block]]
    Return,   # value: exp
    Raise,   # exn: ident; exps: list[exp]
    TryExcept,   # body: block; exn: ident; name: ident; handler: block
    CommRegion,   # contents: comm; reg: region
]

type block = tuple[list[decl], comm]

type decl = Union[
    Import,   # module: ident
    ImportFrom,   # module: ident; names: list[ident]
    InitVar,   # id: ident; value: exp
    InitVars,   # ids: list[ident]; value: exp
    TypedVar,   # id: ident; ty: typ; value: exp
    FunDef,   # id: ident; tps: list[ident]; args: list[tuple[ident, typ]]; ret: typ; body: block
    DataClass,   # id: ident; tps: list[ident]; fields: list[tuple[ident, typ]]
    TypeAlias,   # id: ident; tps: list[ident]; ty: typ
    DeclRegion,   # contents: decl; reg: region
]

type prog = tuple[list[decl], exp | None]
```

Annexe B. Traduction alternative pour les dataclasses

```
type tree<A, B> = Leaf<A> | LNode<A, B>

class Leaf<A> {
    readonly kind = 'Leaf'
    constructor(readonly value: A) {}
}

class LNode<A, B> {
    readonly kind = 'LNode'
    constructor(readonly value: B, readonly left: tree<A, B>, readonly right: tree<A, B>) {}
}

function max(x: number, y: number): number {
    if (x > y) {
        return x
    } else {
        return y
    }
}

function size(t: tree<number, string>): number {
    switch (t.kind) {
        case 'Leaf':
            return 1
        case 'LNode':
            const { left: l, right: r } = t
            return 1 + max(size(l), size(r))
    }
}

const t = new LNode("ok", new Leaf(3), new Leaf(5))
```