UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES
Département d'informatique

Système de type

Pour les sciences des données

2024-09-06

Fabrice H.
Université de Genève
2024
Sciences informatiques

N.1. Abstract

Les sciences des données et les statistiques jouent un rôle de plus en plus important dans nos sociétés. Cependant, les langages de programmation actuels ne sont pas entièrement adaptés à ce nouveau paradigme de développement. L'objectif de cette recherche est de développer un système de type statique permettant de manipuler des tableaux multidimensionnels pour les sciences des données. La méthodologie consiste à créer un mini-langage capable de gérer les tableaux multidimensionnels, en utilisant des génériques et des types dépendants. Nous avons réussi à développer un modèle simple d'un langage capable de réaliser des opérations sur des scalaires, des matrices et des vecteurs, ainsi qu'un module pour les réseaux de neurones. Les résultats montrent qu'il est possible d'assurer un niveau satisfaisant de sécurité avec les types, bien que nous rencontrions des limitations en termes de représentation complète sans rendre l'algorithme de vérification de type indécidable à un certain degré.

Table des matières

I. Introduction

I.1. Context

Aujourd'hui les science des données jouent un rôle capital dans la dynamique de nos sociétés. Avec internet, les réseaux sociaux et même des domaines comme l'IoT, une quantité immense de données sont générées quotidiennement. C'est ce que l'on appelle le Big Data. De plus, la donnée est considérée comme le nouveau pétrole: Elle a beaucoup de valeur seulement dans le cas où on est capable de la raffiner. C'est pourquoi on a vu l'essor de plusieurs métiers basés sur la donnée. Nous avons notamment les métiers de data analyst, data engineer, ML engineer ou bien même data scientist.

Les sciences des données sont basées sur les mathématiques, l'informatique et l'expertise sur un domaine donné. Les mathématiques principalement utilisées en sciences des données sont l'algèbre, les probabilités, les statistiques ou bien même la théorie de l'information. Ces connaissances sont cruciales pour faire de l'analyse de données, du machine learning ou simplement du nettoyage de données. Il faut noter qu'une grande partie du travail fait sur des données reposent sur les tableaux multidimensionnels. Les tableaux multidimensionnels jouent un rôle crucial dans les sciences de données en raison de leur capacité à organiser et à manipuler efficacement des ensembles de données complexes. Contrairement aux tableaux unidimensionnels, qui ne permettent que de stocker des données de manière linéaire, les tableaux multidimensionnels permettent de représenter des structures de données plus riches, telles que des matrices, des tenseurs et des cubes de données. Cette capacité est essentielle pour traiter des jeux de données volumineux et hétérogènes provenant de diverses sources, telles que des expérimentations scientifiques, des bases de données transactionnelles et des données collectées en temps réel.

En sciences de données, les tableaux multidimensionnels facilitent la manipulation et l'analyse de données à haute dimensionnalité, souvent nécessaires pour modéliser des phénomènes complexes et interdépendants. Par exemple, dans l'apprentissage automatique, les images peuvent être représentées sous forme de tableaux multidimensionnels où chaque dimension correspond à une caractéristique spécifique de l'image (comme la hauteur, la largeur et les canaux de couleur). De même, dans l'analyse de séries temporelles ou de données spatiales, les tableaux multidimensionnels permettent de capturer des variations complexes dans le temps ou dans l'espace.

De plus, les opérations sur les tableaux multidimensionnels sont optimisées pour les calculs numériques et statistiques, offrant des performances améliorées par rapport aux structures de données plus simples. Les bibliothèques et les frameworks spécialisés, tels que NumPy en Python et les fonctions intégrées de manipulation de données en R, sont conçus pour exploiter efficacement les capacités des tableaux multidimensionnels, permettant ainsi aux scientifiques des données et aux analystes de réaliser des calculs sophistiqués avec une grande efficacité et précision.

Les experts du domaines utilisent le plus souvent des langages de programmation dynamiques tels que Python, R ou Julia. Ils ont l'avantage de permettre un prototypage rapide et un passage à un code de production avec peu de friction. Dans le domaine des sciences des

données, Python est le langage le plus populaire car il dispose de la syntaxe la plus facile ainsi qu'une quantité importante de modules qui vont bien au-delà des applications sur les données.

Malgré sa notoriété, Python a été aussi critiqué pour certaines de ses faiblesses. Il est considéré comme l'un des langages les plus lent, ce qui a poussé la communauté à développer des modules comme numpy ou panda ou l'essentiel des calculs sont fait dans des langage de plus bas niveau et plus rapides comme C/C++. De plus, l'absence de définition de type peut prendre du code compliqué à maintenir (pour la création de modèles par exemple). Python propose aussi un paradigme de programmation orienté objet mais qui peut être potentiellement mal adapté au simple traitement de données.

Python utilise des librairies comme Numpy, Tensorflow ou Pytorch pour créer des modèles pour les sciences des données. Toutes ces librairies ont en commun leur rapidité et la simplicité de leur API. Le problème rencontré vient surtout du système de type de python qui ne permet pas de saisir la forme des tableaux multidimensionnels (ou tenseurs) et de ce fait, ne permet aucune intelligence pour aider le scientifiques des données/développeur à construire une librairie solide. En effet, on se rend compte qu'un système de type efficace aide beaucoup lorsqu'on a des projets de plus grande envergure. C'est ce qu'on peut voir lorsqu'on jette un œil sur les modèles GPT qui présentent une vingtaine de couches qui peuvent donner assez rapidement du code spaghetti si on ne fait pas attention à la justesse de notre programme.

Comme l'essentiel des outils de sciences de données sont basés sur des langages de programmation dynamiques et souvent faiblement typés, nous rencontrons les problèmes qu'ils génèrent. Le premier problème vient de la maintenance. De par leur nature dynamique. Ces langages ont tendance à faire des « arrangements » en arrière plan et ne permettent pas d'assurer une lecture fidèle à l'exécution du code. Cela rend difficile la création de librairies, modules ou packages. C'est pourquoi cette tâche est laissée à des langages plus performants et mieux typés comme C++ pour créer des librairies. Le problème de ces derniers est qu'on ne trouve pas vraiment de système de type pour la gestion de tableaux multidimensionnels, ce qui est l'un des points les plus essentiels des sciences de données.

L'équipe de Andres Löh et al [1] ont pu developper une implémentation d'un langage combinés à des types dépendants (dans le cas du travail, c'était askell). Ceci a permis le typage et la vérification correcte de quelques concepts du domaines des tableaux multidimensionnels et de leur lien avec l'algèbre linéaire. Le soucis de ces langages vient du fait qu'ils sont trop « abstraits » et difficilement abordables pour l'ingénieur moyen.

Il n'y a donc pas de moyen raisonnablement accessible qui améliore la productivité lors de la construction de librairies pour les sciences de données. C'est là le but de ce projet.

I.2. Solutions

Nous abordons ici plusieurs solutions possibles pour pallier au problème de python concernant la gestion des tableaux multidimensionnels.

Python a développé le concept de « type hint » permettant d'ajouter graduellement des types à notre code et assurer la sécurité dans des zones critiques de notre code. Il y a aussi Cpython qui permet de créer du code à la frontière entre c et python et permet de produire du code plus efficace. Il y a cependant certaines plaintes qui présentent le système de type de

python comme non suffisant pour interagir avec des librairy. L'équipe de [2] ont développé un système de type qui permettrait d'étendre des modules qui manipulent les tableaux multidimensionnels qui sont beaucoup utilisés dans les data sciences. D'autres scientifiques des données ont aussi opté pour d'autres langages comme Julia qui présente les avantages d'un langage construit pour la science des données et qui propose un paradigme proche de l'orienté objet qui a ses avantages. De plus Julia est compilable et utilise par défaut la compilation JIT (Just in time compilation).

Mojo est un langage qui est arrivé récemment et se prononce comme le remplaçant de python. Comme le langage Typescript est un super set du langage Javascript, Mojo est un super set de python apportant un système de type plus robuste, une réelle programmation parallèle et la création de binaires. Mojo a été spécifiquement créé pour le développement d'intelligences artificielles. Ce langage a été fait pour faciliter la transition depuis le python. Malheureusement, Mojo n'a pas de système de type capable de traiter avec les tableaux multidimensionnels. Puisqu'il prend python pour base, il a quelques limitations sur sa syntaxe de base.

R est un langage conçu par les statisticiens, pour les statistiques premièrement et à des applications intéressantes pour les sciences des données et l'intelligence artificielle. Comme les langages abordés précédemment, R est un langage dynamique et faiblement typé, rendant l'aisance d'écriture simple et le prototypage accessible et l'interaction agréable. La particularité de R vient de ses structures de données qui sont basées sur des vecteurs, rendant les calculs basés sur l'algèbre linéaire et le traitement de collection de données faciles. Dans la philosophie de R, tout est vecteur, même la définition de valeur génère automatiquement des vecteurs. Nous traiterons du système de type de R plus en détail tout un peu plus loin.

Alexi Turcotte et al [3] ont élaboré un système de type pour le langage R et ont défini les principales caractéristiques qui le rendrait utilisable. L'équipe a fait le constat que R peut admettre difficilement un système de type au vu de sa nature dynamique. De plus, il faudrait définir un système de type qui apporterait un changement qui pousserait son adoption par la communauté de R. C'est pourquoi ils ont opté pour un système de type plus simple axé sur les signatures de fonctions (Typetracer) et un outils d'évaluation du typage du code en cours d'exécution (ContractR). Leur résultat ont été concluant, le système de type est facile à utiliser et montre un taux d'erreur d'inférence de type inférieur à 2%. Leur système de type est un bon fondement pour l'élaboration d'un éventuel système de type pour R.

I.3. Ma solution

J'ai pris la décision de créer un langage de programmation qui inclura un système de type efficace pour la manipulation de tableaux multidimensionnels ainsi que la création de modules pour les sciences de données. Il serait intéressant dans un future de créer ce langage sur le modèle du langage R comme cela était initié à la base. Pour l'instant, nous nous concentrons sur un langage noyau qui contiendra tous les éléments nécessaires à la manipulation de tableaux multidimensionnels.

Nous verrons en détail notre solution, mais nous pouvons déjà décrire les caractéristiques de notre solution. Celle-ci inclura bien évidemment des notions comme les tableaux, les génériques, les types dépendants et tout autres fonctionnalités qui rendrait le langage plus puis-

sant dans son expression. Cependant nous prendrons aussi en compte le besoin pratique de notre recherche. Il faut que la solution puisse aussi être flexible et raisonnable en termes de courbe d'apprentissage pour éviter de créer un modèle théorique qui ne marchera jamais pour la communauté des scientifiques de données.

II. Pourquoi les types ?

Les systèmes de types détectent les erreurs dans les langages de programmation en analysant les types de données avant l'exécution. Ils imposent des règles strictes pour garantir des opérations cohérentes, comme empêcher l'addition d'un entier et d'une chaîne de caractères. En vérifiant les types lors de la compilation ou avant l'exécution, ils détectent des erreurs telles que les affectations incorrectes, les appels de fonction avec des types inadéquats, et l'accès à des propriétés inexistantes. Ces vérifications réduisent les erreurs d'exécution et facilitent la détection précoce des bogues, améliorant ainsi la fiabilité du code.

Un système de types bien conçu pour un langage utilisant des tableaux multidimensionnels présente plusieurs avantages significatifs dans le domaine de la programmation et des sciences de données. Tout d'abord, un tel système permet de spécifier et de vérifier de manière statique la structure et les dimensions des tableaux utilisés dans le code. Cela aide à prévenir les erreurs courantes telles que les accès hors limites ou les opérations incompatibles sur les tableaux. Par exemple, en définissant des types spécifiques pour les tableaux à deux dimensions (comme matrices) ou à trois dimensions (comme tenseurs), le système de types peut garantir que les opérations effectuées sur ces structures respectent leurs propriétés dimensionnelles attendues.

De plus, un système de types robuste pour les tableaux multidimensionnels facilite la maintenance du code en offrant une documentation intégrée sur la structure et l'utilisation des données. Cela rend le code plus lisible et compréhensible pour les développeurs travaillant sur des projets collaboratifs ou en phase de maintenance. En spécifiant clairement les types des tableaux, les développeurs peuvent également bénéficier de fonctionnalités telles que l'inférence de types et la détection automatique d'erreurs potentielles lors de la compilation ou de l'exécution du programme.

De plus, un système de types bien adapté aux tableaux multidimensionnels peut favoriser l'optimisation automatique des performances. Les compilateurs et les interprètes peuvent utiliser les informations sur la taille et la disposition des tableaux pour générer un code plus efficace, exploitant par exemple la localité spatiale et temporelle des données lors des accès mémoire et des calculs.

Enfin, pour les applications en science de données et en calcul scientifique, où la précision des calculs et la gestion efficace des données sont cruciales, un système de types pour les tableaux multidimensionnels contribue à assurer la cohérence des opérations et la validité des résultats. Cela permet aux chercheurs et aux analystes de se concentrer sur les aspects conceptuels et algorithmiques de leurs travaux sans être constamment préoccupés par les problèmes liés à la gestion des données.

