UNIVERSITÉ DE GENÈVE
FACULTÉ DES SCIENCES
Département d'informatique

Système de type

Pour les sciences des données

2024-06-24

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 présenter de mauvaise pratique pour la définition de code et la modélisation de principes théoriques.

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. Heureusement la plupart de ces langages utilisent un système de type mais (sauf dans le cas de mojo) reste assez laxiste dans l'évaluation du code et une grande partie de la gestion des erreurs et laissé au temps d'exécution. 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.

Des langages comme haskell combinés à des types dépendants (TODO références) 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 [TODO] 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 & co (Designing Types for R empirically) 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 puissant 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.

Les tableaux multidimensionnels comme les matrices ou les tenseurs ne sont que des tableaux récursivement définis (des tableaux de tableaux de tableaux, etc.).

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 concep-

tuels 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 1 —

```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 —

```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 3 —

```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 4 —

```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

[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.

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.

Exemple 5 —

```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 6 —

```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 7 —

```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 8 —

```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 & co (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 nouvelles 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.

Syntaxe

Définition 1 —

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

Évaluation

Définition 2 —

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

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

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

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.

Syntaxe

Définition 3 —

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

Évaluation

Définition 4 —

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

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

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

Typage

Définition 5 —

$$\frac{x : T \in \Gamma}{\Gamma \vdash \mathrm{x} : \mathrm{T}}\ \text{T-VAR}$$

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

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

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.

Définition 6 —

$$
\begin{array}{rrcll}
\text{terms:} & t & ::= & x & \text{variable} \\
 & & | & \lambda x : T.t & \text{abstraction} \\
 & & | & t\ t & \text{application} \\
 & & | & \lambda X.t & \text{type abstraction} \\
 & & | & t\ [T] & \text{application} \\
\text{values:} & v & ::= & \lambda x : T.t & \text{abstraction value} \\
 & & | & \lambda X.t & \text{type abstraction value} \\
\text{types:} & T & ::= & X & \text{type variable} \\
 & & | & T \to T & \text{type of functions} \\
 & & | & \forall X.T & \text{universal type} \\
\text{context:} & \Gamma & ::= & X & \text{empty context} \\
 & & | & \Gamma, \text{x:T} & \text{term variable binding} \\
 & & | & \Gamma, X & \text{type variable binding}
\end{array}
$$

Évaluation

Définition 7 —

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

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

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

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

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

Typage

Définition 8 —

$$\frac{x : T \in \Gamma}{\Gamma \vdash \mathrm{x} : \mathrm{T}}\ \text{T-VAR}$$

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

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

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

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

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 9 —

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.

Définition 9 —

$$\frac{}{n \Rightarrow n}\,\text{NUM}$$

$$\frac{\text{E1} + \text{E2} \Rightarrow \text{E3}}{\text{E1} + \text{E2} \Rightarrow \text{E3}}\,\text{PLUS}$$

$$\frac{\text{E1} * \text{E2} \Rightarrow \text{E3}}{\text{E1} * \text{E2} \Rightarrow \text{E3}}\,\text{PLUS}$$

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 `if...then...else` .

Définition 10 —

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

$$\frac{\text{E1} \Rightarrow \text{true}}{\text{if E1 then E2 else E3} \Rightarrow \text{E3}}\,\text{IF-F}$$

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.

Définition 11 —

$$\frac{}{\text{true} \Rightarrow \text{true}}\,\text{BOOL-T}$$

$$\frac{}{\text{false} \Rightarrow \text{false}}\,\text{BOOL-F}$$

$$\frac{\text{E1} \cap \text{E2} \Rightarrow \text{E3}}{\text{E1 and E2} \Rightarrow \text{E3}}\,\text{AND}$$

$$\frac{\text{E1} \cup \text{E2} \Rightarrow \text{E3}}{\text{E1 or E2} \Rightarrow \text{E3}}\,\text{OR}$$

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.

