



Killian GRICOURT  
INFO  
Rapport de stage 5<sup>e</sup> année

# Utilisation du typage graduel pour améliorer la sécurité des logiciels

Tome Principal  
ET  
Annexe

2023-2024  
Avril-Septembre

# Sommaire

Sommaire.....	2
Table des figures.....	3
Remerciements.....	4
1 - Présentation du contexte.....	5
1.1 - Universidade Federal do ABC.....	5
1.2 - Introduction.....	5
1.3 - Sujet.....	7
2 - Travaux et recherches.....	7
2.1 - Typage et lambda-calcul.....	8
2.1.1 - Qu'est-ce que le lambda-calcul ?.....	8
2.1.2 - Syntaxe du lambda-calcul non typé.....	9
2.1.3 - Les éléments de base.....	9
2.1.4 - Syntaxe du lambda-calcul simplement typé.....	10
2.1.5 - Hindley-Milner.....	12
2.1.6 - Intérêts du lambda-calcul.....	12
2.1.7 - Limitations.....	13
2.2 - Application à Python.....	14
2.2.1 - Étude de Pyright.....	14
2.2.2 - Recherche d'améliorations.....	15
2.3 - Typage des paramètre d'une fonction.....	17
2.3.1 - 1er cas : Les types sont compatibles.....	17
2.3.2 - 2e cas: La signature du callsite est incomplète.....	18
2.3.3 - 3e cas: Les types sont incompatibles.....	18
2.3.4 - Implémentation.....	19
2.4 - Résultats et Ouverture.....	21
3 - Conclusion.....	22
3.1 - Retour sur l'expérience.....	22
3.2 - Compétences acquises.....	22
Références.....	24

# Table des figures

Figure 1: Exemple de code typé statiquement en Java

Figure 2: Exemple de code typé dynamiquement en Python

Figure 3: Exemple de code erroné typé dynamiquement en Python

Figure 4: Message d'erreur lors de l'exécution du code de la figure 3

Figure 5: Exemple d'erreur rapporté par Pyright, programme de typage graduel

Figure 6: Règles de typage extraites du livre "Types and Programming Languages"<sup>[1]</sup>

Figure 7: Exemple d'application des règles de typage à une fonction en Python

Figure 8 : Extrait des résultats des tests de conformances sur les principaux analyseurs de type en Python

Figure 9 : Exemple de code pour la figure 10

Figure 10 : Exemple d'AST de Pyright

Figure 11 : Exemple d'une annotation de type en Python

Figure 12 : Exemple d'inférence automatique de Pyright

Figure 13 : Exemple de code non annoté en Python

Figure 14 : Exemple d'échec de l'inférence automatique par Pyright

Figure 15 : Exemple de code erroné non détecté par Pyright

Figure 16 : Exemple de code Python montrant un type satisfaisant plusieurs contraintes

Figure 17 : Exemple de code Python où il est impossible d'inférer le type

Figure 18 : Exemple de code Python où les types sont incompatibles

Figure 19 : Fonction calculant le type à inférer à partir des candidats

Figure 20 : Fonction calculant le type commun à deux types (incomplète)

Figure 21 : Fonction calculant le type commun à deux types (proposition de fonction complète)

Figure 22 : Fonction calculant l'intersection entre deux types

Figure 23 : Exemple de cas sans erreurs

Figure 24 : Exemple de cas avec erreurs (incompatibilité int/str)

## Remerciements

Tout d'abord, je souhaite exprimer ma profonde gratitude à toutes les personnes qui m'ont soutenu et accompagné tout au long de ce stage.

Je tiens particulièrement à remercier Emilio de Camargo Francesquini pour m'avoir offert l'opportunité d'effectuer ce stage au Brésil, et pour m'avoir accompagné avec bienveillance en tant que tuteur tout au long de cette expérience.

Mes sincères remerciements vont également à Jean-François Méhaut, qui m'a suivi de près durant le stage et m'a prodigué des conseils précieux pour la rédaction de ce rapport, en qualité d'enseignant référent.

Je souhaite également exprimer ma reconnaissance à Fabricio Olivetti de França et Mario Leston Rey pour leur aide précieuse sur la partie technique et leur orientation éclairée dans mes recherches.

Enfin, je remercie chaleureusement le personnel et la direction de l'UFABC pour leur accueil et leur soutien durant toute la durée de mon stage.

# 1 - Présentation du contexte

## 1.1 - Universidade Federal do ABC

L'Université fédérale d'ABC est une université au sud de São Paulo au Brésil. Elle est constituée de deux campus. L'un se situe à São Bernardo do Campo et l'autre à Santo André. Mon stage s'est déroulé dans un laboratoire sur le campus de Santo André, où se situe la partie scientifique et informatique de l'université. Supervisé principalement par mon tuteur Emilio Franceschini, j'ai pu échanger avec Fabricio Olivetti de França et Mario Leston Rey, professeurs de l'université, pour m'orienter dans ma recherche.

## 1.2 - Introduction

Lorsque l'on souhaite développer un logiciel, le choix du langage est important. Chacun d'entre eux possède des avantages et des inconvénients sur plein d'aspects (la sécurité, l'efficacité, la simplicité, etc.). Parmi ces choix, on retrouve notamment la manière de typer, statique ou dynamique, et c'est sur cela que nous allons nous concentrer.

Un type est une catégorie de données qui définit les valeurs possibles qu'une variable peut prendre et les opérations qui peuvent être effectuées sur ces valeurs. Par exemple, dans de nombreux langages de programmation, il existe des types de base comme *int*, *string*, *float*, *bool*, etc. Le choix du type d'une variable est important car il détermine comment la mémoire est allouée pour cette variable, comment les opérations sont effectuées, et comment les erreurs sont détectées. Un langage de programmation peut appliquer cette notion de différentes manières, en fonction de la rigueur avec laquelle il impose le respect des types et du moment où il le fait. C'est ici qu'intervient la distinction entre typage statique et dynamique.

Si un langage est typé statiquement, les types des données sont obligatoirement définis manuellement afin que la cohérence de ces derniers soit vérifiée avant l'exécution<sup>[1]</sup>. Cela apporte une meilleure résistance aux bogues lors de l'exécution ainsi qu'une meilleure vitesse d'exécution. En contrepartie, cela rend le développement plus compliqué et potentiellement plus chronophage. Cela demande un effort du développeur pour réfléchir à la cohérence des types lors de l'écriture du code source. De plus, certains programmes peuvent être rejetés lors de la vérification des types alors qu'en pratique, ils pourraient fonctionner. Voici en figure 1, un exemple simple de code Java illustrant le typage statique.

```
public class Main {  
    public static void main(String[] args) {  
        int nombre = 10;  
        String message = "Bonjour";  
    }  
}
```

Figure 1: Exemple de code typé statiquement en Java

Ici, si l'on essaye de réassigner la variable *nombre* avec une valeur de type *String*, le compilateur de Java générera une erreur, car cela viole les règles de typage statique du langage. Cette vérification à la compilation aide à prévenir les bogues qui pourraient survenir si les types étaient mal utilisés à l'exécution. De la même façon, des expressions telles que *String s = "nombre" + 10;* généreront des erreurs car il n'est pas possible de réaliser d'addition entre les types *int* et *String*.