La solution développée dans ce papier est indépendante du langage de programmation, mais pour développer une solution qui aurait le potentiel d'être utilisée dans le futur, il faut adopter la solution de prendre ce qui existe déjà et en faire une version améliorée. J'ai décidé de choisir le langage R pour plusieurs raisons. Premièrement, par rapport à ses alternatives (Python, Julia), R ne dispose pas d'un système de type explicite permettant d'établir la correction des opérations fait dans le cadre du langage. Deuxièmement, le langage R est la raison pour laquelle ce projet a débuté à l'origine, car j'avais le désir de mettre en avant ce langage pour proposer une alternative intéressante à Python et Julia dans les sciences des données. En effet,

ces deux langages sont principalement construits sur le paradigme orienté objet. Cependant, étant moi-même un partisan des langages de programmation fonctionnels, R était le meilleur candidat pour poser son pied dans le domaine. Troisièmement, ayant certains contacts avec la base d'utilisateurs de R, j'ai pu établir le vrai besoin d'un système de type surtout dans la construction de package efficace. L'idéal serait de construire des package qui puissent être automatiquement accepté par CRAN[1].

[1]CRAN, abréviation de « Comprehensive R Archive Network », est l'organisation qui gère et distribue les packages et les ressources pour le langage de programmation R. Fondée en 1997, CRAN constitue une ressource centrale essentielle pour la communauté R, permettant aux développeurs et aux utilisateurs d'accéder à des milliers de packages R, de documentation, de manuels, et de données associées.

II.1. Le système de type de R

Avant de nous lancer sur la construction de notre solution, il est important de voir le fonctionnement du langage R afin de voir les caractéristiques qui en font un bon candidat en tant que langage de référence.

En R, tout est représenté sous forme de vecteurs, de liste ou de fonction. Même les scalaires sont des vecteurs. Un scalaire est simplement un vecteur de longueur 1, ce qui signifie que toute valeur en R est intrinsèquement un vecteur. Cette uniformité simplifie la manipulation des données, car les mêmes opérations peuvent être appliquées à des valeurs simples ou à des ensembles de valeurs sans distinction.

Exemple 2.1 : Les vecteurs en R

```r
# créer un vecteur
v1 <- c(1, 2, 3, 4) # donne [1, 2, 3, 4]

# La fonction 'c()' concatènes des vecteurs
v2 <- c(c(1, 2), c(3, 4)) # donne [1, 2, 3, 4]

# Définir une valeur crée automatiquement un vecteur
v3 <- 2  # donne [2]
```

De plus, les structures de données plus complexes, comme les listes, sont également basées sur des vecteurs. Les listes en R peuvent contenir des vecteurs de différentes longueurs et de différents types, offrant une grande flexibilité dans la gestion et l'organisation des données. C'est pourquoi il existe de nombreuses fonctions par défaut en R qui sont conçues pour opérer directement sur des vecteurs ou des listes, rendant le langage particulièrement puissant et efficace pour les analyses statistiques et les opérations de données.

Exemple 2.2 : Les listes en R

```r
# Création d'une liste: utilise le mot-clé "list"
nombres <- list(1, 2, 3, 4, 5)
noms <- list("Alice", "Bob", "Charlie")

# Accès aux éléments d'une liste
premier_nombre <- nombres[[1]]  # Résultat : 1
deuxieme_nom <- noms[[2]]  # Résultat : "Bob"

# Ajouter des éléments à une liste
nombres <- append(nombres, 6)  # La liste devient : list(1, 2, 3, 4, 5, 6)
noms <- append(noms, "David")  # La liste devient : list("Alice", "Bob",
"Charlie", "David")

# Supprimer des éléments d'une liste
nombres <- nombres[-which(nombres == 3)]  # La liste devient : list(1, 2,
4, 5, 6)
dernier_nombre <- tail(nombres, n = 1)  # Dernier élément : 6
nombres <- nombres[-length(nombres)] # Retirer le dernier élément, la liste
devient : list(1, 2, 4, 5)
```

Dans le langage R, tout peut avoir des attributs. Les attributs sont des métadonnées associées à un objet R. Ils fournissent des informations supplémentaires sur l'objet, sa structure, ou son comportement. Les vecteurs et les listes peuvent ainsi contenir des informations supplémentaires qui peuvent aider à changer le comportement du code s'il a été prévu pour.

Exemple 2.3 : Les attributs en R

```r
# Créer un vecteur
vecteur <- c(1, 2, 3)

# Afficher les attributs du vecteur
attributes(vecteur)

# Accéder à l'attribut "noms" (qui n'existe pas par défaut)
attr(vecteur, "noms")

# Ajouter un attribut "noms" au vecteur
noms <- c("un", "deux", "trois")
attr(vecteur, "noms") <- noms

# Afficher l'attribut "noms" après l'avoir ajouté
attr(vecteur, "noms")
```

Ces attributs sont des métadonnées qui peuvent être changées durant le temps d'exécution. Elles peuvent aussi être appliquées aux fonctions. Les classes ne sont en fait que des listes

qui ont des attributs spécifiques. Ces attributs peuvent être utilisés lorsque les structures sont passées en tant que éléments de fonction, l'expédition de la bonne fonction à appeler se fait de façon dynamique (dynamic dispatch[2]). C'est ce qu'utilise actuellement le système S3 ou S4 de R.

Les arrays dans R sont la structure de données la plus proche de ce qu'est un tenseur. En effet, c'est une représentation de tenseur. En réalité, les arrays sont des vecteurs avec un attribut spécial décrivant la dimension du dit tenseur. Mais la structure sous-jacente reste le vecteurs.

Exemple 2.4 : Les tableaux en R

```r
# Création d'un array 2x3
array_2x3 <- array(1:6, dim = c(2, 3))
print(array_2x3)

# Création d'un array 3x3x2
array_3x3x2 <- array(1:18, dim = c(3, 3, 2))
print(array_3x3x2)
```

Le fer de lance de R se trouve dans les dataframes qui sont une implémentation presque entièrement intégré dans le langage. Les dataframes sont des structures de données fondamentales en R, utilisées pour organiser des données tabulaires sous forme de lignes et de colonnes. Chaque colonne représente une variable distincte et chaque ligne correspond à une observation ou un enregistrement unique. Les dataframes sont essentiels en R pour plusieurs raisons principales. Tout d'abord, ils permettent de manipuler et d'analyser facilement des ensembles de données complexes et hétérogènes, provenant de diverses sources telles que des fichiers CSV, des bases de données ou des résultats d'expérimentations scientifiques. De plus, R offre un large éventail de fonctions intégrées et de packages spécialisés dédiés à la manipulation, à la transformation et à l'analyse de dataframes, facilitant ainsi des tâches telles que le filtrage, le tri, le calcul de statistiques descriptives et la création de graphiques. Enfin, les dataframes jouent un rôle crucial dans les analyses statistiques et la modélisation de données, permettant aux chercheurs, aux statisticiens et aux scientifiques des données d'effectuer des manipulations complexes tout en maintenant une organisation structurée et facilement interprétable des données. En réalité, un dataframe est une liste de vecteurs par définition.

[2]Le dynamic dispatch, ou la répartition dynamique, est un concept clé en programmation orientée objet qui concerne la façon dont les méthodes sont sélectionnées et exécutées en fonction du type réel d'un objet lors de l'exécution du programme. En d'autres termes, cela permet à un langage de déterminer dynamiquement quelle méthode appeler en fonction du type de l'objet sur lequel la méthode est invoquée.

Exemple 2.5 : Les dataframes en R

```r
# Créer des vecteurs pour chaque variable
nom <- c("Alice", "Bob", "Charlie", "David")
age <- c(18, 21, 22, 19)
filiere <- c("Informatique", "Mathématiques", "Statistique",
"Informatique")
moyenne <- c(16.5, 17.2, 18.1, 15.8)

# Combiner les vecteurs dans un data frame
etudiants <- data.frame(nom, age, filiere, moyenne)

# Afficher le data frame
print(etudiants)
```

En R, les fonctions jouent un rôle central et peuvent être définies de manière anonyme, ce qui est un aspect clé de la programmation fonctionnelle. Une fonction anonyme est une fonction sans nom qui est définie à la volée, souvent utilisée comme argument à d'autres fonctions. En R toute fonction est une fonction anonyme.

Exemple 2.6 : Les fonctions anonymes

```r
# Création et utilisation de fonctions anonymes en R

#Définir une fonction anonyme
add_two <- function(x) { x + 2 }

# Utilisation de la fonction
result <- add_two(3)
print(result)  # Résultat : 5

#Utiliser des fonctions anonymes avec lapply
nombres <- list(1, 2, 3, 4, 5)
```

L'équivalent de la fonction map sont les fonctions apply et lapply. Apply s'applique à des array, mais lapply est plutôt utilisé pour travailler avec des listes.

Exemple 2.7 : Applications sur les listes

```r
# Utiliser une fonction anonyme pour ajouter 2 à chaque élément
result <- lapply(nombres, function(x) { x + 2 })
print(result)  # Résultat : list(3, 4, 5, 6, 7)

# Utiliser des fonctions anonymes avec apply
# Créer une matrice
matrice <- matrix(1:9, nrow = 3)
```

Nous avons aussi des fonctions comme reduce et filter comme pour un langage de programmation fonctionnel utilisant des fonctions d'ordre supérieur.

Exemple 2.8 : Fonctions reduces et filters

```r
# Utiliser des fonctions anonymes avec Reduce
# Utiliser une fonction anonyme pour calculer la somme cumulative des
éléments
result <- Reduce(function(x, y) { x + y }, nombres)
print(result)  # Résultat : 15

# Utiliser des fonctions anonymes avec Filter et Find
# Utiliser une fonction anonyme pour filtrer les éléments pairs
result <- Filter(function(x) { x %% 2 == 0 }, nombres)
print(result)  # Résultat : list(2, 4)
```

Pour le reste les fonctions peuvent directement marcher sur des vecteur à cause de la fonctionnalité de « vectorization » du langage.

Floréal Morandat et al (source: Évaluating the design of the R language) ont donné une évaluation plus formelle du langage R et de sa sémantique.

II.2. Sémantique des langages

Établir une sémantique claire et précise pour un langage de programmation prototype revêt une importance capitale lorsqu'il s'agit de tester des hypothèses fondamentales sur sa conception. La sémantique d'un langage définit le sens exact et le comportement des constructions syntaxiques utilisées, ce qui est essentiel pour assurer la prédictibilité et la fiabilité du langage. En définissant rigoureusement la sémantique, les concepteurs peuvent garantir que chaque expression et chaque instruction est interprétée de manière cohérente par le système, minimisant ainsi les ambiguïtés et facilitant la compréhension par les programmeurs.

Une sémantique bien établie facilite également la vérification formelle du langage, permettant de valider sa correction avant même sa mise en œuvre complète. Cela inclut la capacité à utiliser des techniques telles que la vérification de type, la preuve de programme et les tests automatisés pour identifier et corriger les erreurs potentielles dans la conception du langage. De plus, une sémantique claire aide les développeurs à expérimenter rapidement avec de nou-

velles idées et variations de conception. Ils peuvent ainsi évaluer différentes approches pour la syntaxe, les règles de portée, la gestion de la mémoire et d'autres aspects essentiels du langage, tout en évaluant leur impact sur la facilité d'utilisation et les performances du langage.

Le lambda calcul

Le lambda calcul est un formalisme mathématique développé par Alonzo Church dans les années 1930 pour étudier les fonctions, les variables, et les applications de fonctions de manière abstraite. Il est souvent considéré comme l'un des fondements théoriques de l'informatique et des langages de programmation.

Le lambda calcul est particulièrement important car il fournit un modèle minimaliste et élégant pour la computation, où toutes les opérations peuvent être réduites à l'application de fonctions à des arguments. Il se compose de trois éléments de base : les variables, les abstractions (qui définissent des fonctions), et les applications (qui appliquent des fonctions à des arguments). Malgré sa simplicité, le lambda calcul est Turing-complet, ce qui signifie qu'il peut exprimer toute computation réalisable par une machine de Turing.

Definition 2.1: Synaxe du lambda calcul

$$
\begin{array}{rrll}
\text{Expr} & e & ::= & x \quad \text{variable} \\
 & & | & \lambda x.e \quad \text{abstraction} \\
 & & | & e\ e \quad \text{application} \\
\text{values} & v & ::= & \lambda x.e \quad \text{abstraction value}
\end{array}
$$

,

Definition 2.2: Évaluation du lambda calcul

$$\frac{t_1 \longrightarrow t_1 p}{t_1 t_2 \longrightarrow t_1 p t_2}\ \text{E-APP1}$$

$$\frac{t_2 \longrightarrow t_2 p}{v_1 t_2 \longrightarrow v_1 t_2 p}\ \text{E-APP2}$$

$$\frac{}{(\lambda x.t_{12}) v_2 \longrightarrow [x/v_2]t_{12}}\ \text{E-APPABS}$$

Le lambda calcul est un modèle de computation universel. Le lambda calcul fournit un cadre théorique qui peut simuler n'importe quel autre modèle de computation. Cela en fait un outil puissant pour comprendre les propriétés fondamentales des langages de programmation et pour prouver des théorèmes sur la computation.