Définition 12 —

$$\frac{\Delta \vdash E_1 \longrightarrow E_1 p \quad \Delta, x = E_1 p \vdash E_2 \longrightarrow E_2 p}{\text{let } x : X = E_1 \in E_2 \longrightarrow E_2 p}\,\text{C-EV-VAR}$$

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 contexte convenable.

Les règle **C-EMP** et **C-EXT** sont des règles définissant la création et l'extension du context. Un contexte peut s'étendre en recevant un couple variable-type ajouté par le symbole « , ».

Définition 13 —

$$\frac{}{\emptyset \text{ ctxt}}\,\text{C-EMP}$$

$$\frac{\Gamma\sigma \text{ type}}{x : \sigma \text{ ctxt}}\,\text{C-EXT}$$

Le context doit admettre un opérateur d'égalité pour vérifier que deux contextes sont égaux. Les règles **C-EQ-R**, **C-EQ-S** et **C-EQ-T** définissent respectivement les propriétés de réflexivité, de substitutivité et de transitivité de l'opérateur. **C-EXT-EQ** est la règle qui assure l'équivalence entre deux contextes. Celle-ci ne fonctionne que lorsque les contextes initiaux et les types sont équivalents, amenant donc un appel récursif.

Définition 14 —

$$\frac{\Gamma \text{ ctxt}}{\Gamma = \Gamma \text{ ctxt}}\,\text{C-EQ-R}$$

$$\frac{\Gamma = \Delta \text{ ctxt}}{\Delta = \Gamma \text{ ctxt}}\,\text{C-EQ-S}$$

$$\frac{\Gamma = \Delta \text{ ctxt} \quad \Delta = \Theta \text{ ctxt}}{\Gamma = \Theta \text{ ctxt}}\,\text{C-EQ-T}$$

$$\frac{\Gamma = \Delta \text{ ctxt} \quad \Gamma\sigma = \tau \text{ type}}{\Gamma, x : \sigma = \Delta, y : \tau \text{ ctxt}}\,\text{C-EXT-EQ}$$

Les règles sur l'équivalence de contextes nous poussent aussi à définir l'opérateur d'égalité sur les types. les règle **TY-EQ-R**, **TY-EQ-S** et **TY-EQ-T** qui représentent aussi les propriétés de l'opérateur d'égalité entre les types (respectivement les propriétés de réflexivité, de substitutivité et de transitivité).

Définition 15 —

$$\frac{\Gamma\sigma \text{ type}}{\Gamma\sigma = \sigma \text{ type}} \text{TY-EQ-R}$$

$$\frac{\Gamma\sigma = \tau \text{ type}}{\Gamma\tau = \sigma \text{ type}} \text{TY-EQ-S}$$

$$\frac{\Gamma\sigma = \tau \text{ type} \quad \Gamma\tau = \rho \text{ type}}{\Gamma\sigma = \rho \text{ type}} \text{TY-EQ-T}$$

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.

Définition 16 —

$$\frac{\Gamma, x : \sigma \quad \Delta \text{ ctxt}}{\Gamma, x : \sigma, \Delta x : \sigma} \text{VAR}$$

$$\frac{\Gamma \text{ E1} : \text{T1} \qquad \Gamma, x : \text{T1 E2} : \text{T2}}{\Gamma \text{ let } x : \text{T1} = \text{E1 into E2} : \text{T2}} \text{T-LET}$$

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 10 —

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

Mais ces fonctions n'auraient pas plus d'intérêt que des valeurs.

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.

Définition 17 —

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

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.

Définition 18 —

$$\frac{\mathrm{E1} : (\overline{T} \to T) \quad \overline{a} = \overline{T} \quad \overline{E} : \overline{T}}{(\mathrm{E1}) < \overline{a} > (\overline{E}) : T}\ \text{T-FUNC-APP}$$

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.