Si un langage est typé dynamiquement, la cohérence des types est vérifiée lors de l'exécution<sup>[10]</sup>. Selon les langages, il peut être imposé de typer explicitement le code quand même. Toutefois, la plupart des langages dynamiques optent pour inférer les types lors de l'exécution. Le développement est donc rendu plus simple et plus libre. Beaucoup de programmes rejetés par un typage statique deviennent valides et on voit souvent l'apparition de structures syntaxiques de plus haut niveau rendant le développement plus rapide. Le développeur peut se permettre plus de liberté quant au typage des variables. Toutefois, cela peut amener à introduire des erreurs liées au type des données qui peuvent n'être découvertes que beaucoup plus tard dans le processus de développement. Voici en figure 2 un exemple de code Python illustrant le typage dynamique :

```
def addition(list):  
    return list[0] + list[1]  
  
print(addition([5, 10]))  
print(addition(["Hello, ", "world!"]))  
print(addition([1, 2], [3, 4]))
```

Figure 2: Exemple de code typé dynamiquement en Python

Ici, la fonction *addition* peut être utilisée avec une liste contenant différents types de données (*int, string, list*) sans avoir besoin de spécifier les types à l'avance. Cela rend le développement plus rapide et plus flexible. Cependant, si l'on utilise des types de données incompatibles au sein d'une même liste, comme un entier et une chaîne de caractères, l'addition ne peut plus se faire et générerait une erreur. Cela pourrait être déterminé statiquement si la liste est définie dès le début mais il est tout à fait possible que la liste soit créée pendant l'exécution. L'erreur ne pourrait alors être détectée que lors de l'évaluation de l'expression en question. Par exemple, le programme de la figure 3 ne provoque pas d'erreur avant l'évaluation de l'expression *return list[0] + list[1]*.

```
def addition(list):  
    return list[0] + list[1]  
  
print(addition(["Hello, ", 42]))
```

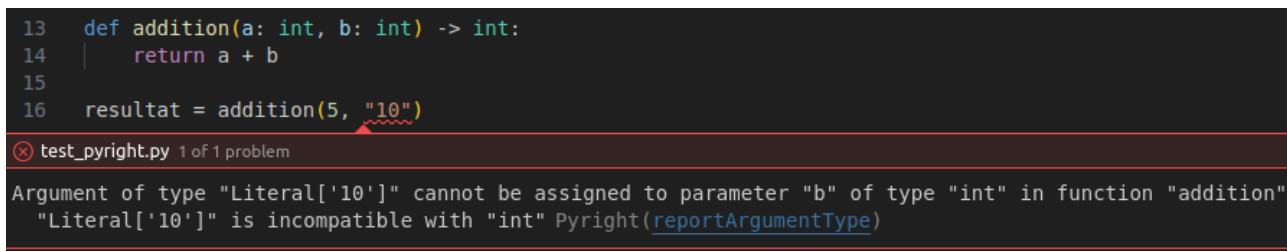
Figure 3: Exemple de code erroné typé dynamiquement en Python

Toutefois, lors de l'exécution, l'erreur en figure 4 est levée.

```
Traceback (most recent call last):  
  File "main.py", line 7, in <module>  
    print(addition([8, "bonjour"]))  
  File "main.py", line 5, in addition  
    return list[0] + list[1]  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Figure 4: Message d'erreur lors de l'exécution du code de la figure 3

Ainsi, il peut être difficile de choisir entre la rapidité et la simplicité de développement ou la stabilité d'un programme. C'est ici qu'intervient le typage graduel. Ce dernier vise à analyser du code typé dynamiquement afin de déterminer statiquement les types des variables avec le moins d'intervention possible du développeur. Ce dernier peut décider de prendre le contrôle de l'analyse et d'annoter les types à sa convenance pour orienter la vérification. Si des erreurs de type sont présentes, le développeur en est informé et peut les corriger aux endroits concernés directement. En passant par du typage graduel, on choisit quelles parties doivent être analysées statiquement en les annotant et lesquelles doivent être laissées dynamiques. De plus, certaines extensions de typage graduel permettent d'utiliser des types spéciaux permettant aux variables de prendre n'importe quel type et donc de contourner l'analyse statique. Voici en figure 5, un exemple de code Python avec l'extension pour le typage graduel PyRight<sup>1</sup> :