Le lambda calcul est aussi une base pour les Langages Fonctionnels. C'est un avantage comme R, notre langage cible, est une langage de programmation orienté vers le fonctionnel. De nombreux langages de programmation modernes, tels que Haskell, Lisp, et même certaines parties de Python, sont fortement influencés par les concepts du lambda calcul. Il sert de base

à la programmation fonctionnelle, un paradigme qui traite les fonctions comme des citoyens de première classe et favorise des concepts comme l'immuabilité et les expressions pures.

Le lambda calcul servira de première pierre à notre langage prototype.

Le lambda calcul simplement typé

Malgré le fait qu'il soit Turing-complet. Le lambda calcul simple manque de pas mal de fonctionnalités qui vont nous aider à représenter les tableaux multidimensionnels. L'une d'entre elle et la notion de type. C'est pourquoi nous introduisons le lambda calcul simplement typé.

En effet, le lambda calcul simplement typé est le prochain pas vers la construction de notre premier langage. Il est une extension du lambda calcul non typé où chaque expression est associée à un type. Ce typage introduit une structure supplémentaire qui aide à prévenir certaines formes d'erreurs computationnelles et à garantir des propriétés de sûreté dans les programmes.

Definition 2.3 : Syntaxe du lambda calcul simplement typé

t	term	x	variable
	|	$\lambda x : T.t$	abstraction
	|	$t\ t$	application
v	value	$\lambda x : T.t$	abstraction value
T	types	$T \to T$	type of functions
Γ	context	$\emptyset$	empty context
	|	$\Gamma, \mathrm{x{:}T}$	term variable binding

Definition 2.4 : Évaluation du lambda calcul simplement typé

$$\frac{t_1 \longrightarrow t_1 p}{t_1 t_2 \longrightarrow t_1 p t_2}\ \text{E-APP1}$$

$$\frac{t_2 \longrightarrow t_2 p}{v_1 t_2 \longrightarrow v_1 t_2 p}\ \text{E-APP2}$$

$$\frac{}{(\lambda x.t_{12}) v_2 \longrightarrow [x/v_2]t_{12}}\ \text{E-APPABS}$$

Definition 2.5 : Typage du lambda calcul simplement typé

$$\frac{\Gamma(x) = T}{\Phi, \Gamma \vdash \mathrm{x} : \mathrm{T}}\ \text{T-VAR}$$

$$\frac{}{\Phi, \Gamma \vdash \lambda x : T_1.t_2 : T_1 \to T_2}\ \text{T-ABS}$$

$$\frac{\Gamma \vdash : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Phi, \Gamma \vdash t_1 t_2 : T_{12}}\ \text{T-APP}$$

Le lambda calculs simplement typé ajoute des types de base au choix et la définition de type de fonction. Le lambda calcul simplement typé est moins expressif que le lambda calcul classique mais offre plus de sécurité dans ses manipulations. On sera en mesure de faire des calculs plus sûrs dans notre langage, mais ce n'est pas encore suffisant pour assurer assez de sécurité pour la manipulation de tableaux multidimensionnels.

Le système F

Si le lambda calcul simplement typé apporte la notion de type au lambda calcul. Le système F apporte la notion de Générique sur ces types. Comme discuté dans le le lambda calcul simplement typé, la notion de type restreint fortement les opérations faisable sur les types. On le sait, on aura aussi parfois besoin d'avoir des fonctions plus générales pour la manipulation de type!

Le système F, également connu sous le nom de polymorphisme de deuxième ordre, est une extension du lambda-calcul typé qui permet l'utilisation de types génériques. Il introduit la quantification universelle sur les types, ce qui permet de définir des fonctions et des structures de données polymorphes. Par exemple, une fonction dans le système F peut être définie pour opérer sur des tableaux de n'importe quel type sans avoir à redéfinir la fonction pour chaque type de tableau.

Definition 2.6 : Syntaxe du système F

terms:	t	$::=$ x	variable
		$\mid \lambda x : T.t$	abstraction
		$\mid t\ t$	application
		$\mid \lambda X.t$	type abstraction
		$\mid t\ [T]$	application
values:	v	$::=$ $\lambda x : T.t$	abstraction value
		$\mid \lambda X.t$	type abstraction value
types:	T	$::=$ X	type variable
		$\mid T \to T$	type of functions
		$\mid \forall X.T$	universal type
context:	Γ	$::=$ X	empty context
		$\mid \Gamma, \mathrm{x{:}T}$	term variable binding
		$\mid \Gamma, X$	type variable binding

Definition 2.7 : Évaluation du système F

$$\frac{t_1 \longrightarrow t_1 p}{t_1 t_2 \longrightarrow t_1 p t_2}\ \text{E-APP1}$$

$$\frac{t_2 \longrightarrow t_2 p}{v_1 t_2 \longrightarrow v_1 t_2 p}\ \text{E-APP2}$$

$$\frac{}{(\lambda x.t_{12}) v_2 \longrightarrow [x/v_2]t_{12}}\ \text{E-APPABS}$$

$$\frac{t_1 \longrightarrow t_1 p}{t_1[T_2] \longrightarrow t_1 p[T_2]}\ \text{E-TAPP}$$

$$\frac{}{(\lambda X.t_{12})[T_2] \longrightarrow [X/T_2]t_{12}}\ \text{E-TAPPTABS}$$

Definition 2.8 : Typage du système F

$$\frac{\Gamma(x) = T}{\Phi, \Gamma \vdash \mathrm{x} : \mathrm{T}}\ \text{T-VAR}$$

$$\frac{}{\Phi, \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2}\ \text{T-ABS}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Phi, \Gamma \vdash t_1 t_2 : T_{12}}\ \text{T-APP}$$

$$\frac{\Gamma, X \vdash t_2 : T_2}{\Phi, \Gamma \vdash \lambda X . t_2 : \forall X . T_2}\ \text{T-TABS}$$

$$\frac{}{\Phi, \Gamma \vdash t_1 [T_2] : [X / T_2] T_{12}}\ \text{T-TAPP}$$

L'ajout de génériques dans le typage est un avantage considérable car il accroît la réutilisabilité et la flexibilité du code tout en maintenant une forte sécurité des types. Cela permet aux développeurs de créer des bibliothèques et des outils plus abstraits et polyvalents, réduisant ainsi le besoin de redondance et minimisant les erreurs. En conséquence, le système F et les types génériques favorisent une programmation plus expressive et plus sûre, où les invariants de type sont vérifiés à la compilation, garantissant une robustesse accrue des applications.

L'ajout de générique est aussi un élément crucial pour la définition de généricité pour des tableaux de taille différente. Nous verrons dans la suite comment ajouter cette notion.

III. Système C3PO

Jusqu'à présent nous avons parlé de modèles de calcul déjà existant et abordé leur particularités fondamentales et leur apport à notre langage prototype. Nous allons maintenant commencer à explorer des routes un peu plus aventureuses et rendre le design de ce prototype plus personnalisé pour notre problème. En partant des fondements du système F, nous pouvons maintenant entreprendre d'incorporer des extensions afin d'accroître l'expressivité du langage, dans le dessein de faciliter la manipulation de structures de données complexes, notamment des tableaux multidimensionnels.

Le prototype final s'appelle C3PO suite aux différentes transformations que nous avons apportées au système F.

Exemple 3.1 : Tableau récapitulatif de l'évolution du lambda calcul jusqu'au système C3PO

Langage	Fonctionnalité ajouté
lambda caclul	computation
lambda calcul simplement typé	typage
système F	génériques
système R	int, bool, opérateurs de base, if...then...else
système R2	context
système R2D	types dépendants sur les entiers positifs
système R2D2	fonctions générales
système C3PO	tableaux

Nous allons aborder les modifications présentées dans le tableau à partir du système R.

III.1. Système R

Le système R est juste une première tentative pour amener des éléments plus communs en programmation et avec lesquels nous allons développer notre prototype. Le but n'est pas d'imiter entièrement le langage de programmation R mais de prendre le minimum pour faire nos calculs. C'est pourquoi nous choisissons les deux types `int` et `bool` qui ont chacun leur rôle. Il est possible d'émuler ces valeurs à l'aide des éléments du système F mais c'est beaucoup plus pratique de travailler avec des valeurs plus communes.

Les nombre entiers positifs `int` vont nous permettre d'avoir une base minimum pour faire des calculs. C'est pourquoi nous ajoutons les opérateurs d'addition et de multiplication pour ne pas tomber sur des nombres à virgules ou des nombres négatifs.

Definition 3.1 : Le type de base « int »

$$\frac{}{\Delta \vdash n \longrightarrow n}\ \text{NUM}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} + \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} + \text{E2} \longrightarrow \text{E3}}\ \text{PLUS}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} * \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} * \text{E2} \longrightarrow \text{E3}}\ \text{TIME}$$

Les booléens se comportent aussi de la même façon que dans les langages de programmations classiques comme python. Ils s'appuient aussi sur l'arithmétique classique.

Definition 3.2 : Le type de base « bool »

$$\frac{}{\Delta \vdash \text{true} \longrightarrow \text{true}}\ \text{BOOL-T}$$

$$\frac{}{\Delta \vdash \text{false} \longrightarrow \text{false}}\ \text{BOOL-F}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1} \cap \text{E2} \longrightarrow \text{E3}}{\Delta \vdash \text{E1 and E2} \longrightarrow \text{E3}}\ \text{AND}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1} \cup \text{E2} \longrightarrow \text{E3}}{\Delta \vdash \text{E1 or E2} \longrightarrow \text{E3}}\ \text{OR}$$

Les booléens vont nous permettre d'ajouter de la logique à notre code et à simplifier le traitement conditionnel impliqué par le contrôle de flux `if...then..else` qui est un opérateur ternaire. Ce choix nous permet d'avoir une structure régulière capable d'émuler des `else if` par imbrication de contrôle de flux `ìf...then...else` .

Definition 3.3 : Les conditions

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{true}}{\Delta \vdash \text{if E1 then E2 else E3} \longrightarrow \text{E2}}\ \text{IF-T}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{false}}{\Delta \vdash \text{if E1 then E2 else E3} \longrightarrow \text{E3}}\ \text{IF-F}$$

Il nous faut aussi des opérateurs de base pour faire des testes logique. Ces opérateurs marcherons principalement pour les entiers car c'est le cas qui nous intresse le plus. L'opérateur d'égalité marchera pour tout les types (primitifs ou structure) du moment que les deux membre

de l'opération sont du même type. Il est à noté qu'on pourra utiliser les opérateurs « and » ou « or » pour créer des combinaisons de propriété plus complexes.

Definition 3.4 : Les opérateurs de bases pour les conditions

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} == \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} == \text{E2} \longrightarrow \text{E3}}\,\text{EQ}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} < \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} < \text{E2} \longrightarrow \text{E3}}\,\text{LOW}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} > \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} > \text{E2} \longrightarrow \text{E3}}\,\text{GRT}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} \leq \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} \leq \text{E2} \longrightarrow \text{E3}}\,\text{LOW-EQ}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} \geq \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} \geq \text{E2} \longrightarrow \text{E3}}\,\text{GRT-EQ}$$

On a maintenant un noyau qui nous donne la capacité de faire des opérations sur des ensembles de valeurs définis. Ce qui va nous permettre de traiter avec des opérations plus complexes.

III.2. Système R2

Bien que le système R nous donne plus de souplesse et de flexibilité dans nos calculs, nous pouvons toujours finir avec des représentations énormes car nous n'avons pas la possibilité de stocker temporairement des valeurs dans des variables. C'est pourquoi nous introduisons l'expression `let ... in ...` . Ce mécanisme va forcer le développement d'un contexte d'évaluation.

Il nous faut aussi une règle qui permet de vérifier que l'appel d'une variable est bel et bien valide, à savoir, que nous pouvons faire référence à une variable uniquement si elle a été préalablement définit à l'aide du mot clé « let ». Nous pouvont faire ceci en créant la règle « VAR ».

Definition 3.5 : Définition des variable et du contexte

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta, x = \text{E1p} \vdash \text{E2} \longrightarrow \text{E2p}}{\Delta \vdash \text{let } x : T = \text{E1 in E2} \longrightarrow \text{E2p}}\,\text{LET}$$

$$\frac{\Delta(x) = E}{\Delta \vdash x \longrightarrow E}\,\text{VAR}$$

III.3. Système R2D

Le système R2D va nous permettre d'introduire les types dépendants.

Les types dépendants sont des types qui dépendent de valeurs, ce qui permet d'exprimer des invariants et des contraintes plus précises directement dans le système de types. Par exemple, un vecteur de longueur n pourrait avoir un type qui dépend de n, garantissant que seules les opérations valides pour cette longueur sont permises. Cependant, les types dépendants peuvent involontairement devenir un obstacle pour les critères de notre langage. L'introduction de types dépendants rendent l'inférence de type indécidable. C'est pourquoi il est nécessaire de restreindre ses capacités. Ceci peut être fait avec l'arithmétique de Presburger.

