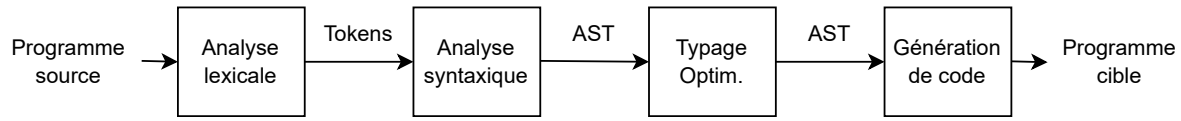
Projet de compilateur

Partie 3 : Vérification des types

Cours Turing

Introduction

Dans cette étape, nous allons faire la vérification des types des expressions qui composent les programmes de notre langage. Pour cela, nous allons devoir traverser l'arbre syntaxique abstrait (AST) des programmes (produits par le parser de l'étape précédente) et vérifier que les types des expressions sont compatibles avec les opérations à effectuer.



Techniquement, nous allons ajouter une méthode `type` à la classe `Expression` contenue dans le fichier `trees.py` et l'implémenter pour chaque sous-classe de `Expression`. La méthode aura un paramètre pour l'environnement de typage, qui contiendra les types des variables déclarées dans le contexte de l'expression. La méthode retournera le type de l'expression, ou lancera une exception de type `TypeError` si la vérification de type échoue.

```
class Expression:

    # Reste du code de la classe Expression

    def type(self, env):
        raise NotImplementedError("type method not implemented")
```

1 Types et environnement de typage

Les fichiers fournis contiennent déjà une structure de donnée pour représenter les types. Vous trouverez la définition des types dans le fichier `typechecking.py`. Une présentation du contenu de ce fichier est donnée ci-dessous.

Dans `ALAN`, nous avons deux constructions pour les types : les types simples et les types de fonction.

1.1 Types simples

Les types simples sont les types de base de notre langage. Nous avons les types suivants :
— `int` : pour les entiers
— `str` : pour les chaînes de caractères
— `bool` : pour les booléens
— `unit` : pour indiquer l'absence de valeur utile

Ces types sont tous représentés à l'aide de la classe `SimpleType` qui est définie dans le fichier `typechecking.py`. Le constructeur de cette classe prend un seul argument, le nom du type, sous la forme d'une chaîne de caractères.

Additionnellement, `typechecking.py` contient des constantes pour les quatre types simples de notre langage :

```
INT_TYPE = SimpleType("int")
STR_TYPE = SimpleType("str")
BOOL_TYPE = SimpleType("bool")
UNIT_TYPE = SimpleType("unit")
```

1.2 Types de fonction

En plus des types simples, notre langage supporte les types de fonction. Les types de fonctions sont caractérisés par la liste des types des arguments et le type de retour de la fonction. Vous trouverez, dans `typechecking.py`, une classe `FunctionType` pour représenter les types de fonction.

Le constructeur de `FunctionType` prend deux arguments : `arg_types` et `return_type`. `arg_types` est une liste de types, représentant les types des arguments de la fonction, et `return_type` est le type de retour de la fonction.

```
class FunctionType(Type):
    def __init__(self, arg_types, return_type):
        self.arg_types = arg_types
        self.return_type = return_type
```

1.3 Environnement de typage

L'environnement de typage est une collection qui contient les types des variables déclarées dans le contexte de l'expression où une expression doit être analysée.

Nous avons ainsi une classe `TypeEnv` pour représenter l'environnement de typage. Cette classe offre les méthodes suivantes :
— `set` : pour ajouter une variable à l'environnement. La méthode prend deux arguments : le nom de la variable et son type.

— `get` : pour récupérer le type d'une variable dans l'environnement. La méthode prend un seul argument, le nom de la variable. La méthode retourne le type de la variable, ou lance une exception de type `KeyError` si la variable n'est pas déclarée dans l'environnement.
— `extend` : pour créer un nouvel environnement de typage qui hérite des types de l'environnement actuel. La méthode ne prend aucun argument.