```
13 def addition(a: int, b: int) -> int:
14     return a + b
15
16 resultat = addition(5, "10")
```

test\_pyright.py 1 of 1 problem

Argument of type "Literal['10']" cannot be assigned to parameter "b" of type "int" in function "addition" "Literal['10']" is incompatible with "int" Pyright([reportArgumentType](#))

Figure 5: Exemple d'erreur rapporté par Pyright, programme de typage graduel

Ici, la fonction *addition* est annotée pour accepter deux *int*, et retourner un *int*. Cependant, lors de l'appel de la fonction, le deuxième argument est une *string*. Ainsi, PyRight relève l'erreur détectée grâce aux annotations.

## 1.3 - Sujet

Mon sujet de recherche porte sur l'utilisation du typage graduel pour améliorer la sécurité des logiciels. Le typage graduel combine les avantages des langages à typage statique et dynamique, en permettant d'introduire des annotations de type dans des portions de code tout en laissant d'autres parties dynamiquement typées. L'objectif est d'accroître la robustesse des logiciels en détectant plus tôt les erreurs de type, contribuant ainsi à renforcer la sécurité des logiciels. Pour mettre en œuvre et évaluer cette approche, on a choisi d'appliquer le typage graduel à Python. Mon travail consiste donc à explorer et définir dans quelles mesures nous pouvons utiliser le typage graduel pour améliorer la robustesse des programmes écrits en Python.

## 2 - Travaux et recherches

Nous allons maintenant voir les différentes pistes explorées ainsi que l'évolution des objectifs au fur et à mesure des découvertes. Le typage graduel reposant principalement sur l'inférence et la vérification de la cohérence entre les types, nous commencerons par poser les bases nécessaires pour comprendre ces notions. On a choisi d'aborder ce sujet en passant par le lambda-calcul, car cela sert de base pour comprendre comment les langages peuvent être typés de manière statique ou dynamique. Le typage graduel se situe à mi-chemin entre les deux, permettant de mélanger le typage statique et dynamique au sein d'un même programme. Le typage graduel peut

<sup>1</sup> Pyright : <https://microsoft.github.io/pyright/#/> (Pyright, n.d.)

être vu comme une extension du lambda-calcul simplement typé, où les types peuvent être progressivement ajoutés à des extraits de code non typé. Nous continuerons sur l'exploration de Pyright, le type checker le plus efficace actuellement pour Python<sup>2</sup>, afin de voir dans quelles mesures nous pouvons appliquer le principe du typage graduel pour l'améliorer. L'idée initiale d'explorer le lambda-calcul vient du livre "Types and Programming Languages" de Benjamin C. Pierce<sup>[2]</sup>. Il aborde la notion de typage d'une manière très fonctionnelle et justement basée sur le lambda-calcul.

## 2.1 - Typage et lambda-calcul

Dans cette partie, nous allons voir les bases du lambda-calcul et la manière dont on peut l'utiliser pour typer des expressions. Les notions abordées ici, des bases du lambda-calcul non-typé (paragraphe 2.1.2) jusqu'au lambda-calcul simplement typé (paragraphe 2.1.4), seront utiles pour la suite de la recherche, car elles décrivent un modèle simple pour raisonner sur le typage des expressions. Cela se fait par l'établissement de règles simples et d'algorithmes tel que l'algorithme d'inférence d'Hindley-Milner (*Hindley-Milner Type System*, n.d.)<sup>[4]</sup>. Certaines notions plus avancées seront omises, car elles ne sont pas nécessaires pour comprendre la suite du sujet. Plus d'informations et de détails pourront être trouvés dans l'ouvrage "Types and Programming Languages"<sup>[2]</sup>, sur lequel je base cette partie.

Le lambda-calcul est une notion importante pour la suite de ce rapport car il se situe au cœur des concepts fondamentaux liés aux systèmes de typage dans les langages de programmation. En abordant le lambda-calcul, nous pouvons exploiter ces concepts bien connus pour les intégrer dans des outils de typage modernes tels que Pyright. L'objectif est d'adapter ces notions classiques dans un contexte actuel, en renforçant la sécurité et la robustesse du typage graduel en Python.

### 2.1.1 - Qu'est-ce que le lambda-calcul ?

Le lambda-calcul est un modèle dans lequel tout est défini par des fonctions. Dans sa version la plus pure, nous n'y trouvons aucun des éléments de calcul usuels tels que les nombres entiers, les booléens, etc. Dans ce modèle, une expression ressemble à ceci :

$$\lambda x \rightarrow x$$

Elle introduit une variable  $x$  qui est directement utilisée. Cela représente une fonction identité.

Pour utiliser une fonction, que l'on appellera abstraction, il suffit de l'appliquer à une autre expression. Si l'on considère une expression  $e$  et qu'on lui applique l'abstraction identité, cela s'écrit :

$$(\lambda x \rightarrow x) e$$

---

<sup>2</sup> selon le site de référence pour le typage en Python (Static Typing With Python, n.d.)<sup>[9]</sup>.



et cela résulte simplement en:

$$e$$

Il existe plusieurs types de lambda-calcul tels que le lambda-calcul non typé, le lambda-calcul simplement typé, le lambda-calcul polymorphe, etc. Chacun d'entre eux a des spécificités permettant d'obtenir certains comportements spécifiques. Ici, nous nous concentrerons simplement sur les deux premiers mentionnés, car ils sont suffisants pour illustrer ce dont nous avons besoin. Ces explications font écho à celles données dans "Types and Programming Languages"<sup>[2]</sup>

### 2.1.2 - Syntaxe du lambda-calcul non typé

Pour définir formellement la syntaxe nécessaire pour définir une lambda-expression, nous avons besoin de trois termes différents :

- Une variable, représentée par son nom
- Une abstraction, représentée par l'introduction d'une variable et d'un corps " $\lambda x \rightarrow e$ ". Avec " $x$ " un nom de variable et " $e$ " une lambda-expression quelconque.
- Une application, représentée par deux lambda-expressions dont la première est une abstraction. Exemple : " $(\lambda x \rightarrow x) e$ " avec " $x$ " un nom de variable et " $e$ " une lambda-expression quelconque.

Une lambda-expression ne prend qu'un seul paramètre. Toutefois, il est possible de chaîner plusieurs applications si le résultat de la première est une abstraction. La valeur utilisée dans ce cas est l'expression la plus à gauche. Exemple :

$$\begin{aligned} &(\lambda x \rightarrow (\lambda y \rightarrow x)) e1 e2 \\ &\Leftrightarrow (\lambda y \rightarrow e1) e2 \\ &\Leftrightarrow e1 \end{aligned}$$

Dans la première étape, on remplace les occurrences de  $x$  par  $e1$

Dans la deuxième, on remplace les occurrences de  $y$  par  $e2$ , comme il n'y en a pas,  $e2$  disparaît.

Pour des soucis de lisibilité, nous remplacerons, si nécessaire, les expressions imbriquées  $(\lambda x \rightarrow (\lambda y \rightarrow \dots))$  par une version contractée notée  $(\lambda x. y. \dots)$ .

Ces simples règles serviront de base pour la suite et seront enrichies de quelques éléments pour introduire la notion de typage. Toutefois, il est important de noter que sans définir d'autres éléments, il est possible de simuler une machine de Turing. Cela fait du lambda-calcul un modèle Turing complet, et ainsi, il est possible d'exprimer des idées complexes, de résoudre des problèmes complexes et de concevoir des applications logicielles sophistiquées.

### 2.1.3 - Les éléments de base

Avant de passer au lambda-calcul simplement typé, nous allons avoir besoin d'expressions sensées auxquelles attribuer un type. En effet, même sans attribuer un type directement à une expression, il est possible de représenter simplement des éléments de base tels que des booléens ou des entiers.

Pour définir les booléens, commençons par se demander comment sont utilisés les booléens dans un langage de programmation. Généralement, on utilise les booléens pour faire un choix dans des structures conditionnelles : *if b then e1 else e2*. Si on représente la valeur *true* par  $\lambda t \rightarrow (\lambda f \rightarrow t)$  et *false* par  $\lambda t \rightarrow (\lambda f \rightarrow f)$  on obtient deux expressions qui permettent de faire un choix. En effet, si on applique deux valeurs *e1* et *e2* à ces expressions, on obtient :

$$\begin{aligned} & \text{true } e1 \ e2 \\ & (\lambda t \rightarrow (\lambda f \rightarrow t)) \ e1 \ e2 \\ & \Leftrightarrow (\lambda f \rightarrow e1) \ e2 \\ & \Leftrightarrow e1 \end{aligned}$$

et

$$\begin{aligned} & \text{false } e1 \ e2 \\ & (\lambda t \rightarrow (\lambda f \rightarrow f)) \ e1 \ e2 \\ & \Leftrightarrow (\lambda f \rightarrow f) \ e2 \\ & \Leftrightarrow e2 \end{aligned}$$

Ainsi, cela nous permet de choisir entre les deux branches d'une expression conditionnelle. En effet, en définissant la structure conditionnelle comme ceci :

$$IF = \lambda g. e1. e2 \rightarrow g \ e1 \ e2$$

Considérant *b* comme la garde, *e1* l'expression si *true* et *e2* l'expression si *false*. On a bien le comportement attendu, à savoir :

$$\begin{aligned} IF \ \text{true} \ e1 \ e2 & \Leftrightarrow e1 \\ IF \ \text{false} \ e1 \ e2 & \Leftrightarrow e2 \end{aligned}$$

À remarquer que l'expression *IF* n'est pas nécessaire ici. Nous pouvons tout à fait nous contenter d'utiliser directement les expressions *true* ou *false*. Néanmoins, cela permet d'améliorer la lisibilité des expressions et de mieux comprendre ce qu'il se passe. De plus, nous verrons plus tard que cela aura une importance lors de l'introduction de la notion de type dans ces expressions.

### 2.1.4 - Syntaxe du lambda-calcul simplement typé

Nous allons aborder la notion de type. Pour les représenter, nous utiliserons une lettre en majuscule, généralement  $T$ . Nous représenterons également le type d'une fonction par la notation  $T_1 \rightarrow T_2$  indiquant qu'elle prend une expression de type  $T_1$  en entrée et renvoie une expression de type  $T_2$ .

La syntaxe du lambda-calcul simplement typé est très similaire à celle du non typé. En se basant sur celle-ci, il suffit de rajouter l'information du type attendu dans les abstractions. Nous noterons donc maintenant les abstractions comme ceci :  $\lambda x: T \rightarrow e$  avec  $T$ , le type attendu pour la variable liée  $x$ . Pour utiliser correctement cette nouvelle information, nous allons introduire les trois éléments suivants :

- Contexte : noté  $\Gamma$ , c'est une liste de couples (nom de variable:type), qui permet de stocker l'information des types associés aux variables. Si l'on note  $\Gamma \vdash e$ , cela signifie que l'on considère l'expression  $e$  dans le contexte  $\Gamma$ .
- Règles de typage : ensemble de règles décrivant comment déterminer le type d'une expression.
- Constantes : Valeurs de base dont le type est connu.

Lors du calcul du type d'une expression, le contexte joue un rôle crucial. Comme nous n'évaluons pas les expressions, nous avons besoin de savoir quels types sont associés aux variables liées sans effectuer les applications. Le contexte se construit à partir des annotations contenues dans les abstractions et est utilisé pour déterminer le type du corps de ces dernières. Ce mécanisme est expliqué dans l'image ci-dessous, décrivant les règles de typage pour le lambda-calcul simplement typé et étant tirée de "Types and Programming Languages"<sup>[2]</sup>. Ces règles sont décrites de manière à mettre en évidence les prérequis pour appliquer la règle au-dessus de la barre et le résultat en figure 6.

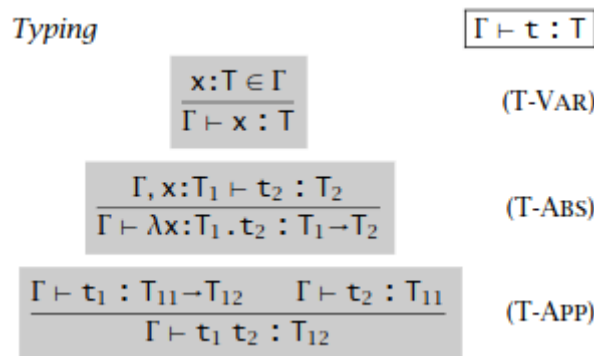


Figure 6: Règles de typage extraites de "Types and Programming Languages"<sup>[2]</sup>

On y voit trois règles, une pour chacune des règles de syntaxe :

- T-VAR exprime qu'un couple  $(x : T)$  appartenant au contexte  $\Gamma$  implique que  $x$  possède le type  $T$  pour ce contexte.
- T-ABS exprime que si une expression  $t_2$  possède le type  $T_2$  dans un contexte  $\Gamma$  où  $x$  est de type  $T_1$ , alors l'abstraction  $\lambda x: T_1 \rightarrow t_2$  est de type  $T_1 \rightarrow T_2$ . Dans les faits, cela signifie que lorsqu'on cherche à obtenir le type du corps d'une abstraction, on se place dans un contexte enrichi de l'information du type de la variable liée.

- T-APP exprime que si deux termes  $t_1$  et  $t_2$  ont respectivement les types  $T_{11} \rightarrow T_{12}$  et  $T_{11}$  dans un contexte  $\Gamma$ , alors l'application est de type  $T_{12}$  dans ce même contexte.

Afin d'illustrer l'utilisation de ces éléments, voici un exemple :

Considérant une expression  $e$  de type  $Int \rightarrow Bool$ , on cherche à déterminer le type de l'expression :

$$(\lambda x: Int \rightarrow e x)$$

C'est une abstraction, donc on applique la règle T-ABS. Cette règle nous dit que l'expression possède un type de la forme  $T_1 \rightarrow T_2$  dans un certain contexte  $\Gamma$  (vide, car nous n'avons pas ajouté d'élément auparavant). On sait  $T_1 = Int$  grâce à l'annotation, mais nous n'avons pas la valeur de  $T_2$ . Nous savons juste que  $T_2$  est le type de l'expression  $e x$  dans le contexte  $\Gamma, (x: Int)$  ( $= (x, Int)$  car  $\Gamma$  est vide).

Nous cherchons donc à déterminer le type de  $e x$  dans le contexte  $(x: Int)$ . On applique donc la règle T-APP. On sait que  $e$  est de type  $Int \rightarrow Bool$  et par le contexte, on sait que  $x$  est de type  $Int$ . Ainsi, on a que l'expression  $e x$  a le type  $Bool$ .

De cette façon, nous pouvons dire que  $T_2 = Bool$  et ainsi conclure que l'expression initiale est de type  $Int \rightarrow Bool$ .

On peut voir que ce principe s'applique très bien à l'exemple python de la figure 7.