L'arithmétique de Presburger est une théorie de l'arithmétique des entiers naturels avec l'addition, introduite par Mojżesz Presburger en 1929. Elle se distingue par sa décidabilité : il existe un algorithme qui peut déterminer si une proposition donnée dans cette théorie est vraie ou fausse. Cette propriété en fait un outil précieux en informatique, en particulier dans le domaine des types dépendants.

Grâce à l'arithmétique de Presburger, on peut formaliser et vérifier des propriétés comme la somme des longueurs de deux tableaux, ou la relation entre les indices dans une matrice, directement dans le système de types. Cela augmente la capacité du compilateur à détecter les erreurs à la compilation, avant même que le programme ne soit exécuté. En outre, cela aide à garantir la correction des programmes en prouvant mathématiquement des propriétés essentielles du code.

Afin de pouvoir accueillir des génériques ainsi que des types dépendants. Nous avons la nécessité de définir un système limité et simplifié.

Avec le contexte défini, nous avons la capacité de créer des variables adoptant un certain type. La règle **VAR** permet de vérifier l'assignation d'une variable si elle est présente dans le contexte. Le term **let** illustré par la règle **T-LET** permet la création desdites variables. Si les types de l'assignation correspondent, l'expression finale retourne un certain type.

Definition 3.6 : Context pour les types et les génériques

$$\frac{\Phi; \Gamma \vdash \mathrm{E1} : \mathrm{T1} \quad \Phi; \Gamma, x : \mathrm{T1} \vdash \mathrm{E2} : \mathrm{T2}}{\Phi; \Gamma \vdash \mathrm{let}\ x : \mathrm{T1} = \mathrm{E1}\ \mathrm{in}\ \mathrm{E2} : \mathrm{T2}}\ \text{T-LET}$$

$$\frac{\Gamma(x) = \sigma}{\Phi; \Gamma \vdash x : \sigma}\ \text{T-VAR}$$

III.4. Système R2D2

Jusqu'à présent les fonctions étaient représentées par des lambda et se décomposaient en deux types: Les abstractions lambda, et les abstractions de type. La plupart du temps, nous sommes obligés de faire la composition des deux pour créer des fonctions génériques. C'est pourquoi l'usage d'une représentation qui combine ces deux notions sera plus facile à traiter. Nous introduisons le concept de fonctions génériques. Cette fonction permet de combiner plusieurs

génériques et plusieurs paramètres en un coup et donc énormément simplifier la création de fonction en bloc.

Bien sûr, on peut toujours définir des fonctions sans générique, il suffit juste de laisser le champ des génériques vide. De même, si on ne veut pas de paramètre, on peut laisser le champ des paramètres vides. On peut créer des fonctions de cette forme:

Exemple 3.2 : Création d'une simple fonction

```
func<>(){7}
```

Comme mentionné précédemment, les fonctions sont l'un des outils les plus puissants et sophistiqués de ce langage, le but étant de pouvoir sécuriser les opérations sur des tableaux multidimensionnels. Ici, les fonctions peuvent admettre des génériques en plus des types sur chaque paramètre.

Definition 3.7 : Typage de fonction

$$\frac{\Phi, \overline{a : K}; \Gamma, \overline{x : T} \vdash E : T}{\Phi; \Gamma \vdash \text{func} < \overline{a : K} > \left(\overline{x : T}\right) \to T\{E\} :< \overline{K} > \left(\overline{T}\right) \to T} \text{T-FUNC}$$

Ces génériques peuvent être réservés pour contenir des types ou des valeurs et ainsi créer des fonctions spécifiques. On peut le voir quand à l'application de fonction.

Definition 3.8 : Typage d'une application de fonction

$$\frac{\begin{array}{c}\Phi; \Gamma \vdash \text{E1} :< \overline{\text{a:K}} > \left(\overline{T}\right) \to T \\ \Phi \text{ a:K}; \Gamma \vdash \forall t \in \overline{T}, e \in \overline{\text{E2}}, (\text{t:Tp} \wedge \text{e:Tp})\end{array}}{\Phi; \Gamma \vdash (\text{E1}) < \overline{\text{a:K}} > \left(\overline{\text{E2}}\right) : T} \text{T-FUNC-APP}$$

III.5. Système C3PO

Toutes ces fonctionnalités ont été introduites afin de favoriser la création et l'emploi de tableaux multidimensionnels. C'est pourquoi le dernier élément manquant n'est autre que la notion de tableau. Ici un tableau n'est autre qu'une liste d'éléments du même type avec une longueur déterminée.

Comme nous le verrons plus loin, l'élaboration de tableaux multidimensionnels se fera par la combinaison de plusieurs tableaux.

III.6. Syntax du langage

Après avoir défini ses différents composants, nous allons maintenant faire une présentation complète du langage prototype système C3PO.

Definition 3.9 : Syntax du langage Système C3PO

$\Phi : a \longrightarrow K$ **contexte de typage de variable de type**

$\Gamma : X \longrightarrow T$ **contexte de typage**

$\Delta : X \longrightarrow V$ **contexte d'évaluation**

Expression	E	::=	let x = E1 in E2	**let**
		|	func $< \overline{\text{a:K}} > (\overline{x : T}) \rightarrow T\{E\}$	**func**
		|	if E1 then E2 else E3	**if**
		|	op	**bop**
		|	(E1) $< \overline{K} > (\overline{E})$	**func_app**
		|	first(E)	**first_arr**
		|	rest(E)	**rest_arr**
		|	V	**V value**
Value	V	::=	$n \in \mathbb{N}$	**number**
		|	true	**true**
		|	false	**false**
		|	$[\overline{E}]$	**array**
Type	T	::=	$< \overline{K} > (\overline{T}) \rightarrow T$	**function_type**
		|	$[I;T]$	**array_type**
		|	int	**int**
		|	bool	**bool**
bop	`op`	::=	E1 and E2	**and**
		|	E1 or E2	**or**
		|	E1 + E2	**plus**
		|	E1 :: E2	**concat**
		|	E1 == E2	**equal**
		|	E1 < E2	**lower**
		|	E1 $\leq$ E2	**lower or equal**
		|	E1 > E2	**greater**
		|	E1 $\geq$ E2	**greater or equal**
Kind	K	::=	Type	**Type**
		|	Dim	**Dimension**

III.7. Sémantique d'Évaluation

Definition 3.10 : Rèlges d'évaluation part.1

$$\frac{}{\Delta \vdash n \longrightarrow n}\,\text{NUM}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} + \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} + \text{E2} \longrightarrow \text{E3}}\,\text{PLUS}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} * \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} * \text{E2} \longrightarrow \text{E3}}\,\text{TIME}$$

$$\frac{}{\Delta \vdash \text{true} \longrightarrow \text{true}}\,\text{BOOL-T}$$

$$\frac{}{\Delta \vdash \text{false} \longrightarrow \text{false}}\,\text{BOOL-F}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1} \cap \text{E2} \longrightarrow \text{E3}}{\Delta \vdash \text{E1 and E2} \longrightarrow \text{E3}}\,\text{AND}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1} \cup \text{E2} \longrightarrow \text{E3}}{\Delta \vdash \text{E1 or E2} \longrightarrow \text{E3}}\,\text{OR}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} == \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} == \text{E2} \longrightarrow \text{E3}}\,\text{EQ}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} < \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} < \text{E2} \longrightarrow \text{E3}}\,\text{LOW}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} > \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} > \text{E2} \longrightarrow \text{E3}}\,\text{GRT}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} \leq \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} \leq \text{E2} \longrightarrow \text{E3}}\,\text{LOW-EQ}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \text{E2} \longrightarrow \text{E2p} \quad \Delta \vdash \text{E1p} \geq \text{E2p} \longrightarrow \text{E3}}{\Delta \vdash \text{E1} \geq \text{E2} \longrightarrow \text{E3}}\,\text{GRT-EQ}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{true}}{\Delta \vdash \text{if E1 then E2 else E3} \longrightarrow \text{E2}}\,\text{IF-T}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{false}}{\Delta \vdash \text{if E1 then E2 else E3} \longrightarrow \text{E3}}\,\text{IF-F}$$

Definition 3.11 : Règles d'évaluation part.2

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta, x = \text{E1p} \vdash \text{E2} \longrightarrow \text{E2p}}{\Delta \vdash \text{let } x : T = \text{E1 in E2} \longrightarrow \text{E2p}}\text{LET}$$

$$\frac{\Delta(x) = E}{\Delta \vdash x \longrightarrow E}\text{VAR}$$

$$\frac{E \longrightarrow \text{Ep}}{\Delta \vdash \text{func} < \overline{a} > \left(\overline{x : P}\right) \rightarrow T\{E\} \longrightarrow \text{func} < \overline{a} > \left(\overline{x : P}\right) \rightarrow T\{\text{Ep}\}}\text{FUNC}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{func} < \overline{a} > \left(\overline{x : P}\right) \rightarrow T\{E\} \quad \Delta, \overline{\text{x:E}} \vdash E \longrightarrow \text{Ep}}{\Delta \vdash (\text{E1}) < \overline{T} > \left(\overline{E}\right) \longrightarrow \text{Ep}}\text{FUNC-APP}$$

$$\frac{\Delta \vdash \overline{E} \longrightarrow \overline{\text{Ep}}}{\Delta \vdash \left[\overline{E}\right] \longrightarrow \left[\overline{\text{Ep}}\right]}\text{ARR}$$

$$\frac{\Delta \vdash \left[\overline{\text{E1}}\right] \longrightarrow \left[\overline{\text{E1p}}\right] \quad \Delta \vdash \left[\overline{\text{E2}}\right] \longrightarrow \left[\overline{\text{E2p}}\right]}{\Delta \vdash \left[\overline{\text{E1}}\right] :: \left[\overline{\text{E2}}\right] \longrightarrow \left[\overline{\text{E1p}}, \overline{\text{E2p}}\right]}\text{CONC}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \overline{\text{E2}} \longrightarrow \overline{\text{E2p}}}{\Delta \vdash \text{first}\left(\left[\text{E1}, \overline{\text{E2}}\right]\right) \longrightarrow \text{E1p}}\text{FIRST-ARR}$$

$$\frac{\Delta \vdash \text{E1} \longrightarrow \text{E1p} \quad \Delta \vdash \overline{\text{E2}} \longrightarrow \overline{\text{E2p}}}{\Delta \vdash \text{rest}\left(\left[\text{E1}, \overline{\text{E2}}\right]\right) \longrightarrow \overline{\text{E2p}}}\text{REST-ARR}$$

III.8. Sémantique de typage

Definition 3.12 : Règle de typage part.1

$$\frac{n \in \mathbb{N}}{n : \text{int}}\ \text{T-NUM}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{int} \quad \text{E2} : \text{int}}{\Phi; \Gamma \vdash \text{E1} + \text{E2} : \text{int}}\ \text{T-PLUS}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{int} \quad \Phi; \Gamma \vdash \text{E2} : \text{int}}{\Phi; \Gamma \vdash \text{E1} * \text{E2} : \text{int}}\ \text{T-TIME}$$

$$\frac{}{\Phi, \Gamma \vdash \text{true} : \text{bool}}\ \text{T-TRUE}$$

$$\frac{}{\Phi, \Gamma \vdash \text{false} : \text{bool}}\ \text{T-FALSE}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{bool} \quad \Phi; \Gamma \vdash \text{E2} : \text{bool}}{\Phi; \Gamma \vdash \text{E1 and E2} : \text{bool}}\ \text{T-AND}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{bool}; \text{E2} : \text{bool}}{\Phi; \Gamma \vdash \text{E1 or E2} : \text{bool}}\ \text{T-OR}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : T; \text{E2} : T}{\Phi; \Gamma \vdash \text{E1} == \text{E2} : \text{bool}}\ \text{T-EQ}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{int} \quad \Phi; \Gamma \vdash \text{E2} : \text{int}}{\Phi; \Gamma \vdash \text{E1} < \text{E2} : \text{bool}}\ \text{T-LOW}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{int}; \text{E2} : \text{int}}{\Phi; \Gamma \vdash \text{E1} > \text{E2} : \text{bool}}\ \text{T-GRT}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{int}; \text{E2} : \text{int}}{\Phi; \Gamma \vdash \text{E1} \leq \text{E2} : \text{bool}}\ \text{T-LOW-EQ}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{int}; \text{E2} : \text{int}}{\Phi; \Gamma \vdash \text{E1} \geq \text{E2} : \text{bool}}\ \text{T-GRT-EQ}$$

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{bool} \quad \Phi; \Gamma \vdash \text{E2} : T \quad \Phi; \Gamma \vdash \text{E3} : T}{\Phi; \Gamma \vdash \text{if E1 then E2 else E3} : T}\ \text{T-IF}$$