Définition 19 —

Expression	E	::=	let x = E1 into E2	**let**
		|	func $< \overline{a} > \left(\overline{x : T}\right) \to T\{E\}$	**func**
		|	if E1 then E2 else E3	**if**
		|	E1 op E2	**bop**
		|	$(\text{E1}) < \overline{a} > \left(\overline{E}\right)$	**func_app**
		|	first(E)	**first_arr**
		|	rest(E)	**rest_arr**
		|	V	**V value**
Value	V	::=	$n \in N$	**number**
		|	true	**true**
		|	false	**false**
		|	$\left[\overline{E}\right]$	**array**
Type	T	::=	$\left(\left(\overline{T}\right) \to T\right)$	**function_type**
		|	$[n; T]$	**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**
context	ctx	::=	$\vdash \Gamma$ ctxt	**valid_context**
		|	$\Gamma \vdash \sigma$ type	**type_in_context**
		|	$\Gamma \vdash M : \sigma$	**term_of_type_in_context**
		|	$\vdash \Gamma = \Delta$ ctxt	**equal_contexts**
		|	$\Gamma \vdash \sigma = \tau$ type	**equal_types_in_context**
		|	$\Gamma \vdash M = N : \sigma$	**equal_terms_of_type_in_context**

III.7. Sémantique d'Évaluation

L'évaluation est l'une des deux parties de la sémantique du langage qui va nous permettre de voir que les opérations définies font bien ce qu'on leur demande.

Les opérations de base sont simples. Ce sont l'addition et la multiplication pour les nombre entiers (la soustraction et la division nous feraient sortir de cet ensemble si on les prend dans leur sens premier). Nous avons aussi l' opérateur and et or pour les booléens (nous somme capable de facilement créer l'opérateur unitaire **not** avec la définition de notre langage).

$$\frac{}{n \Rightarrow n}\,\text{NUM}$$

$$\frac{\text{E1} + \text{E2} \Rightarrow \text{E3}}{\text{E1} + \text{E2} \Rightarrow \text{E3}}\,\text{PLUS}$$

$$\frac{\text{E1} * \text{E2} \Rightarrow \text{E3}}{\text{E1} * \text{E2} \Rightarrow \text{E3}}\,\text{PLUS}$$

$$\frac{}{\text{true} \Rightarrow \text{true}}\,\text{BOOL-T}$$

$$\frac{}{\text{false} \Rightarrow \text{false}}\,\text{BOOL-F}$$

$$\frac{\text{E1} \cap \text{E2} \Rightarrow \text{E3}}{\text{E1 and E2} \Rightarrow \text{E3}}\,\text{AND}$$

$$\frac{\text{E1} \cup \text{E2} \Rightarrow \text{E3}}{\text{E1 or E2} \Rightarrow \text{E3}}\,\text{OR}$$

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

$$\frac{\text{E1} \Rightarrow \text{true}}{\text{if E1 then E2 else E3} \Rightarrow \text{E3}}\,\text{IF-F}$$

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

$$\frac{\text{E1} \Rightarrow \text{func} < \overline{T} > \left(\overline{x : P}\right) \rightarrow T\{E\} \quad E\left[\frac{\overline{x}}{\overline{E}}\right] \Rightarrow \text{E2}}{(\text{E1}) < \overline{T} > \left(\overline{E}\right) \Rightarrow \text{E2}}\,\text{FUNC-APP}$$

L'une des fonctionnalités cruciales de ce langage se résume dans les tableaux. En effet, c'est là où nous aborderons la notion de typage pour des opérations correctes sur ces structures de données. Nous définissions dans l'actualité l'opération de concaténation qui n'est pas une opération commutative. Celle-ci prendra un tableau et y ajoutera un élément. La définition du tableau est syntaxiquement simple. Nous pouvons comprendre qu'un tableau vide est représenté par un « [] ». C'est pourquoi il n'y a pas de définition récursive.

$$\frac{}{\left[\overline{E}\right] \Rightarrow \left[\overline{E}\right]}\,\text{ARR}$$

$$\frac{}{\left[\text{E1}\right] :: \left[\text{E2}\right] \Rightarrow \left[\overline{\text{E1}, \text{E2}}\right]}\,\text{CONC}$$