```
def f(n:int):
    return n + 2
```

Figure 7: Exemple d'application des règles de typage à une fonction en Python

On peut voir que, connaissant le type du paramètre  $n$ , on peut déduire que  $n + 2$  est type  $int$  et donc que le type de la fonction est  $int \rightarrow int$ .

## 2.1.5 - Hindley-Milner

L'algorithme d'inférence de types Hindley-Milner<sup>[4]</sup> est un algorithme utilisé pour déduire automatiquement les types des expressions dans un langage basé sur le lambda-calcul simplement typé, sans nécessiter que le programmeur spécifie explicitement les types. Partant du lambda-calcul, l'algorithme Hindley-Milner associe un type à chaque expression en analysant sa structure. Il commence par attribuer des variables de type aux expressions, puis génère des équations de type (par exemple, en appliquant une fonction à un argument). Ensuite, il résout ces équations pour trouver un type qui satisfait toutes les contraintes. Si cela réussit, l'algorithme produit un type général, sinon, il détecte une erreur de type. Cela permet de garantir la cohérence des types tout en laissant le plus de liberté possible aux programmeurs.

## 2.1.6 - Intérêts du lambda-calcul

Le lambda-calcul est un modèle simple. Cela permet de n'avoir à considérer qu'un faible nombre de cas lorsque l'on souhaite raisonner, notamment par récurrence structurelle. Si l'on arrive à représenter la structure des types d'un programme par une lambda-expression, il devient très

simple d'en vérifier la cohérence. Pour représenter la structure des types, nous avons juste besoin de récupérer ou déduire les types des variables, des fonctions, et leurs applications. Ensuite, il suffit de faire passer cette représentation dans un programme de vérification des types qui consiste simplement à appliquer les règles une par une jusqu'à ce que toutes les expressions soient typées. De plus, en considérant des algorithmes tels que l'algorithme d'inférence de Hindley-Milner, il serait possible de vérifier la cohérence des types sans avoir besoin de les annoter, car on peut déterminer les types des expressions et sous-expressions. Si l'on ajoute à cela de nouvelles règles telles que le sous-typage, l'héritage, etc... ou des concepts tels que les types Any et Undefined, on se retrouve avec un modèle simple capable de gérer des concepts de haut niveau, suffisant pour prétendre faire du typage graduel. De cette façon, il est très simple de comprendre les mécanismes utilisés. Cela permettra donc par la suite de plus facilement les utiliser sur un langage ayant une syntaxe plus évoluée et permettant des opérations de plus haut niveau. C'est d'ailleurs, ce que nous avons fait dans l'exemple de la figure 7.

De plus, le lambda-calcul simplement typé introduit l'idée que chaque lambda-expression possède un type, et que ces derniers doivent respecter certaines règles pour que l'expression soit correctement typée. Étudier le lambda-calcul permet de comprendre les bases de l'inférence de type, ce qui constitue une partie essentielle du typage graduel. En effet, on peut construire un système où, avec un minimum d'information, il est possible de garantir la cohérence des types de toutes les expressions, sous réserve que ces dernières entrent dans le cadre défini par les règles utilisées pour l'inférence.

### 2.1.7 - Limitations

Bien que la représentation par lambda-calcul soit avantageuse pour sa simplicité et sa flexibilité, elle possède certaines limitations. La représentation de la structure des types en lambda-calcul peut être compliquée. La simplicité de sa syntaxe amène autant de complexité à représenter les éléments d'autres langages. En effet, certains langages manipulent des structures de données complexes qu'il serait difficile de convertir en lambda-expression pure. Bien que cela reste possible, cela prendrait un temps conséquent.

## 2.2 - Application à Python

Python est un langage à typage dynamique. Cela signifie que les vérifications de la cohérence des types sont effectuées uniquement lors de l'évaluation des instructions. Pour permettre aux développeurs d'éviter les erreurs liées aux types avant l'exécution, plusieurs analyseurs existent. Ils parcourent le code et établissent un diagnostic, indiquant les endroits où les types ne correspondent pas. Il en existe quatre principaux: Mypy, Pyright, Pyre et Pytype, selon le site de référence pour le typage en Python<sup>[9]</sup>. Selon les résultats de tests effectués disponible sur le dépôt Git de ce même site<sup>[7]</sup>, nous pouvons voir que Pyright est l'analyseur ayant les meilleurs résultats conformément aux critères définis par Python<sup>[5]</sup>. Voici en figure 8, un extrait des tests réalisés:

Python Type System Conformance Test Results				
	mypy 1.11.1 13.2sec	pyright 1.1.374 1.5sec	pyre 0.9.22 5.4sec	pytype 2024.04.11 35.2sec
<b>Type annotations</b>				
annotations_coroutines	Pass	Pass	Pass	Pass
annotations_forward_refs	Partial	Pass	Partial	Partial
annotations_generators	Partial	Pass	Partial	Partial
annotations_methods	Pass*	Pass*	Pass*	Pass*
annotations_typeexpr	Pass	Pass	Pass	Partial
<b>Special types in annotations</b>				
specialtypes_any	Pass	Pass	Partial	Pass
specialtypes_never	Pass	Pass	Partial	Unsupported
specialtypes_none	Pass	Pass	Pass	Partial
specialtypes_promotions	Pass	Pass	Partial	Pass
specialtypes_type	Partial	Pass	Partial	Partial

Figure 8 : Extrait des résultats des tests de conformances sur les principaux analyseurs de type en Python

Par la suite, nous choisirons donc d'utiliser Pyright en raison de ses performances mais aussi car le code est plus facilement disponible et il possède le soutien de Microsoft. Dans cette partie, nous allons faire l'état des lieux de ce qui peut être amélioré dans Pyright et décrire les solutions que nous pouvons apporter.

### 2.2.1 - Étude de Pyright

Pyright est un programme pouvant être utilisé directement en ligne de commande ou en tant que plugin sur VSCode. Il est codé en TypeScript et prend en entrée un ensemble de fichiers source en Python. Il va ensuite déterminer quels fichiers sont à analyser en fonction des modifications ayant été apportées depuis la dernière vérification, puis va procéder à celle-ci. Pour chaque fichier à analyser, il va lire le code source, le lexer et le parser pour produire un AST (Abstract Syntax Tree)<sup>3</sup>. Chaque instruction du programme est représentée dans l'arbre par un objet Node étant spécialisé en fonction du type d'instruction. Ces objets sont ensuite parcourus par l'implémentation d'un pattern de visiteur et les règles sont appliquées en fonction du type de Node. Voici un exemple d'AST simplifié représentant une fonction additionnant deux nombres :