Definition 3.13 : Règle de typage part.2

$$\frac{}{\Phi;\Gamma \vdash \text{int} : \text{Type}}\,\text{INT}$$

$$\frac{}{\Phi;\Gamma \vdash \text{bool} : \text{Type}}\,\text{BOOL}$$

$$\frac{m \in \mathbb{N}}{\Phi;\Gamma \vdash m : \text{Dim}}\,\text{DIM2}$$

$$\frac{\Phi;\Gamma \vdash \text{E1} : \text{T1} \quad \Phi;\Gamma, x : \text{T1} \vdash \text{E2} : \text{T2}}{\Phi;\Gamma \vdash \text{let } x : \text{T1} = \text{E1 in E2} : \text{T2}}\,\text{T-LET}$$

$$\frac{\Gamma(x) = \sigma}{\Phi;\Gamma \vdash x : \sigma}\,\text{T-VAR}$$

$$\frac{\Phi, \overline{a : K}; \Gamma, \overline{x : T} \vdash E : T}{\Phi;\Gamma \vdash \text{func} < \overline{a : K} > \left(\overline{x : T}\right) \to T\{E\} :< \overline{K} > \left(\overline{T}\right) \to T}\,\text{T-FUNC}$$

$$\frac{\Phi;\Gamma \vdash \text{E1} :< \overline{\text{a:K}} > \left(\overline{T}\right) \to T \qquad \Phi\ \text{a:K};\Gamma \vdash \forall t \in \overline{T}, e \in \overline{\text{E2}}, (\text{t:Tp} \wedge \text{e:Tp})}{\Phi;\Gamma \vdash (\text{E1}) < \overline{\text{a:K}} > \left(\overline{\text{E2}}\right) : T}\,\text{T-FUNC-APP}$$

$$\frac{\text{len}(E) \Rightarrow n \quad n : \text{Dim} \quad \Phi;\Gamma \vdash \forall e \in E, e : T \quad T : \text{Type}}{\Phi;\Gamma \vdash E : [n, T]}\,\text{T-ARR}$$

$$\frac{\Phi;\Gamma \vdash \left[\overline{\text{E1}}\right] : [m, T] \quad \Phi;\Gamma \vdash \left[\overline{\text{E2}}\right] : [n, T]}{\left[\overline{\text{E1}}\right] :: \left[\overline{\text{E2}}\right] \Rightarrow [\text{m+n}, T]}\,\text{T-CONC}$$

$$\frac{\Phi;\Gamma \vdash E : [m, T]}{\Phi;\Gamma \vdash \text{first}(E) : T}\,\text{T-FIRST-ARR}$$

$$\frac{\Phi;\Gamma \vdash E : [m + 1, T]}{\Phi;\Gamma \vdash \text{rest}(E) : [m, T]}\,\text{T-REST-ARR}$$

Ce langage est très simple car il se limite au strict minimum pour le projet.

IV. Sûrtée du langage

La « soundness » (ou sûreté) d'un système de types d'un langage de programmation est une propriété fondamentale qui garantit que le typage statique est fiable. En d'autres termes, un programme typé de manière statique ne produira pas d'erreurs de type lors de son exécution. Cette propriété assure que les programmes qui passent la vérification de types (c'est-à-dire qui sont considérés comme bien typés par le compilateur) ne rencontreront pas d'erreurs de type au moment de l'exécution, évitant ainsi un certain nombre de bogues et de comportements imprévisibles.

Pour prouver la soundness d'un système de types, deux théorèmes principaux sont généralement démontrés :

En combinant ces deux théorèmes, on peut démontrer que les programmes bien typés dans le système de types donné ne produiront pas d'erreurs de typage au moment de l'exécution, assurant ainsi la soundness du système de types. La préservation garantit que les types sont maintenus tout au long de l'évaluation, tandis que la progression assure que l'évaluation des programmes bien typés avance correctement.

Le théorème de la progression stipule que si une expression est bien typée, alors cette expression est soit une valeur (c'est-à-dire qu'elle est entièrement évaluée et ne peut plus être réduite), soit il existe une autre expression à laquelle elle peut se réduire (c'est-à-dire qu'il existe une étape de réduction possible). Formellement, si un term t est bien typée (c'est-à-dire $t : T$), alors t est soit une valeur, soit il existe un term t' telle que $t \longrightarrow t'$. Ce théorème garantit qu'un programme bien typé ne se bloquera pas de manière inattendue (c'est-à-dire qu'il continuera de progresser jusqu'à être entièrement évalué).

Theorem 4.1: Théorème de la progression

Si $\Phi, \Gamma \vdash t : T$, **alors** $t \in \text{Valeurs}$ **ou** $\exists t', t \longrightarrow t'$

Le théorème de la préservation affirme que si un programme est bien typé et qu'une opération de réduction (ou d'évaluation) est appliquée à ce programme, le résultat de cette opération est également bien typé. Formellement, si une expression t a un type T (c'est-à-dire $t : T$) et que t se réduit à t (noté $t \longrightarrow t'$), alors t' doit aussi avoir le type T (c'est-à-dire $t' : T$). Ce théorème garantit que les opérations de réduction ne violent pas les contraintes de typage.

Nous allons commencer la progression pour les éléments les plus simples du langage. Pour les valeurs comme true, false, les nombres ou les tableaux, la preuve est triviale.

Nous allons regarder la preuve pour le terme « T-IF »:

$$\frac{\Phi; \Gamma \vdash \text{E1} : \text{bool} \quad \Phi; \Gamma \vdash \text{E2} : T \quad \Phi; \Gamma \vdash \text{E3} : T}{\Phi; \Gamma \vdash \text{if E1 then E2 else E3} : T}\ \text{T-IF}$$

Expression T-IF:

- t = if t_1 then t_2 else t_3

Contraintes T-IF:

- t_1 : bool
- t_2 : T
- t_3 : T

Les expression t_2 et t_3 sont supposé être du même type. Ce qui nous intéresse est le term t_1. Par induction, t_1 est soit une valeur ou il existe un terme t_1′ où t_1 -> t_1′:

Expression t_1:

- t_1 is true => t_2 : T
- t_1 is false => t_3 : T

- t_1 -> t_1′ => if t_1′ then t_2 else t_3 : T

On voit donc que l'évaluation finit sur une valeur ou peut toujours progresser si t_2 et t_3 sont des expressions valides. On va maintenant se concentrer sur les opérations classiques. On les regroupe par leur type pour faire les choses plus simplement. Donc l'opérateur « and » va être le représentant de « or » et l'opérateur « + » va être le représentant des opérations « < », « > », « <= », « >= ». Il ne reste plus que les opérateurs « == » et « :: ». Commençons par la règle de l'opérateur « and ».

$$\frac{\Phi;\Gamma \vdash \text{E1} : \text{bool} \quad \Phi;\Gamma \vdash \text{E2} : \text{bool}}{\Phi;\Gamma \vdash \text{E1 and E2} : \text{bool}}\,\text{T-AND}$$

Expression T-AND:
- t = t_1 and t_2

Contraintes:
- t_1 : bool
- t_2 : bool

Case t_1:
- t_1 is v ∈ [true; false] => v and t_2 : bool
- t_1 -> t_1′ => t_1′ and t_2 : bool

Case t_2:
- t_2 is v ∈ [true; false] => t_1 and v : bool
- t_2 -> t_2′ => t_1 and t_2′ : bool

On voit donc que t_1 et t_2 peuvent toujours être des valeurs ou progresser vers d'autres valeurs.

$$\frac{\Phi;\Gamma \vdash \text{E1} : \text{int} \quad \text{E2} : \text{int}}{\Phi;\Gamma \vdash \text{E1} + \text{E2} : \text{int}}\,\text{T-PLUS}$$

—

Theorem 4.2: Théorème de la préservation

Si $\Phi, \Gamma \vdash t : T$ **et** $t \longrightarrow t'$, **alors** $\Phi, \Gamma \vdash t' : T$

Par définition, les valeurs ne peuvent être dérivées vers des nouveaux termes. Donc leur type est en quelque sorte « conservé ».

Prenons le cas du term « T-IF »:

Case T-IF:
- t = if t_1 then t_2 else t_3
  - t_1: Bool
  - t_2: T
  - t_3: T

Cela dérive en trois autres règles:

Subcase E-IFTrue:

- case: t_1 = true
- t' = t_2

So

- t_2: T

Subcase E-IFFalse:

- case: t_1 = false
- t' = t_3

So

- t_3: T

Subcase E-IF:

- t1 –> t_1′
- t' = if t_1′ then t_2 else t_3

So

- t_1: bool -> t_1 : bool
- t_2 : T and t_3 : T -> t' : T

V. Théorie

Dans cette section je vais montrer l'émulation de certains concepts utilisés dans la programmation pour les sciences des données.

Le but est de voir comment ceux-ci peuvent être interprété dans un langage fonctionnel fortement typé. Les concepts de représentation de tenseurs (matrice, vecteur et même scalaire), de broadcasting, de type embedding et autres seront un sujet important à traiter pour l'élaboration d'une librairie pour les sciences des données.

V.1. Types de données basiques

Nous allons représenter les données qu'on voit habituellement en algèbre linéaire et voir comment ceux-ci s'articulent dans le concept de notre langage.

Aujourd'hui les réseaux de neurones sont l'outil le plus populaire utilisé jusqu'à présent dans les sciences de données ou le machine learning. J'ai trouvé intéressant de voir ce que notre nouveau système de type est capable de faire pour ce type de cas.

Pour la réalisation de notre projet, il nous faut d'abord définir les éléments nécessaires à l'établissement de ce module, à savoir, les matrices, les vecteurs et les scalaires.

Les matrices sont à la base de l'algèbre linéaire et sont grandement utilisées pour simuler des réseaux de neurones. Dans notre cas, une matrice peut simplement être représentée comme un vecteur de vecteurs.

En sciences des données un tenseur est une généralisation des vecteurs et matrices. Il permet d'avoir une représentation homogène des données et de simplifier certains calculs. En effet, une matrice est un tenseur de 2 dimensions, un vecteur est un tenseur de 1 dimension et un scalaire est un tenseur de dimension 0. De plus, un hypercube est en fait un tenseur de dimension 3. C'est une façon de représenter les données de façon efficace.

Bien qu'on puisse travailler avec des tenseurs de plusieurs dimensions, on se rend compte en réalité que les datascientistes travaillent le plus souvent avec des tenseurs allant jusqu'à la dimension 5 au maximum. En effet, prendre de plus grandes dimensions rend les données difficiles à interpréter. De plus, l'essentiel des opérations se réalise au niveau du calcul matriciel, le reste des dimensions servant principalement de listes pour la représentation des informations.

On peut partir de l'hypothèse qu'un simple tableau (array) est un vecteur. Ce qui va nous permettre de sauter directement aux matrices.

Exemple 5.1: exemplte de type Scalaire et tenseurs

```
int = Scalaire
[1, int] = Vecteur d'une ligne et d'une colonne
[2, [3, int]] = matrice de deux lignes et 3 colonnes
[3, [3, [3, bool]]] = tenseur de degré 3 cubique
```

L'un des éléments fondamentaux de l'algèbre linéaire sont les matrices. En effet, les concepts de statistiques, de probabilité et de mathématiques sont représentés à l'aide de calculs sur les matrices. Le vecteur de vecteurs est la façon la plus simple de représenter les matrices. En effet les matrices ont une forme rectangulaire au carré ce qui fait que les sous vecteurs sont tous de la même taille ce qui va faciliter la représentation par des types.

Une matrice:

Exemple 5.2 : Simple matrice

$$\begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix}$$

Peut-être représentée comme ceci:

Exemple 5.3 : Représentation d'une simple matrice dans le langage système C3PO

```
[[2, 2], [2, 2]]
```

Il n'y a pas d'évaluation intéressante pour une valeur. Le typage est simple:

Exemple 5.4 : Exemple de la sémantique de typage d'une matrice

$$\dfrac{\dfrac{\text{len}([[2,\,2],\,[2,\,2]]) \Rightarrow 2 \quad \vdash 2 \text{ index} \qquad \dfrac{\text{len}([2,\,2]) \Rightarrow 2 \quad \vdash 2 \text{ index} \quad \Gamma \vdash 2 : \text{int}}{\Gamma \vdash [2,\,2] : [2,\,\text{int}]}\,\text{T-ARR}}{\Gamma \vdash [[2,\,2],\,[2,\,2]] : [2,\,[2,\,\text{int}]]}\,\text{T-ARR}$$

Plus loin nous traitons la notion de broadcasting pas mal utilisé dans le domaine des sciences des données et vu comment celle-ci peut-être représentée avec notre système de type. Ici nous nous intéressons à des opérations simples et faciles à typer.

Par la représentation actuelle des matrices il est facile de représenter les opérations d'addition ou de multiplication. En effet il faut que les matrices aient la même forme pour fonctionner ce qui équivaut à voir deux matrices de même type.