Comme pour l'environnement de valeurs que nous avions dans notre interpréteur de la partie 0 du projet, l'environnement de typage est extensible localement. Lors de l'analyse d'un bloc de code, il est possible d'ajouter des variables locales à l'environnement de typage, qui seront visibles uniquement dans ce bloc. On utilise pour cela la méthode `extend`.

2 Instructions détaillées

Ci-dessous, vous trouverez les détails de l'implémentation de la vérification des types pour chaque type d'expression de notre langage.

2.1 `Literal`

Le type d'une expression `Literal` dépend du type (Python) de la valeur littérale qu'elle contient :

— Si la valeur littérale est un entier, le type de l'expression est `INT_TYPE`.
— Si la valeur littérale est une chaîne de caractères, le type de l'expression est `STR_TYPE`.
— Si la valeur littérale est un booléen, le type de l'expression est `BOOL_TYPE`.
— Si la valeur littérale est `None`, le type de l'expression est `UNIT_TYPE`.
— Si la valeur littérale a un autre type, une exception de type `TypeError` est levée.

En Python, pour vérifier si une valeur a un certain type, on peut utiliser la fonction `isinstance`, comme dans l'exemple ci-dessous :

```
if isinstance(value, int):
    print(f'{value} est un entier')
```

2.2 `Variable`

Le type d'une expression `Variable` est le type de la variable dans l'environnement de typage. Si la variable n'est pas déclarée dans l'environnement, une exception de type `TypeError` doit être levée.

Pour capturer une exception de type `KeyError`, on peut utiliser un bloc `try except` comme dans l'exemple ci-dessous :

```
try:
    var_type = env.get(self.name)
except KeyError:
    raise TypeError(f"Variable {self.name} is not declared")
```

2.3 `Assignment`

Une expression de type `Assignment` a pour type le type de la variable assignée. Si la variable n'est pas déclarée dans l'environnement, une exception de type `TypeError` doit être levée.

Le type de l'expression qui donne la valeur à la variable doit être égal au type de la variable. Si les types ne sont pas les mêmes, une exception de type `TypeError` doit être levée.

Pour obtenir le type de l'expression qui donne la valeur à la variable, on peut appeler la méthode `type` de l'expression de manière récursive.

2.4 `Primitives`

Les expressions `Primitives` ont un type qui dépend de l'opération effectuée. Voici les types pour chaque opération primitive, données sous la forme d'un dictionnaire Python.

```
PRIM_TYPES = {
    "PLUS": FunctionType([INT_TYPE, INT_TYPE], INT_TYPE),
    "MINUS": FunctionType([INT_TYPE, INT_TYPE], INT_TYPE),
    "TIMES": FunctionType([INT_TYPE, INT_TYPE], INT_TYPE),
    "DIV": FunctionType([INT_TYPE, INT_TYPE], INT_TYPE),
    "MOD": FunctionType([INT_TYPE, INT_TYPE], INT_TYPE),
    "LT": FunctionType([INT_TYPE, INT_TYPE], BOOL_TYPE),
    "GT": FunctionType([INT_TYPE, INT_TYPE], BOOL_TYPE),
    "LTE": FunctionType([INT_TYPE, INT_TYPE], BOOL_TYPE),
    "GTE": FunctionType([INT_TYPE, INT_TYPE], BOOL_TYPE),
    "EQ": FunctionType([INT_TYPE, INT_TYPE], BOOL_TYPE),
    "NEQ": FunctionType([INT_TYPE, INT_TYPE], BOOL_TYPE),
    "NOT": FunctionType([BOOL_TYPE], BOOL_TYPE),
    "CONCAT": FunctionType([STR_TYPE, STR_TYPE], STR_TYPE),
    "PRINT": FunctionType([STR_TYPE], UNIT_TYPE),
    "INPUT": FunctionType([STR_TYPE], STR_TYPE),
    "STR_TO_INT": FunctionType([STR_TYPE], INT_TYPE),
    "INT_TO_STR": FunctionType([INT_TYPE], STR_TYPE),
}
```

Il pourrait être judicieux d'inclure ce dictionnaire dans le fichier `primitives.py` puis de l'importer dans `typechecking.py`.