```
def add(a, b):
    c = a+b
    return c
```

Figure 9 : Exemple de code pour la figure 10

<sup>3</sup> Pour plus d'information consulter "Compilers: Principles and Practices" (Dave & Dave, 2012) <sup>[3]</sup>

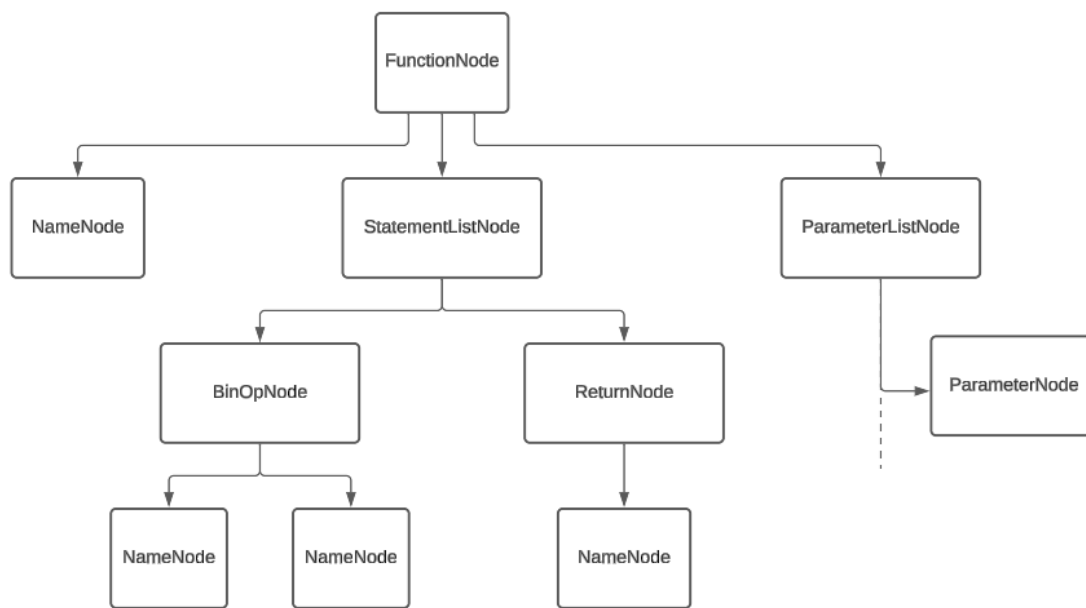


Figure 10 : Exemple d'AST de Pyright

Les fonctions définies pour visiter chaque nœud sont parfois très longues. Par exemple, la fonction permettant de visiter un `FunctionNode` fait environ 300 lignes et traite beaucoup de cas différents. En effet, beaucoup de notations sont facultatives lorsque l'on écrit une fonction, et prendre tous ces cas en compte implique d'augmenter la taille de la fonction. Par exemple, on peut ajouter des décorateurs, des annotations pour les types des paramètres, des paramètres par défaut, etc. Lors du parcours de ces nœuds, les différents types impliqués sont calculés et si des erreurs sont détectées, des diagnostics sont créés pour expliquer les détails. Cela comprend l'endroit de l'erreur, le type attendu, le type donné, et d'autres informations moins importantes pour l'objectif que l'on considère.

## 2.2.2 - Recherche d'améliorations

Pour pouvoir définir quels aspects du module on peut améliorer, on a expérimenté avec ce dernier. On a analysé différents cas de typage pour voir s'il restait des cas limite où le type n'était pas défini correctement. À force de recherches, un cas en particulier a fini par émerger : le typage des paramètres d'une fonction.

Pour pouvoir déterminer le type dans le corps d'une fonction, Pyright utilise les annotations de type mise à disposition par Python. Cela signifie que lorsque le développeur écrit une fonction, il renseigne les types attendus et ceux-ci sont utilisés dans le corps, comme montré dans la figure 11.

```
def foo(a:int):
    b = a + 1
    return b
```

Figure 11 : Exemple d'une annotation de type en Python

Le type inféré pour la variable `b` est bien `int`, comme le montre le résultat de Pyright dans la figure 12.