Pour ce faire, nous devons définir la notion de mapping. Notre langage est tiré du lambda calculus et exploite donc les notions de programmation fonctionnelle. Nous n'utilisons pas de boucle mais nous avons la notion de récursivité. Imaginons que nous voulions incrémenter de 1 tout les éléments d'un tableau avec notre langage. Le système de type nous prévient de faire des opération erronées comme:

Exemple 5.5 : Opération non valide dans le système de type

$$\frac{[1, 2, 3, 4] : [4, \text{int}] \quad 1 : \text{int}}{[1, 2, 3, 4] + 1 : ?}\text{PLUS}$$

Car l'opération d'addition de mon langage ne peut seulement se faire qu'entre deux entiers. Pour être en mesure d'appliquer le « + 1 » pour chaque membre, il va falloir itérer dessus à l'aide de la récursivité.

Exemple 5.6 : Fonction définie par l'utilisateur

```
let tableau_plus_1: ([N, int]) -> [N, int] =
  func<N>(tableau: [N, int]){
      if tableau == [] then
        []
      else
        [first(tableau) + 1] :: plus_1(rest(tableau))
  } in ...
```

Notez qu'on utilise déjà les notions de généricité pour travailler avec des tableaux de taille différentes. On pourrait refaire la même chose avec pour l'opérateur multiplication pour les entiers et « and » et « or » pour les booléens, mais le plus facile serait de créer la fonction map qui va nous permettre de simplifier les futures opérations avec les vecteurs et les matrices. La fonction map pourrait être définie comme:

Exemple 5.7 : Création de la fonction map pour les tableaux

```
let map: ((T) -> U, [N, T]) -> [N, U] =
  func<T, U, N>(f: (T) -> U, tableau: [N, V]) -> [N, U] {
      if tableau == [] then
        []
      else
        [f(first(tableau))] :: (map(f, rest(tableau)))
  } in ...
```

Comme l'application de « + 1 » n'est pas une fonction, on peut en créer une spécialisée:

Exemple 5.8 : Fonction unaire utilisable avec la fonction map

```
let plus_1: (int) -> int =
  func<>(num: int) -> int { N + 1 }
```

On est donc en mesure d'appeler la fonction de mapping avec cette fonction:

Exemple 5.9: Évaluation de l'addition d'un tableau avec un scalaire à l'aide des fonctions map et plus_1

$$\frac{}{\text{map}(\text{plus\_1}, [1, 2, 3, 4]) => [2, 3, 4, 5]}$$

Exemple 5.10: Typage de l'addition d'un tableau avec un scalaire à l'aide des fonctions map et plus_1

$$\frac{}{\text{map}(\text{plus\_1}, [1, 2, 3, 4]) : [4, \text{int}]}$$

Cette expression est correctement typée et donne [2, 3, 4, 5].

Si nous voulons faire le même type d'opération sur les matrices (donc des tableaux de tableau), il va nous falloir définir une fonction de mappage d'un degré plus haut. En s'appuyant sur la définition de notre fonction map préalablement établie, on peut créer notre fonction map2 comme suite:

Exemple 5.11: Fonction map pour une matrice

```
let map2: ((T) -> U, [N, [M, V]]) -> [N, [M, V]] =
  func<>(f: (T) -> U, mat: [N, [M, V]]) -> [N, [M, V]]{
      if mat == [] then
        []
      else
        [map(f, first(mat))] :: (map2(f, rest(mat)))
  }
```

L'avantage de la définition choisie des tenseurs se présente dans la similarité entre les deux fonctions « map » et « map2 ». Faire la fonction map3 donnerait un résultat similaire, il suffirait juste de faire appel à « map3 » à la place de « map2 » et de « map2 » à la place de « map1 ». Malheureusement notre langage ne supporte pas assez de fonctionnalité pour faire un mapping généralisé pour tout les tenseurs.

Une autre chose intéressante serait de pouvoir appliquer des opérations entres différents tenseurs. En algèbre linéaire, il faut que les matrices aient la même forme. Notre système de type peut assurer ça. On verra un peu plus loin le cas particulier du broadcasting. On aimerait être en mesure d'additionner ou de multiplier des matrices de même forme. Comme vu tout à l'heure, faire des calculs en utilisant directement l'opérateur ne marche pas et est prévenu par notre système de type:

Exemple 5.12 : Comment faire une addition entre deux tableaux ?

$$\frac{}{[1, 2, 3, 4] + [4, 3, 2, 1] : ?}$$

Une solution serait d'appliquer la notion de mapping mais pour les fonctions binaires (à deux opérateurs). Il y aurait un moyen de réutiliser les fonctions map définies précédemment à l'aide de tuple, mais comme notre langage ne l'implémente pas, on se contentera de simplement créer des des fonctions binaires et une fonction « map_op ». La fonction map_op est assez simple à réaliser, il suffit d'augmenter ce qu'on a déjà:

Definition 5.1 : Fonction pour appliquer une opération sur deux tableaux

```
let map_op: ((T, T) -> U, [M, T], [M, T]) -> [M, U] =
  func<T, U, M>(
    f: (T, T) -> U,
    tableau1: [M, T],
    tableau2: [M, T]) -> [M, U] {
      if and(tableau1 == [], tableau2 == []) then
        []
      else
      (f(first(tableau1), first(tableau2))) :: (map_op(f, rest(tableau1),
rest(tableau2)))
  }
```

On part du principe que la fonction « f » prend deux éléments du même type « T », mais on pourrait la généraliser davantage. On peut alors définir des fonctions binaires à l'aide des opérateurs de base.

Exemple 5.13 : Fonction d'addition à partir de l'opérateur d'addition

```
let plus: (int, int) -> int =
  func<>(a: int, b: int) -> int {
    a + b
  }
```

Exemple 5.14 : Fonction de multiplication à partir de l'opérateur de multiplication

```
let mul: (int, int) -> int =
  func<>(a: int, b: int) -> int {
    a * b
  }
```

Exemple 5.15 : Fonction band à partir de l'opérateur and

```
let band: (bool, bool) -> bool =
  func<>(a: bool, b: bool) -> bool {
    a and b
  }
```

Exemple 5.16 : Fonction bor à partir de l'opérateur or

```
let bor: (bool, bool) -> bool =
  func<>(a: bool, b: bool) -> bool {
    a or b
  }
```

L'addition entre deux vecteurs donnerait:

Exemple 5.17 : Addition entre deux tableaux avec les fonctions map_op et plus
map_op(plus, [1, 2, 3, 4], [4, 3, 2, 1]) => [5, 5, 5, 5]

Pour faire la même chose avec les matrice, il suffirait de reproduire « map_op » en « map_op2 ».

Exemple 5.18 : Opération pour les matrices

```
let map_op2: ((T, T) -> U, [M, [N, T]], [M, [N, T]]) -> [M, [N, U]] =
  func<T, U, M, N>(f: (T, T) -> U, tableau1: [N, T], tableau2: [N, T]) ->
[N, U] {
          (map_op(first(tableau1),   first(tableau2)))   ::   (map_op2(f,
rest(tableau1), rest(tableau2)))
  }
```

Ici encore, il suffit juste de reprendre la fonction « map_op » et de remplacer toutes les instance de « map_op » en « map_op2 ». Nous pouvons maintenant utiliser les opérations de bases sur les matrices.

Exemple 5.19 : Évaluation

$$\Gamma \vdash \text{map\_op2<int, int, 2, 2>(minus, [[2, 2], [2, 2]], [[1, 1], [1, 1]]) : [[1, 1], [1, 1]]}$$

Nous avons été capable de représenter les opérateur de base (« + », « * », « and », « or ») entre un scalaire et un vecteur, un scalaire et une matrice ainsi qu'un vecteur avec un vecteur et une matrice avec une matrice. Les opérations entre matrices et vecteurs sont en général traitées par le broadcasting. Nous laissons ce sujet pour la suite.

Nous pouvons voir un exemple avec la librairie numpy de python:

Exemple 5.20 : Broadcasting avec numpy

```
import numpy as np

col = np.array([1, 2, 3, 4])
lin = np.array([[4, 3, 2, 1]])

res1 = np.dot(lin, col)  = 20
res1 = np.dot(col, lin)  = error

= ValueError: shapes (4,) and (1,4)
= not aligned: 4 (dim 0) != 1 (dim 0)
```

Un autre exemple avec la librairie de pytorch:

Exemple 5.21 : Broadcasting avec pytorch 1

```
import torch

col = torch.tensor([1, 2, 3, 4])
lin = torch.tensor([[4, 3, 2, 1]])

res1 = torch.dot(lin, col)  = error
res2 = torch.dot(col, lin)  = error

= RuntimeError: 1D tensors expected,
= but got 2D and 1D tensors
```

Exemple 5.22 : Broadcasting avec pytorch 2

```
import torch

col = torch.tensor([1, 2, 3, 4])
lin = torch.tensor([[4, 3, 2, 1]])

res1 = lin + col  = tensor([[5, 5, 5, 5]])
res2 = col + lin  = tensor([[5, 5, 5, 5]])

print("res1:", res1)
print("res2:", res2)
```

Ce qui va nous intéresser maintenant est l'utilisation du produit matriciel. C'est une propriété fondamentale de l'algèbre linéaire qui nous servira à l'établissement de la librairie de réseaux de neurones. Si nous avons deux matrice A_{MxP} et B_{PxN} le produit matriciel A * B donnera C_{MxN}. Par exemple:

Exemple 5.23 : Produit matriciel $A \cdot B$

$$\begin{pmatrix} 1 & 2 & 0 \\ 4 & 3 & -1 \end{pmatrix} \cdot \begin{pmatrix} 5 & 1 \\ 2 & 3 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 9 & 7 \\ 23 & 9 \end{pmatrix}$$

Exemple 5.24 : Produit matriciel $B \cdot A$

$$\begin{pmatrix} 5 & 1 \\ 2 & 3 \\ 3 & 4 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 0 \\ 4 & 3 & -1 \end{pmatrix} = \begin{pmatrix} 9 & 13 & -1 \\ 14 & 13 & -3 \\ 19 & 18 & -4 \end{pmatrix}$$

La signature de cette fonction est simple à représenter:

Exemple 5.25 : Signature générale du produit matriciel

```
func<M, P, N>(A: [M, [P, int]], B: [P, [N, int]]) -> [M, [N, int]]
```

Pour définir le corps de la fonctions, il nous faudrait developper d'autres fonctions comme la transposition (car le produit matriciel est un produit ligne colonne) et modifier map_op pour être en mesure d'avoir des opérateurs qui prennent des type différent (signature (T1, T2) -> T3). Par soucis de simplicité, on ne le définira pas ici.

Exemple 5.26 : Création du corps de la fonction dot

```
let reduce_op : <T, U, N>((T, T) -> U, [N, T], [N, T]) =
  func<T, N>(op: (T, T) -> U, a: [N, T], b: [N, T]) -> U {
     If and(a == [], b == []) then
        []
     else
        prepend(op(first(a), first(b)), reduce(op, rest(a), rest(b))
      end
}

let scalar_dot : <N, T>([N, T], [N, T]) -> T =
  func<N, T>(a: [N, T], b: [N, T]) -> T {
     reduce(plus, map_op(mul, a, b))
}

let right_dot : <M, N, T>(B: [M, [N, T]], a: [N, T]) =
  func<M, N, T>(B: [M, [N, T]], a: [N, T]) {
   map_op(scalar_dot, a, B)
}

let dot : <M, N, T>(A: [M, [N, T]], B: [M, [N, T]]) =
  func<M, N, T>(A: [M, [N, T]], B: [M, [N, T]]) {
    map_op(right_dot, transpose(B), A)
}
```

Pour faire le produit matriciel (ici appellé dot) on doit utiliser la fonction de transposition ainsi que les fonctions définis right_dot, scalar_dot et la méthode reduce qu'on défini pour ce cas d'usage. Ainsi on va faire une boucle entre chaque ligne de la matrice de gauche et pour chacune de ces ligne faire une boucle avec les colonnes de la matrice de droit en faisant simplement la some du produit, élément par élément des deux tableaux.

Avec ces restrictions, nous sommes en mesure de représenter le typage concrèt de cette fonctions.

Dans le domaine des sciences des données, nous avons aussi un ensemble de constructeures que nous pouvons utiliser. Notre langage ne nous permet pas la génération de nombre aléatoire donc on aura pas de générateur de matrice avec des éléments aléatoire bien que cela est assez pratique dans la création de réseaux de neurones.