Nous avons aussi les fonctions d'extraction de valeurs. On pourrait utiliser l'indexation, mais le plus simple dans notre situation est de se limiter au minimum est d'émuler la fonction à partir des fondamentaux du langage.

$$\frac{}{\text{first}\big(\,[\text{E1}, \overline{\text{E2}}]\,\big) \Rightarrow \text{E1}}\,\text{FIRST-ARR}$$

$$\frac{}{\text{rest}\big(\,[\text{E1}, \overline{\text{E2}}]\,\big) \Rightarrow \overline{\text{E2}}}\,\text{REST-ARR}$$

III.8. Sémantique de typage

Le typage est la partie la plus importante de ce travail, c'est à partir de là qu'on va créer la logique qui nous permettra de rendre sûr les action faites avec les tenseurs et tableaux multi-dimensionnels.

Afin de pouvoir accueillir des génériques ainsi que des types dépendants. Nous avons la nécessité de définir un contexte convenable.

Les règle **C-EMP** et **C-EXT** sont des règles définissant la création et l'extension du context. Un contexte peut s'étendre en recevant un couple variable-type ajouté par le symbole « , ».

$$\frac{}{\emptyset \text{ ctxt}} \text{C-EMP}$$

$$\frac{\Gamma\sigma \text{ type}}{x : \sigma \text{ ctxt}} \text{C-EXT}$$

Le context doit admettre un opérateur d'égalité pour vérifier que deux contextes sont égaux. Les règles **C-EQ-R**, **C-EQ-S** et **C-EQ-T** définissent respectivement les propriétés de réflexivité, de substitutivité et de transitivité de l'opérateur. **C-EXT-EQ** est la règle qui assure l'équivalence entre deux contextes. Celle-ci ne fonctionne que lorsque les contextes initiaux et les types sont équivalents, amenant donc un appel récursif.

$$\frac{\Gamma \text{ ctxt}}{\Gamma = \Gamma \text{ ctxt}} \text{C-EQ-R}$$

$$\frac{\Gamma = \Delta \text{ ctxt}}{\Delta = \Gamma \text{ ctxt}} \text{C-EQ-S}$$

$$\frac{\Gamma = \Delta \text{ ctxt} \quad \Delta = \Theta \text{ ctxt}}{\Gamma = \Theta \text{ ctxt}} \text{C-EQ-T}$$

$$\frac{\Gamma = \Delta \text{ ctxt} \quad \Gamma\sigma = \tau \text{ type}}{\Gamma, x : \sigma = \Delta, y : \tau \text{ ctxt}} \text{C-EXT-EQ}$$

Les règles sur l'équivalence de contextes nous poussent aussi à définir l'opérateur d'égalité sur les types. les règle **TY-EQ-R**, **TY-EQ-S** et **TY-EQ-T** qui représentent aussi les propriétés de l'opérateur d'égalité entre les types (respectivement les propriétés de réflexivité, de substitutivité et de transitivité).

$$\frac{\Gamma\sigma \text{ type}}{\Gamma\sigma = \sigma \text{ type}} \text{TY-EQ-R}$$

$$\frac{\Gamma\sigma = \tau \text{ type}}{\Gamma\tau = \sigma \text{ type}} \text{TY-EQ-S}$$

$$\frac{\Gamma\sigma = \tau \text{ type} \quad \Gamma\tau = \rho \text{ type}}{\Gamma\sigma = \rho \text{ type}} \text{TY-EQ-T}$$

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.

$$\frac{\Gamma, x : \sigma \quad \Delta \text{ ctxt}}{\Gamma, x : \sigma, \Delta x : \sigma} \text{VAR}$$

$$\frac{\Gamma \text{ E1} : \text{T1} \qquad \Gamma, x : \text{T1 E2} : \text{T2}}{\Gamma \text{ let } x : \text{T1} = \text{E1 into E2} : \text{T2}} \text{T-LET}$$

Le typage des types primitifs est trivial et peut être fait facilement en même temps pour les nombre et les booléens.