```
b = a + 1  
Full name: test_pyright.foo.b  
(variable) b: int
```

Figure 12 : Exemple d'inférence automatique de Pyright

En revanche, si le type n'est pas annoté, Pyright n'est pas en mesure de typer correctement les expressions, comme le montre les captures des figures 13 et 14.

```
def foo(a):  
    b = a + 1  
    return b
```

Figure 13 : Exemple de code non annoté en Python

```
b = a + 1  
Full name: test_pyright.foo.b  
(variable) b: Unknown
```

Figure 14 : Exemple d'échec de l'inférence automatique par Pyright

Par conséquent, on pourrait se retrouver dans des situations où la variable  $a$  est utilisée à plusieurs endroits où des types différents sont attendus. On se rend bien compte du problème dans l'exemple de la figure 15.

```
def useInt(a:int):  
    ...  
  
def useString(a:str):  
    ...  
  
def foo(a):  
    useInt(a)  
    useString(a)
```

Figure 15 : Exemple de code erroné non détecté par Pyright

Dans un cas, on s'attend à ce que la variable  $a$  soit de type *int* et dans l'autre, on s'attend à ce qu'elle soit de type *string*. Pour autant, Pyright ne relève aucune erreur. Il donne le type *Unknown* à la variable  $a$  et ne se pose pas plus de questions. On pourrait s'attendre à ce que Pyright infère un type à  $a$  de par son usage. Cela permettrait de typer le paramètre sans intervention du développeur et en même temps de détecter les erreurs de ce style. On a donc identifié une des limitations de Pyright. De plus, cela fait écho à l'utilisation de l'algorithme d'inférence d'Hindley-Milner dont nous avons parlé dans la première partie de ce rapport, faisant le lien et permettant d'appliquer les concepts liés au lambda-calcul.



## 2.3 - Typage des paramètre d'une fonction

L'idée de base pour typer un paramètre selon l'algorithme d'Hindley-Milner est simple. Nous avons un paramètre  $a$  dont nous ne connaissons pas le type, mais nous connaissons l'endroit où celui-ci est utilisé (callsite)<sup>4</sup>. Considérant que le callsite renvoie vers une fonction correctement typée, nous pouvons récupérer la signature, extraire le type attendu et l'inférer à notre paramètre. L'algorithme d'Hindley-Milner effectue cela en utilisant un système d'équations de type, toutefois, nous procéderons un peu différemment afin de mieux s'intégrer au système déjà en place. Pour mettre en place cette logique, la principale difficulté réside dans le fait de prendre en compte les différents cas de figure. Cela comprend le cas où les types sont compatibles, le cas où la signature du callsite est incomplète et les cas où les types sont incompatibles. Nous allons détailler ces différents cas dans les parties suivantes.

### 2.3.1 - 1er cas : Les types sont compatibles

Il existe plusieurs manières de définir si des types sont compatibles. Tout d'abord, nous allons récupérer tous les callsites où le paramètre concerné apparaît pour obtenir les types attendus. Ces types sont donc des candidats pour l'inférence du paramètre et nous pouvons les étudier. Nous avons isolé 3 cas que voici :

- Un lien d'héritage est présent entre les candidats. Dans ce cas, nous pouvons inférer le type le plus spécialisé. Exemple :

On a deux callsites qui nous donnent deux candidats A et B tels que B hérite de A. Comme nous pouvons utiliser des variables de type B dans les deux cas, nous pouvons inférer le type B.

- Une intersection impliquant un type union est non vide. Dans ce cas, nous pouvons inférer le type de l'intersection. Exemple :

On a deux callsites qui nous donnent deux candidats  $A|B$  et  $B|C$ . On remarque que les deux callsites acceptent des variables de type B, donc nous pouvons inférer le type B.

- Un cast sans perte d'information est possible entre les candidats. Si nous pouvons convertir un type en un autre sans perdre d'information, alors nous pouvons l'inférer. Exemple :

On a deux callsites qui nous donnent deux candidats *Int* et *Float*. On remarque que l'on peut caster un *Int* en *Float* sans perdre en précision. Dans ce cas, nous pouvons utiliser un *Int* dans les deux callsites sans que cela pose de problème. La conversion sera faite en cas de besoin. Toutefois, l'inverse n'est pas possible, car convertir un *Float* en *Int* fera perdre l'information sur la partie décimale.

Il faut aussi considérer que nous pouvons avoir une combinaison de tous ces cas. Il est tout à fait possible d'avoir à trouver un type qui correspond à la fois à *Float* et  $Int | String$ . Dans cet

---

<sup>4</sup> Nous nous concentrerons uniquement sur les callsites pour des raisons temporelles. Toutefois ce n'est pas le seul élément exploitable pour inférer le type.

exemple, le type *Int* convient. Il peut être utilisé dans les deux cas. Une fois par casting et une fois par spécialisation du type Union. La figure 16 montre que le type *int* convient dans les deux appels de fonction (il n'y a pas d'erreur).

```
def handleFloat(param: float):  
    ...  
  
def handleIntOrStr(param: Union[int, str]):  
    ...  
  
def process(param: int):  
    handleIntOrStr(param)  
    handleFloat(param)
```

Figure 16 : Exemple de code Python montrant un type satisfaisant plusieurs contraintes

### 2.3.2 - 2e cas: La signature du callsite est incomplète

Si la signature est incomplète, alors il est impossible d'inférer le type. Nous ne pouvons pas en créer un de toutes pièces. Dans ce cas, la solution est de laisser le paramètre avec un type Unknown comme c'est le cas dans la version actuelle de Pyright. Il faut quand même noter que cela ne s'applique que si tous les callsites ont des signatures incomplètes. En effet, si un callsite ne permet pas d'inférer le type, mais qu'il y en a d'autres pouvant amener à une inférence, on se retrouve dans le premier cas. On doit alors déterminer si les autres candidats sont compatibles. L'exemple de la figure 17 montre un cas où il n'est pas possible d'inférer un type.

```
def functionWithNoSignature(param):  
    ...  
  
def process(param):  
    functionWithNoSignature(param)
```

Figure 17 : Exemple de code Python où il est impossible d'inférer le type

### 2.3.3 - 3e cas: Les types sont incompatibles

Si les types sont incompatibles, cela signifie que le programme est erroné. Il faut garder à l'esprit que l'objectif initial est de prévenir le développeur lorsqu'une erreur de type est détectée. Dans ce cas, il faut utiliser les outils mis à disposition dans le code de Pyright afin de créer un diagnostic rendant compte des détails de l'erreur. L'exemple de la figure 18 montre un cas où le type ne peut pas être inféré à cause d'une incompatibilité entre les types.