Exemple 5.27 : Constructeurs de matrice

```
let vector: <T,M>(int, T) -> [M, T] =
  func <T, M>(len: int, value: T) -> [M, T] {
    if len == 0 then
      []
    else
      [value] :: vector(len-1, value)
  }

let matrix: <T, M, N>(int, int, T) -> [M, [N, T]] =
  func <int, M, N>(dim1: int, dim2: int, value: T) {
    if dim1 == 0 then
      []
    else
     vector(dim2, value) :: matrix(dim1-1, value)
  } in

let zeros: (int, int) -> [M, [N, int]] =
  func <>(dim1: int, dim2: int) -> int {
    matrix(N1, N2, 0)
  } in

let ones: (int, int) -> [M, [N, int]] =
  func <>(dim1: int, dim2: int) -> [M, [N, int]]{
    matrix(dim1, dim2, 1)
  } in

let trues: (int, int) -> [M, [N, bool]] =
  func <M, N>(dim1: int, dim2: int) -> [M, [N, bool]] {
    matrix(dim1, dim2, true)
  } in

let falses: (int, int) -> [M, [N, bool]] =
  func <M, N>(dim1: int, dim2: int) -> [M, [N, bool]] {
    matrix(dim1, dim2, false)
  }
```

Exemple 5.28 : Autres fonctions utilitaires pour les matrices

```
let length: <M, N, T>([M, [N, T]]) -> int =
  func <M, N, T>(m: [M, [N, T]]) -> int {
    M * N
} in

let fill: <M, N, T>([M, [N, T]], T) -> [M, [N, T]] =
  func <M, N, T>(m: [M, [N, T]], value: T) -> [N1, [N2, T]] {
    matrix(M, N, value)
} in

let reduce : <T, U, N>((T) -> U, [N, T]) =
  func<T, N>(op: (T, T) -> U, a: [N, T], b: [N, T]) -> U {
     If and(a == [], b == []) then
        []
     else
        prepend(op(first(a), first(b)), reduce(op, rest(a), rest(b))
      end
} in

let concat: <M, T, N>([M, T], [N, T]) -> [M+N, T] =
func <M, T, N>(m1: [M, T], m2: [N, T]) -> [M+N, T] {
  m1 :: m2
}

let linearize: <M, N, T, O>([M, [N, T]]) -> [O, T] =
  func <M, N, T, O>(m: [M, [N, T]]) -> [O, T] {
    reduce(concat, m)
  }
```

En algèbre linéaire on est capable de faire un produit matriciel entre des vecteurs et des matrices un produit matriciel entre un vecteur ligne à gauche une matrice à droite aussi donner un vecteur colonne le produit matriciel entre une matrice à gauche et un vecteur colonne à droite va donner un vecteur à ligne.

Le souci est que la définition actuelle du vecteur est un tableau d'une seule ligne mais la définition du produit matricielle demande l'implémentation de deux matrices.

On pourrait créer une fonction matricielle dédiée mais on peut faire quelque chose de plus malin. En effet on peut représenter les vecteurs lignes et colonne comme des cas particuliers de matrice. Un vecteur ligne serait matrice de une ligne et de N colonne. Avec cœur colonne serait une matrice de une colonne et de n lignes.

Avec cette représentation on est non seulement capable de faire une distinction entre les vecteurs lignes et les vecteurs colonnes mais on les rend aussi compatibles avec l'opération du produit matriciel.

Pour donner toute la vérité les vecteurs sont maintenant compatibles avec toutes les opérations qu'on a défini sur les matrices. On peut donc additionner ou multiplier les vecteurs avec les même fonctions.

C'est pourquoi nous ferons donc une distinction entre les tableaux et les vecteurs au vu de tout les avantages que cela nous apporte.

Pour aller plus loin on pourrait essayer de représenter les scalaires en tant que matrices. La représentation la plus évidente serait de prendre une matrice de une ligne et de une colonne. C'est ce qu'on va faire.

Encore une fois, si les scalaires sont un cas particulier de matrice. Ils bénéficient de toutes les fonctions qu'on a établies pour les matrices. Ce qui nous intéresse le plus est de voir leur comportement face au produit matriciel. Étant donné qu'un scalair est une matrice A_{1x1}, le produit scalaire demanderai que l'autre matrice soit de la forme 1xN si elle se trouve à droite et Mx1 si elle se trouve à gauche.

Cela signifie qu'un produit matriciel entre un scalaire et un vecteur peut être vu comme la multiplication classique entre un scalaire et un vecteur en algèbre linéaire classique. On pourrait tenter d'étendre cette fonctionnalité sur les matrices en général mais on se rend compte qu'il nous faudrait alors une matrice scalaire carré. Elle contiendra toujours la même valeur mais devra changer de forme.

Nous avons donc une représentation pour les scalaires des vecteurs et les matrices. Escalaires sont un cas particulier des matrices étant des matrices carrées. Les vecteurs dans votre carte particulier des matrices ayant l'une de leurs dimensions signalées à un. La natation générale de la matrice représente tout.

Nous nous rendons compte que dans les sciences des données nous utilisons aussi des danseurs qui vont au-delà de la dimension 2. Généralement c'est danseur en plus un rôle de représentation car les calculs qui se font se feront souvent au niveau matriciel.

Un tenseur de dimension 3 représente souvent une image nous avons la dimension des coordonnées X la dimension des coordonnées y ainsi que la dimension pour les couleurs RGB qui est généralement de longueur 3.

Un tenseur des dimensions 3 peut aussi représenter une liste de matrices. Cela peut aider pour définir une application de filtre sur une image comme par exemple. Cela peut aussi être une liste d'images en noir et blanc. Donc les représentations peuvent être vraiment multiples. parfois on peut attendre les dimensions 5 ou 6 parce que nous pouvons travailler avec des objets 3D en tant que liste avec aussi l'ajout d'un batch size bien même des vidéos, etc. Les calculs fait sur ces tenseurs prélève souvent d'une application matricielle combiné au broadcasting.

V.2. Broadcasting

Le broadcasting fait référence à la capacité d'une bibliothèque de manipuler des tableaux de différentes dimensions ensemble, en étendant implicitement la forme des tableaux plus petits pour qu'ils correspondent à la forme des tableaux plus grands. Cela permet de réaliser des opérations élémentaires sans explicitement dupliquer des données, ce qui est crucial pour l'efficacité computationnelle.

Le but est de voir comment le broadcasting peut-être implémenté dans notre système de type. L'idéal serait de pouvoir le faire pour les tenseurs avec les opération binaires (+, -, /, etc.). En général, l'extension se fait par la dernière dimension.

Cas simple

Dans cet exemple, fait avec python et la librairie numpy, l'opération marche sans problème. Nous avons deux tenseurs de même taille. Ce genre de cas est facile a implémenter et ne dépend pas du broadcasting. Cet exemple est juste pour montrer que c'est une propriété du broadcasting.

Exemple 5.29 : Broadcasting avec numpy vecteur x vecteur

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b = array([2.,  4.,  6.])
```

Propriété 1: extension axiale

Une des propriétés intéressantes du broad casting est l'extension axiale. On est capable de prendre deux tableaux de taille différente. Nous avons ici, deux tenseurs de dimension 1 mais de longueur différente. Le broadcasting permet d'étendre la longueur de l'axe le plus court.

Exemple 5.30 : Broadcasting avec numpy vecteur x scalaire

```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0])
# a * b = array([2.,  4.,  6.])
```

Si nous avons deux tenseurs de dimension N, T1: (n1, n2, ..., nN) et T2: (m1, m2, .., mN). Le broadcasting va nous permettre d'ajuster la taille de la dimension la plus petite à la dimension la plus grande. On aurait une signature du type:

op((n1, n2, ..., nN), (m1, m2, ..., mN)) -> (max(n1, m1), max(n2, m2), ..., max(nN, mN))

Dans le cas de l'exemple précédent, nous avons cette transformation: *((3), (1)) -> (3).

Propriété 2: extension dimensionelle (extension sur la gauche)

Une autre propriété du broadcasting est sa capacité à ajuster la forme d'un tenseur en ajustant le nombre de dimension.

Exemple 5.31 : Broadcasting avec numpy matrice x vecteur

```
a = np.array([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = np.array([2.,  4.,  6.])
# a * b =  np.array([[ 2.  8. 18.], [ 8. 20. 36.]])
```

Si nous avons deux tenseurs de dimensions différentes T1: (n1, ...), T2: (m1, ..) et dim(T1) = N, dim(T2) = M le broadcasting va nous permettre d'ajuster la dimension la plus petite en ajoutant des axes supplémentairs de longueur 1. Après quoi, on applique l'extension axial mensionné précédement. Il serait difficile de représenter une signature pour des tenseurs de dimensions différentes sans ajouter des notations bizarre.

Notre exemple de tout à l'heure nous donnerait: *((2, 3), (3)) -> (2, 3). Comme mentionné précédement, il n'y a pas de méthode possible avec notre système de type pour représenter ce genre d'opération. Pour accomplir cela, on pourait inclure la dimension dans la description de type comme dans le travail de Justin Slepak et al [4].

Autre propriété

Dans les langage dynamique comme python, le broadcasting est couplé à une transformation dynamique de type. Notament, on peut faire des calcules élémentaires entre des nombres et des matrices, ce qui est proche de la vision mathématique de l'algèbre linéaire.

Exemple 5.32 : Broadcasting avec numpy vecteur x scalaire

```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b = array([2.,  4.,  6.])
```

On se réalise que le broadcasting est la combinaison de trois applications: l'extension dimensionnelle, l'extension axiale et l'application de l'opération élément par élément. On peut donc séparer ça en trois fonctions différentes. Comme on l'a vu, les fonctions d'extension dimensionnelle et d'extension axiale.

L'équipe [2] qui voulait developper un système de type pour les ndarray de numpy a proposé une représentation du broadcasting intéressante mais qui n'offre pas la sécurité qui nous intéresse.

Exemple 5.33 : Exemple de typage pour le broadcasting

```
def broadcast(x: Ndarray[*A], Ndarray[Broadcast[A]])
-> Ndarray ?:
```

La représentation des tenseurs que nous avons adopté dans le chapitre précédent reste quand même très puissante et assez fidèle à la représentation mathématique des vecteurs et matrices

tout en gardant l'habilité de travailler avec des tenseurs de dimensions supérieur à 2. Malheureusement le broadcasting ne peut pas être fidèlement typé et sécurisé dans ce langage et sera donc mis de côté.

VI. Implémentation

Dans le cadre de ce projet j'ai construit le prototype d'un interpréteur en Prolog pour appliquer et vérifier la faisabilité du langage construit jusqu'à présent. Le premier but est de traduire la syntaxe de base de notre langage prototype en son équivalent prolog et implémenter les règles d'évaluation et de typage.

Voici la version « Prolog » du langage:

Definition 6.1: Traduction des expressions en syntaxe prolog

Expression	E	::=	$\text{let}(\text{var}(x), \text{E1}, \text{E2})$	**let**
		|	$\text{func}\left(\overline{\text{gen}(a)}, [x, T], T, E\right)$	**func**
		|	$\text{if}(\text{E1}, \text{E2}, \text{E3})$	**if**
		|	op	**bop**
		|	$\text{app}\left(\text{E1}, \overline{a}, \overline{E}\right)$	**func_app**
		|	$\text{first}(E)$	**first_arr**
		|	$\text{rest}(E)$	**rest_arr**
		|	V	**V value**
Value	V	::=	$n \in N$	**number**
		|	v_true	**true**
		|	v_false	**false**
		|	$\text{v\_array}\left(\overline{E}\right)$	**array**
Type	T	::=	$\text{t\_func}\left(\overline{T}, T\right)$	**function_type**
		|	$\text{t\_array}(n, T)$	**array_type**
		|	t_int	**int**
		|	t_bool	**bool**
bop	`op`	::=	$\text{and}(\text{E1}, \text{E2})$	**and**
		|	$\text{or}(\text{E1},\text{E2})$	**or**
		|	$\text{plus}(\text{E1},\text{E2})$	**plus**
		|	$\text{time}(\text{E1},\text{E2})$	**time**
		|	$\text{append}(\text{E1}, \text{E2})$	**concat**
		|	$\text{equal}(\text{E1},\text{E2})$	**equal**
		|	$\text{lower}(\text{E1},\text{E2})$	**lower**
		|	$\text{low\_eq}(\text{E1},\text{E2})$	**lower or equal**
		|	$\text{greater}(\text{E1},\text{E2})$	**greater**
		|	$\text{gre\_eq}(\text{E1}, \text{E2})$	**greater or equal**
context	ctx	::=	$\vdash \Gamma$ ctxt	**valid_context**
		|	$\text{is\_ctxt}(\text{context})$	**valid_context**
		|	$\text{is\_type}(\text{type})$	**type_in_context**
		|	$\text{add}(\text{context}, M : \sigma)$	**term_of_type_in_context**
		|	$\text{equal\_ctx}(\Gamma, \Delta)$	**equal_contexts**
		|	$\text{equal\_type\_in\_context}(\sigma, \tau)$	**equal_types_in_context**
		|	$\text{equal\_terms}(\text{M}, \text{N}, \text{sigma})$	**equal_terms_of_type_in_context**

La transition est plutôt simple comme il suffit de transformer les expressions en fonction prenant un paramètre spécial par élément de la syntaxe. Pour donner une distinction, les types

prennent un « « au début du nom pour préciser que ce sont des types. Il faut aussi encapsuler les variable et les générique (respectivement var(x) et gen(a)) pour donner une distinction dans l'evaluation des règles. Il n'y a pas besoin de créer les règles pour les symboles true et false.

Après cela, il a fallut créer deux règles pour la sémantique dévaluation (evaluation(Context, Term, Result)) et la sémantique de typage (typage(Context, Term, Result)).

On peut prendre par exemple la fonction d'itentité représenté avec le langage système C3-PO, ainsi que sa sémantique d'évaluation et sa sémantique de typage.

Exemple 6.1: Application de la fonction identité dans la syntaxe de Système C3PO