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

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

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

Les tableaux sont le moyen de construire de nouveaux types. Ces tables ont des propriétés intéressantes. Premièrement, ils s'appuient sur des types dépendants. Leur type [n, T] permet de décrire un type de tableau de longueur **n** et de type T. Donc [2, int] qui représente les tableaux d'entier de longueur 2 est un type différent de [3, int] un tableau d'entier de longueur 3. Cela va être pratique pour préciser la dimension des tableaux et sera un outil très puissant combiné avec les génériques. La deuxième propriété réside dans sa nature récursive. Étant donné qu'un tableau [n, T] est un type, on peut construire un tableau $[n_2, [n, T]]$ qui est lui aussi un type. On le verra plus tard, mais cela peut être considéré comme l'équivalent d'une matrice.

Pour vérifier le typage d'un tableau donné, il faut vérifier deux choses: la taille de ce tableau ainsi que si tous ses membres sont du même type. C'est une autre propriété des tableaux tel que nous le définissons ici. On appel ce type de type des **invariants**.

$$\frac{\text{count}(\overline{E}) == n \quad \forall e \in \overline{E} \quad e : T}{\Gamma \left[\overline{E}\right] : [n, T]} \text{T-ARR}$$

Le typage des opérateurs de base est facile à faire. Les opérations d'addition, multiplication ainsi que l'opérateur « and » et « or » s'appliquent uniquement sur des élément du même type et ce type doit être respectivement **int** pour « + » et « * » ainsi que **bool** pour les opération **and** et **or**.

$$\frac{\text{E1} : N \quad \text{E2} : N}{\text{E1} + \text{E2} : N} \text{T-PLUS}$$

$$\frac{\text{E1} : N \quad \text{E2} : N}{\text{E1} * \text{E2} : N} \text{T-MUL}$$

$$\frac{\text{E1} : \text{bool}; \text{E2} : \text{bool}}{\text{E1 and E2} : \text{bool}} \text{T-AND}$$

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

Ces opérations sont définies par raison de convenance, le reste des fonctions peut être construit avec les définitions préalablement établies.

Nous définition la sémantique de type du « if...then...else » à l'aide de la règle **T-IF**. Cette opération est assez trivial car on doit juste s'assurer que les deux expressions retournent le même type.

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

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.

$$\frac{\overline{x : T}, E : T}{\text{func} < \overline{a} > (\overline{x : T}) \rightarrow T\{E\} : ((\overline{T}) \rightarrow T)} \text{T-FUNC}$$

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.

$$\frac{\text{E1} : (\overline{T} \rightarrow T) \quad \overline{a} = \overline{T} \quad \overline{E} : \overline{T}}{(\text{E1}) < \overline{a} > (\overline{E}) : T} \text{T-FUNC-APP}$$

IV. 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.

IV.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 11 —

```
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 12 — C'est une matrice

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

Peut-être représentée comme ceci:

Exemple 13 —

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

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

Exemple 14 —

$$\frac{[2,2] : [2,\ \text{int}]}{[[2,2],[2,2]] : [2,\ [2,\ \text{int}]]}$$

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 15 —

$$[1, 2, 3, 4] + 1 : ?$$

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 16 —

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

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 17 —

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

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

Exemple 18 —

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

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

Exemple 19 —

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

Exemple 20 —

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

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 21 —

```
let map2 = func<>(f: (T) -> U, mat: [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 22 —

$$[1, 2, 3, 4] + [4, 3, 2, 1] : ?$$

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:

Définition 20 — Définition de la fonction map_op

```
let map_op = func<T, U, N, V, M, W>(f: (T, T) -> U, tableau1: [N, V], tableau2:
[M, W]) {
    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 23 —

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

Exemple 24 —

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

Exemple 25 —

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

Exemple 26 —

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

L'addition entre deux vecteurs donnerait:

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 27 —