```
def handleStr(param: str):  
    ...  
  
def handleInt(param: int):  
    ...  
  
def process(param):  
    handleStr(param)  
    handleInt(param)
```

Figure 18 : Exemple de code Python où les types sont incompatibles

### 2.3.4 - Implémentation

Pour implémenter cette logique dans le code, nous devons tout d'abord réfléchir à un algorithme qui pourrait nous permettre de détecter et de résoudre les différents cas que l'on considère. Une première étape serait de définir quelles sont les informations dont nous avons besoin en entrée et quelles sont celles que l'on attend en sortie :

- En entrée, nous souhaitons avoir une liste de candidats. Cela constitue notre base pour pouvoir déterminer s'il est possible de déduire un type qui correspond à tous, et si oui, lequel.
- En sortie, nous souhaitons avoir un type, le plus général possible, qui soit compatible avec tous les callsites.

Pour la suite, nous allons considérer une fonction en TypeScript *findCommonType()* qui prend deux types en entrée et renvoie le type le plus général pouvant être utilisé à leur place (nous verrons les détails de son implémentation plus bas). Si aucun type ne correspond alors la fonction renverra *undefined*. À partir de cela, nous pouvons donc définir une fonction et l'algorithme mentionné plus haut comme le montre la figure 19:

```
inferTypeFromCandidates(types: Type[]): Type | undefined {  
    if (types.length == 0) return undefined;  
    let typeToInfer: Type | undefined = types[0]  
    types.forEach((t) => {  
        if (typeToInfer === undefined) return;  
        typeToInfer = findCommonType(typeToInfer, t)  
    })  
    return typeToInfer;  
}
```

Figure 19 : Fonction calculant le type à inférer à partir des candidats

Ici, la logique est de maintenir une variable *typeToInfer* contenant le type le plus général possible et étant compatible avec tous les autres. Pour cela, on initie *typeToInfer* avec le premier candidat puis on le met à jour avec le résultat de la fonction appliqué à lui-même et à chacun des autres candidats. De cette façon, on s'assure d'avoir un type compatible avec chacun des candidats.

On remarque que si deux types sont incompatibles, *typeToInfer* devient *undefined*, ce qui signifie que l'on a trouvé une erreur de typage. On pourra donc par la suite créer un diagnostic d'erreur.

La fonction *findCommonType()* a un seul objectif, trouver si un type commun aux deux types d'entrée. Dans l'idéal nous aurions souhaité prendre en compte tous les cas mentionnés dans la section 2.3.1, mais par manque de temps, seul le cas concernant les types unions a été traité. En figure 20, nous pouvons voir l'implémentation de la fonction *findCommonType()* incomplète. Toutefois nous pouvons imaginer que cette fonction puisse être améliorée pour les autres stratégies simplement en les essayant si les précédentes échouent, comme le montre la figure 21.

```
function findCommonType(type1: any, type2: any): Type | undefined {  
  return intersection(type1, type2)  
}
```

Figure 20 : Fonction calculant le type commun à deux types (incomplète)

```
function findCommonType(type1: any, type2: any): Type | undefined {  
  return intersection(type1, type2) || cast(type1, type2) || inheritance(type1, type2)  
}
```

Figure 21 : Fonction calculant le type commun à deux types (proposition de fonction complète)

Bien évidemment, si aucune stratégie ne fonctionne, alors la fonction renverra *undefined*. Cela permettra de déterminer que l'inférence a échoué et qu'un diagnostic doit être créé.

Pour terminer, la fonction *intersection()* a pour objectif de calculer l'intersection entre deux ensembles de types, représentés par des types unions. Afin de profiter de cette fonction nous représentons les types simples par un type union ne contenant qu'un seul type. Cela permet de traiter ces deux cas simultanément sans dupliquer de code. L'implémentation de cette fonction peut être trouvée en figure 22.

```
function intersection(type1: Type, type2: Type): Type | undefined {  
  let intersection: Type = [];  
  type2.forEach((t2) => {  
    if (type1.some((t1) => t1 === t2)) intersection.push(t2);  
  })  
  if (intersection.length === 0) return undefined  
  else return intersection  
}
```

Figure 22 : Fonction calculant l'intersection entre deux types

On peut voir que le fonctionnement se limite à chercher les types qui apparaissent à la fois dans *type1* et *type2*. S'il n'y en a pas, alors la fonction renvoie *undefined*.

De cette façon et en ajoutant les diagnostics lorsque les types sont incompatibles, on obtient les résultats escomptés comme le montre les figures 23 et 24. Dans la figure 23, on peut inférer le type *str* à la variable *a* sans problème, il n'y a donc pas de diagnostic créé. Toutefois, dans la figure 24, les types *int* et *str* sont incompatibles et un diagnostic est créé avec les détails de l'erreur.

```
def IntOrStr(n:"int|str"):  
    return n  
  
def Str(s:"str"):  
    return s  
  
def func(a):  
    IntOrStr(a)  
    Str(a)  
    return a
```

Figure 23 : Exemple de cas sans erreurs