```
func<T>(a: T) -> T {
  a
}(7)

# va retourner 7
```

Exemple 6.2: Application de la fonction identité : sémantique d'évaluation

```
evaluation([],
     app(
            func([gen(t)], [[var(a), get(t)]], gen(t), var(a)),
            [],
            [7]
     ),
     7
).
```

Exemple 6.3: Application de la fonciton identité : sémantique de typage

```
typing([],
      app(
          func([gen(t)], [[var(a), gen(t)]], gen(t), var(a)),
          [int],
          par([7])
      ),
      int
      ).
```

Le code prolog se trouve dans mon repository github. Les deux fichiers principaux sont `evaluation.pl` et `typage.pl` . Il existe une liste de testes dans ces deux fichiers.

VII. Librairie de réseaux de neurones

Maintenant que nous avons été capable d'établire un formalisme pour la création et la manipulation de nos tableaux multidimensionnels. Nous pouvons nous lancer le défi de définir la base d'une librairie de réseaux de neuronnes et voir comment notre système de type peut nous aider à créer ce type de construction. Nous n'entrerons pas en détail sur les notions comme la backpropagation.

Dans son article, Joyes Xu [5] a mentionné la possibilité de créer des réseaux de neuronnes par le biais de la programmation fonctionnelle. Nous allons faire la démonstration avec notre langage.

VII.1. Couches de réseaux de neurones

Un réseau de neurones en machine learning est un modèle computationnel inspiré du cerveau humain, conçu pour reconnaître des motifs et effectuer des tâches d'apprentissage automatique. Il est constitué de neurones artificiels, ou nœuds, organisés en couches. Chaque neurone reçoit des signaux d'entrée, les traite, et transmet une sortie aux neurones de la couche suivante. Les réseaux de neurones sont particulièrement efficaces pour des tâches complexes comme la reconnaissance d'images, la traduction de langues et la prédiction de séries temporelles.

Les couches d'un réseau de neurones peuvent être représentées à l'aide de matrices, facilitant ainsi les calculs mathématiques nécessaires pour l'apprentissage et l'inférence. Considérons un réseau de neurones simple avec une couche d'entrée, une couche cachée et une couche de sortie. La couche d'entrée reçoit les données initiales, souvent représentées par un vecteur x de dimension n, où n est le nombre de caractéristiques des données d'entrée.

Les neurones de la couche cachée effectuent des transformations linéaires sur les entrées, suivies de l'application de fonctions d'activation non linéaires. Les poids et biais de cette couche peuvent être représentés par une matrice W de dimension m * n (où m est le nombre de neurones dans la couche cachée) et un vecteur de biais b de dimension m. Le calcul des activations pour la couche cachée est donné par z = Wx + b, où z est un vecteur de dimension m.

Enfin, la couche de sortie produit la sortie finale du réseau. Si cette couche a p neurones, elle a une matrice de poids W' de dimension p * m et un vecteur de biais b de dimension p. Les sorties sont calculées de manière similaire : y = W' z + b', où y est le vecteur de sortie.

Chaque couche du réseau effectue donc une transformation linéaire suivie d'une application de fonction d'activation, ces transformations étant exprimées sous forme de multiplications matricielles et d'additions vectorielles. En empilant plusieurs couches de ce type, les réseaux de neurones peuvent modéliser des relations complexes dans les données. Les paramètres (poids et biais) de ces matrices sont ajustés durant l'entraînement du réseau via des algorithmes comme la rétropropagation, qui minimise l'erreur de prédiction en ajustant progressivement les poids et les biais.

On peut en premier lieu tenter de représenter une couche à l'aide de notre système de type. Une couche peut avoir N entrées et O sorties et peut être construite à l'aide d'une matrice $M_{\{\mathrm{NxO}\}}$ et d'un vecteur v de taille O. On peut représenter ça comme une fonction qui prend

en entrée une matrice et un vecteur puis retourn une fonction layer qui respecte ce protocole. Cette fonction prend un vecteur ligne et ne fera seulement que d'appliquer l'opération linéaire et retourner un vecteur colonne.

Exemple 7.1 : Création d'une couche de réseau de neurones

```
let NNLayer: ([N, [O, T]], [O, T]) -> [1, [O, T]] =
  func <N, O, T>(m: [N, [O, T]], b: [O, T]) {
    func <N, T>(v: [1, [N, T]]){
        plus2(dot(v, m), b)
    }
  }
```

Pour éviter que les applications faites dans les réseaux de neuronnes restent linéaires (car ceci peut entraîner le fameux « vanishing gradient »), les fonction non linéaires ont étés inventée. Nous avons notamment la fonction sigmoïde, la fonction ReLU, etc. Dans notre cas, le langage prototype que nous avons à notre disposition ne peut pas émuler ce comportement. Nous allons donc faire une fonction d'activation faussement linéaire. Le but est juste de montrer que ce type d'opération peut être typé et donc protégé.

Exemple 7.2 : Pseudo fonction sigmoïd

```
let p_sigmoid: ([N, T]) -> ([1, [N, T]]) -> [N, [1, T]] =
  func<N, T>(v: [N, T]){
    transpose(V)
  }
```

La pseudo fonction d'activation « p_sigmoid » prendra un vecteur colonne et retourner a un vecteur ligne de même longeure qui sera passé à la prochaine couche. Ici on ne fera que de transposer le vecteur par soucis de simplicité.

VII.2. Réseaux de neurones (forward propagation)

Nous pouvons facilement constater qu'une couche d'un réseau de neuronnes ainsi que la fonction d'activation peuvent être représentés comme des fonctions sur des vecteurs (de deux dimensions). Un réseaux de neuronnes n'est seulement qu'une suite d'application de fonctions. Par exemple le passage d'un vecteur sur un réseaux de 3 couches serait représenté comme ceci:

Exemple 7.3 : Combinaison de fonction à l'aide de système C3PO

```
let couche1: type = ... in
let activation1: type = ... in
let couche2: type = ... in
let activation2: type = ... in
let couche3: type = ... in
let activation3: type = ... in
  activation3(couche3(activation2(couche2(activation1(couche1(v))))))
```

Heureusement, il existe plusieurs solution syntaxique qui nous permettent de facilité la lecture d'un tel réseau. Ces solutions ne font pas directement partie du noyaux du projet mais permettrait une implémentation plus concrète du typage de réseaux. La notion d'appel de fonction uniforme qui nous permet d'avoir une représentation plus lisible à l'oeil humain à l'aide de tuyaux:

Exemple 7.4 : Combinaison de fonction à l'aide de tuyaux

```
v
  |> couche1
  |> activation1
  |> couche2
  |> activation2
  |> couche3
  |> activation3
```

Pour les personnes plus adepte du passage par message:

Exemple 7.5 : combinaison de fonction à l'aide de passage de message

```
v
.couche1()
.activation1()
.couche2()
.activation2()
.couche3()
.activation3()
```

Ces deux exemples seront transformé pour donné le premier exemple durant la compilation. Cela nous permet plus de flexibilité dans la notation. Non seulement nous sommes capable de représenter les réseaux de neuronnes de façon plus simplifié, mais nous avons aussi la garanti de contrôl au niveau de la compatibilité d'application de ces fonctions. Si la sortie d'une de ces fonction ne correspond pas à la sortie de la suivant, le compilateur sera en mesure de détecter

ce problème. Cela permet donc de cérer des réseaus de neuronnes de plus grande envergure de façon plus intrépide.

VIII. Conclusion

VIII.1. Synthèse

Dans ce projet, j'ai tenté de construir un système de type qui pourrait aider à la construction de module et librairies pour les sciences de données plus facile. Il s'est trouvé que le projet est s'est révélé plus efficace que prévu dans certains cas mais inéfficace dans d'autre. En effet avoir un système de type apporte vraiment une plus value aux projets de sciences de données dans la modélisation des tenseurs et de la sécurité sur les opérations faites dessus.

Notre langage et son système de type ont porter du fruit. Nous sommes en mesure d'exprimer des restrictions sur les tableaux multidimensionnels et exprimer les opérations qui se font dessus. Dans le cadre des sciences de données, ce type de fonctionnalité sera pratique dans l'établissement de modèles complexes.

Un autre succès vient du traitement du cas du broadcasting qui est une fonctionnalité des langages de programmations dynamique. En effet, celui-ci s'appuie beaucoup sur la conversion de type pour rendre des calculs compatibles. Nous lui avons trouvé un remplacement en implémentant des types et des opérations dédiées centrées sur la notion de matrices. Nous avons trouvé des effets satisfaisants et souvent plus proche des opérations qu'on ferait dans l'algèbre linéaire. Ceci pouvant faciliter l'application de modèles développés théoriquement. Bien sûr, notre solution s'est confrontée à des obstacles. Un typage statique est moins souple qu'un typage dynamique ce qui entraîne une perte de souplesse concernant le prototypage. Une autre limitation vient du déterminisme de notre système de type. Si nous voulons avoir l'option de l'inférence de type, il faut que le typage de nos types dépendant se limitent au nombres entiers positif et à l'arithmétique de Presburger, ce qui rends le typage de certaines fonctions (comme la fonction de linéirisation) inexprimable.

Un élément qui a été vu comme une victoire mais peut-être vu comme un echec et le fait que notre langage et son système de type ne peuvent pas exprimer de façon sécurisée le broadcasting. C'est en effet dû que celui-ci s'appuie en partie sur la transformation de type implicite (ce qui n'est pas permi pour un langage qui souhaite un minimum de sécurité pour la construction de modèles utilisable par le grand publique), mais cela reste quand même une perte d'une fonctionnalité puissante qui fait partie de la boite à outil du Scientifique des données.

VIII.2. Projets futures

Il serait intéressant d'ajouter la notion de records dans et de polymorphisme de ligne dans l'usage des dataframes. Ceci augmenterai la flexibilité tout en assurant la constructions sécurisée de module ou librairies. Il serait aussi intéressant de discuter de la meilleurs manière d'implémenter la programmation orienté objet avec notre langage et apporter une notion de mutabilité. Pour finir, il serait intéressant de developper un langage complet qui puisse transpiler dans du code R et/ou créer des binaires.

Nous avons élaboré un modèle de langage capable de manipuler des scalaires, des matrices et des vecteurs, ainsi qu'un module pour les réseaux de neurones. Nos résultats démontrent qu'il est possible de garantir un niveau de sécurité satisfaisant à l'aide de systèmes de types, malgré des défis rencontrés dans la représentation exhaustive des données sans rendre l'algo-

rithme de vérification de type indécidable dans une certaine mesure. La plus part des notions et pratiques faite dans le domaines des sciences des données ont pu être importer dans un context statiquement typé et fonctionnelle ce qui est un encouragement pour le developpement de langages futures basés sur la programmation fonctionnelle.

IX. Remerciements

Je tiens à exprimer ma profonde gratitude à toutes les personnes qui ont contribué à la réalisation de cette thèse. Leur soutien, leurs conseils et leurs encouragements ont été inestimables tout au long de ce parcours.

Tout d'abord, je remercie sincèrement Monsieur Didier Buchs, mon directeur de thèse, pour sa supervision éclairée. Son expertise et sa rigueur scientifique ont grandement enrichi ce travail.

Je suis également reconnaissant envers Dr. Damien Morard et Mr. Aurélien Coet, les membres du laboratoire SMV qui m'ont accompagné dans ce projet. Leur expertise, leur soutien moral et leurs échanges enrichissants on su me donner un cadre stable où évoluer. Leur camaraderie a rendu cette expérience de recherche plus agréable et stimulante.

Un grand merci à Mr. Alexi Turcott et Mr. Jan Vitek, membres de l'université de Northeastern University, pour leur travail de recherche sur le langage R et l'élaboration d'un système de type. Leurs suggestions constructives et leurs encouragements pour ce projet de recherche.

Je n'oublie pas Mr. John Coene, ainsi que la communauté des utilisateurs de R à Genève pour leur aide précieuse à comprendre les besoin actuels des statisticiens, scientifiques des données et constructeur de librairies. Leur expertise technique et leur feedback ont été d'une grande aide à la réalisation du design du langage.

À tous, je vous exprime ma profonde reconnaissance et mes remerciements les plus sincères.

Bibliographie

[1] A. Löh, C. McBride, et W. Swierstra, « A tutorial implementation of a dependently typed lambda calculus », *Fundam. Inform.*, vol. 102, nº 2, p. 177-207, 2010.

[2] T. Liu, *A Type System for Multidimensional Arrays*. 2020.

[3] A. Turcotte, A. Goel, F. Křikava, et J. Vitek, « Designing types for R, empirically », *Proc. ACM Program. Lang.*, vol. 4, nº OOPSLA, nov. 2020, doi: 10.1145/3428249.

[4] J. Slepak, O. Shivers, et P. Manolios, « An Array-Oriented Language with Static Rank Polymorphism », in *Programming Languages and Systems*, Z. Shao, Éd., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, p. 27-46.

[5] J. Xu, « Functional programming for deep learning — towardsdatascience.com ». 2017.