2.5 `Block`

Le type d'un bloc d'instructions est le type de la dernière instruction du bloc. Si le bloc est vide, le type est `UNIT_TYPE`.

À noter que toutes les expressions du bloc doivent être analysées pour vérifier les types, même si le type de l'expression n'est pas utilisé.

Dans le cadre des expressions d'un bloc, l'environnement de typage doit être étendu pour inclure les variables locales au bloc.

2.6 `IfThenElse`

Le type d'une expression `IfThenElse` est le type du bloc `then` et du bloc `else`. Les deux blocs doivent avoir le même type. Si les types des deux blocs ne sont pas les mêmes, une exception de type `TypeError` doit être levée.

De plus, le type de l'expression conditionnelle doit être `BOOL_TYPE`. Si le type de l'expression conditionnelle n'est pas `BOOL_TYPE`, une exception de type `TypeError` doit être levée.

2.7 `While`

Le type d'une expression `While` est `UNIT_TYPE`. L'expression conditionnelle doit être de type `BOOL_TYPE`. Si le type de l'expression conditionnelle n'est pas `BOOL_TYPE`, une exception de type `TypeError` doit être levée.

Le corps de la boucle doit être analysé pour vérifier les types, et ce même si le type de l'expression n'est pas utilisé par la suite.

2.8 `Abstraction`

Le type `Abstraction` représente des fonctions anonymes. Le type d'une `Abstraction` est un `FunctionType` dont les arguments sont les types des paramètres de la fonction et le type de retour est le type du corps de la fonction.

Lors de l'analyse du type du corps de la fonction, l'environnement de typage doit être étendu localement pour inclure les paramètres de la fonction.

2.9 `Application`

Finalement, dans le cadre d'une expression `Application`, le type de l'expression est le type de retour de la fonction appliquée.

Le type des arguments de l'application doit correspondre aux types des paramètres de la fonction. Cela veut notamment dire que le nombre d'arguments doit être égal au nombre de paramètres de la fonction. Si les types ne correspondent pas, une exception de type `TypeError` doit être levée.

3 Améliorations

Les étapes listées ci-dessus sont les étapes minimales pour la vérification des types dans notre langage. Cependant, il est possible d'ajouter des améliorations à votre implémentation. Ci-dessous sont listées quelques idées d'améliorations que vous pourriez apporter à votre code.

3.1 Tester votre implémentation

La première amélioration concrète que vous pouvez apporter au code est d'ajouter des tests pour vérifier votre implémentation. Pour vérifier que votre implémentation est correcte, il est recommandé de tester votre code avec des programmes qui utilisent les différentes constructions de notre langage.

Pour ce faire, construisez des programmes corrects et incorrects pour tester les différentes constructions de notre langage. Assurez-vous que votre vérification de type fonctionne correctement pour les programmes corrects et qu'elle lève des exceptions pour les programmes incorrects. Vous pouvez vous inspirer des tests fournis dans la partie 0 du projet pour écrire vos propres tests.

3.2 Inclure la position des erreurs

Lorsque vous levez une exception de type `TypeError`, vous pouvez inclure la position dans le code source où l'erreur a été détectée. Cela permettra à l'utilisateur de votre langage de localiser plus facilement l'erreur, et idéalement de la corriger plus rapidement.

Pour ce faire, il vous faudra ajouter des informations sur la position de l'expression dans l'AST. Pour cela, vous pouvez vous inspirer de ce qui est fait pour les tokens.

Faites ensuite en sorte de renseigner ces informations lors de la construction de l'AST et de les utiliser lors de la levée d'une exception de type `TypeError`.