```
1  def Int(n:"int"):  
2      return n  
3  
4  def Str(s:"str"):  
5      return s  
6  
7  def func(a):  
8      Int(a)  
9      Str(a)  
10     return a
```

test\_pyright.py 1 of 1 problem

Type incompatibility between int|str Pyright([reportGeneralTypeIssues](#))

Figure 24 : Exemple de cas avec erreurs (incompatibilité int/str)

## 2.4 - Résultats et Ouverture

Nous avons exploré plusieurs cas de typage pendant cette recherche. Bien que plusieurs améliorations soient encore possibles, nous avons pu contribuer à une solution déjà en place. Grâce à cette contribution, il est maintenant possible, dans certains cas, d'inférer le type aux paramètres des fonctions et d'en vérifier la cohérence conformément aux principes du typage graduel. Cela a pour effet de renforcer la robustesse des programmes en permettant de garantir la cohérence des types de manière plus précise et plus générale, en prenant en compte un plus grand nombre de cas de typages.

Dans cette recherche, nous nous sommes concentrés sur l'utilisation des appels de fonctions pour obtenir des candidats et inférer les types des paramètres. Toutefois, il est possible de récupérer des candidats dans d'autres types de déclaration. On pourrait par exemple supposer que le corps d'une fonction itérant sur le contenu d'un de ses paramètres permet de déduire que le paramètre doit avoir un type itérable. En analysant chacune des instructions possibles en Python, il serait possible de dresser une liste des structures permettant de multiplier les façons d'obtenir des candidats pour inférer le type des paramètres des fonctions. Cela pourrait ensuite être utilisé dans l'algorithme permettant de déduire un type à partir des différents candidats. Il est difficile d'estimer la quantité de travail que cela représente, mais cela laisse la porte ouverte à de futures évolutions.

## 3 - Conclusion

### 3.1 - Retour sur l'expérience

Découvrir le travail de recherche a été une expérience très enrichissante. J'ai découvert une nouvelle façon de travailler et de m'organiser. Comme je n'ai pas été confronté aux mêmes difficultés que celles que l'on retrouve en entreprise, j'ai dû m'adapter à ce nouveau contexte. J'ai eu beaucoup de doutes et de questions tout au long de ce stage. Au début, j'avais du mal à me projeter. Le sujet de recherche est vague et j'ai eu du mal à trouver comment me lancer. Je ne savais pas vers où aller et j'avais peur de ne pas aboutir à quelque chose d'intéressant. Toutefois, à force de me renseigner sur le sujet, je me suis posé des questions de plus en plus précises, ce qui m'a permis de trouver un angle de recherche intéressant.

Le cadre d'un stage en recherche diffère de celui d'un stage en entreprise de par l'autonomie et la nature du travail demandé. En effet, les objectifs ne sont pas clairement définis dès le début, mais se précisent au fur et à mesure de l'avancement. Cela commence par un travail de recherche bibliographique permettant de comprendre les enjeux et de se familiariser avec les notions importantes liées au sujet. Généralement, c'est lors de ce travail que les premières pistes et questions apparaissent. Une fois qu'une problématique est identifiée, on peut commencer à réfléchir aux différentes solutions permettant de la résoudre. Lors de cette phase de recherche, beaucoup de solutions peuvent être envisagées. Il faut alors trier et définir les critères qui amèneront à un choix plutôt qu'un autre. C'est ainsi que l'on pourra commencer à étudier une des solutions permettant de résoudre la problématique.

### 3.2 - Compétences acquises

La principale compétence que j'ai développée pendant ce stage a été de faire évoluer et adapter un système existant. En effet, Pyright est un programme plutôt grand (~124800 lignes), beaucoup de logiques différentes sont utilisées et certaines parties sont difficiles à lire. Parfois, certains fichiers manquent de commentaires et le code n'est pas très parlant. J'ai donc dû passer plusieurs jours, à plusieurs reprises, à parcourir le code et à le tester. Au-delà d'avoir dû mettre en place un environnement de débogage spécifique à ce module dans VsCode, j'ai dû fournir un gros travail de rétro-ingénierie avec lequel je n'étais pas familier. Pour pouvoir contribuer à ce programme, je devais être capable de comprendre le code en place et les outils mis à disposition afin d'éviter d'ajouter des bogues et de ne pas recréer des fonctions déjà existantes. Je pense que cela m'a permis d'aiguiser ma manière de parcourir un code inconnu. Le code étant très volumineux, j'ai dû réfléchir à comment isoler les parties qui m'intéressent, comprendre comment elles fonctionnent et la manière qu'elles ont d'interagir avec le reste du code. J'ai aussi dû prendre le temps d'analyser la façon dont j'allais ajouter mon code. Savoir dans quels fichiers et quels endroits il est plus intéressant de se placer afin de suivre l'organisation déjà en place. Cela évite de mélanger plusieurs organisations différentes et de justement désorganiser le code.

Une autre compétence que j'ai pu développer dans une moindre mesure est liée à la nature exploratoire de la recherche. Il peut être difficile de s'organiser dans un contexte où on ne sait pas quels seront nos objectifs d'une semaine à l'autre. J'ai eu beaucoup de moments de doutes par rapport à la direction que je devais prendre. Lorsque je cherchais un angle sous lequel aborder le typage graduel avec Python et que je me suis rendu compte que des programmes fonctionnant très

bien existaient déjà. J'étais peu confiant dans le fait de trouver quelque chose d'intéressant à apporter. Toutefois, j'ai appris à aborder les choses beaucoup plus méthodiquement afin d'isoler les points intéressants à creuser. C'est comme cela que j'ai fini par trouver les limites de Pyright concernant le typage des paramètres des fonctions.

# Références

- [1]Castagna, G., Lanvin, V., & G. Siek, J. (2019). Gradual Typing: A New Perspective. *Proceedings of the ACM on Programming Languages*, 3(POPL), 1-32. <https://doi.org/10.1145/3290329>
- [2]C. Pierce, B. (2002). *Types and Programming Languages*. Mit Press.
- [3]Dave, P. H., & Dave, H. B. (2012). *Compilers: Principles and Practice*. Pearson Education India.
- [4]Hindley, R. (1969, Décembre). The Principal Type-Scheme of an Object in Combinatory Logic. *American Mathematical Society*, 146, 32. <https://doi.org/10.2307/1995158>  
Milner, R. (1978, Décembre). A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3), 348-375.  
[https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4)
- [5]PEP 729. (2023, Septembre 19). PEP 729 – Typing governance process.  
<https://peps.python.org/pep-0729/>
- [6]Pyright. (n.d.). Microsoft Open Source. Consulté le 22 Août 2024, à  
<https://microsoft.github.io/pyright/>
- [7]Python Type System Conformance Test Results. (2024, Août). Github.  
<https://htmlpreview.github.io/?https://github.com/python/typing/blob/main/conformance/results/results.html>
- [8]S. New, M., R. Licata, D., & Amhed, A. (2019). Gradual type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL), 1-31. <https://doi.org/10.1145/3290329>
- [9]Static Typing with Python. (n.d.). Static Typing with Python. Consulté le 23 Août 2024, 2024, à  
<https://typing.readthedocs.io/en/latest/#>
- [10]Typage dynamique — Wikipédia. (n.d.). Wikipédia. Consulté le 23 Août 2024, à  
[https://fr.wikipedia.org/wiki/Typage\\_dynamique](https://fr.wikipedia.org/wiki/Typage_dynamique)
- [11]Typage statique — Wikipédia. (n.d.). Wikipédia. Consulté le 23 Août 2024, à  
[https://fr.wikipedia.org/wiki/Typage\\_statique](https://fr.wikipedia.org/wiki/Typage_statique)
- [12]Type System Conformance. (n.d.). Github.  
<https://github.com/python/typing/tree/main/conformance>



## DOS DU RAPPORT

Etudiant (nom et prénom) : GRICOURT Killian

Année d'étude dans la spécialité : 5

Entreprise : Universidade Federal do ABC

Adresse complète : Av. dos Estados, 5001 - Bangú, Santo

André - SP, 09280-560, Brasil

Téléphone (standard) : (11) 3356-7000

Responsable administratif : MANDELLI Dalmo,  
représentante légale de l'UFABC

Courriel: ri@ufabc.edu.br

Tuteur de stage : Emilio de Camargo Francesquini

Courriel: e.francesquini@ufabc.edu.br

Enseignant-référent : Jean-François Méhaut

Courriel:jean-francois.mehaut@univ-grenoble-alpes.fr

Titre : Utilisation du typage graduel pour améliorer la sécurité des logiciels

Résumé : (minimum 15 lignes).

Français:

Il existe deux manières principales de typer un programme : statique ou dynamique. Le premier choix offre plus de résistance aux bogues lors de l'exécution en indiquant les erreurs dès la compilation. Cela évite d'avoir des bogues cachés qui peuvent parfois être longs à détecter. Toutefois, cela demande plus de travail et de réflexion au développeur, ce qui peut lui compliquer la tâche et allonger le temps de développement. Cela peut amener à des situations où le compilateur rejette des programmes qui pourraient fonctionner. Le deuxième choix donne plus de souplesse lors de l'écriture du code source en laissant la possibilité d'écrire des programmes qui seraient rejetés par un typage statique. La syntaxe est généralement plus permissive et permet des opérations de plus haut niveau. Toutefois, cela peut amener à des erreurs lors de l'exécution qui peuvent être très difficiles à identifier. Ainsi, il peut être difficile de choisir entre rapidité et simplicité ou stabilité d'un programme. C'est dans cet objectif qu'intervient le typage graduel. À mi-chemin entre les deux, il permet de s'assurer au maximum de la stabilité des types tout en gardant la flexibilité offerte par le typage dynamique. L'objectif de ce stage sera donc d'explorer les possibilités offertes par le typage graduel pour améliorer la qualité des logiciels. Nous nous focaliserons principalement sur l'ajout de typage graduel à Python en raison de sa popularité

English:

There are two main ways when it comes to type a program: statically or dynamically. The first option offers more resistance to runtime bugs by flagging errors during compilation. This helps prevent hidden bugs that can sometimes be difficult to detect. However, it requires more effort and thinking from the developer, which can complicate the task and increase development time. This can lead to situations where the compiler rejects programs that could work. The second option provides more flexibility when writing source code, allowing to write programs that would be rejected by static typing. The syntax is generally more permissive and allows for higher-level operations. However, this can lead to runtime errors that can be very difficult to identify. Thus, it can be challenging to choose between speed and simplicity or program stability. This is where gradual typing brings some solutions. Midway between static and dynamic typing, gradual typing ensures type stability while maintaining the flexibility offered by dynamic typing. The goal of this internship will be to explore the possibilities offered by gradual typing to improve software quality. We will mainly focus on adding gradual typing to Python as it is very popular nowadays.