```
let map_op2 = func<T, U, N, V, M, W>(f: (T, T) -> U, tableau1: [N, V], tableau2:
[M, W]) {
   (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 28 —

```
map_op2(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 29 —

```
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 30 —

```
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 31 —

```
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 32 —

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

Exemple 33 —

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

La signature de cette fonction est simple à représenter:

Exemple 34 —

```
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.

La concatenation fait aussi parti des fonctionnalités utilisées en sciences des données. Cette opération se fait aussi affecter par le broadcasting (mais ce n'est pas le sujet de ce chapitre). On établit qu'on ne peut concatener que si les MDA sont de même dimension. On admet que la concaténation fait toujours une addition sur la droite

Exemple 35 —

```
concat(1, 1) faux car concat(int, int)
concat([1, 2], 1) faux car concat([2, 2], int)
concat([1, 2], [3]) juste car concat([2, int], [1, int]) -] [3, int]
concat([[1], [2]], [3, 4]) faux car concat([2, [1, int]], [2, int])
concat([[1], [2]], [[3, 4], [5, 6]]) vrai car concat([2, [1, int]], [2, [2,
int]]) -] [2, [3, int]]
```

Avec ces restrictions, nous sommes en mesure de représenter le typage concrèt de cette fonctions. (Doit faire l'implémentation de la fonction de concatenation).

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 36 —

```
let matrix = func <T, N1, N2>(dim1: N1, dim2: N2, value: T) -> [N1, [N2, T]]
let zeros = func <N1, N2>(dim1: N1, dim2: N2) -> [N1, [N2, int]]
let ones = func <N1, N2>(dim1: N1, dim2: N2) -> [N1, [N2, int]]
let trues = func <N1, N2>(dim1: N1, dim2: N2) -> [N1, [N2, int]]
let falses = func <N1, N2>(dim1: N1, dim2: N2) -> [N1, [N2, int]]
```

Exemple 37 —

```
    let length = func <T, N1, N2>(m: [[T, N2], N1]) -> int
    let shape = func <T, N1, N2>(m: [[T, N2], N1]) -> (int, int)
    let eltype = func <T, N1, N2>(m: [[T, N2], N1]) -> T
    let fill = func <T, N1, N2>(m: [[T, N2], N1], e: T) -> [[T, N2], N1]
    let linearize = func <T, N1, N2>(m1: [[T, N2], N1]) -> [T, N1*N2]
    let reshape = func <T1, T2, T3>(m1: [[T1, N2], N1], shape: T2) -> T3
```

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.

IV.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 38 —

```
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 39 —

```
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 40 —

```
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 (TODO : mentionner la source).

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 41 —

```
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 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 (TODO : citer la source).

Exemple 42 —

```
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.

Le type embedding

Une notion intéressante que j'ai découvert dans le langage de programmation de Go est l'intégration de (type embedding). C'est un concept qui favorise la composition par rapport à l'héritage.

L'héritage est un concept de la programmation orientée objet où une classe (dite sous-classe) hérite des propriétés et des méthodes d'une autre classe (dite super-classe). Cela permet de réutiliser le code existant et de créer des relations hiérarchiques entre les classes.

Dans le cadre du sous-typage, l'héritage permet à une sous-classe de se comporter comme une super-classe. Cela signifie qu'un objet de la sous-classe peut être utilisé partout où un objet de la super-classe est attendu. Le sous-typage garantit que la sous-classe respecte le contrat de la super-classe, c'est-à-dire qu'elle implémente ou redéfinit ses méthodes de manière compatible.

Bien que très utile pour la réutilisation de code, l'héritage présentes quelques fragilités qui rendent la maintenance du code difficile. L'héritage apporte un fort couplage entre les classes et leur sous-classes et peut « casser » ces dernières si les super-classes subissent des changements.

La composition est plus flexible et permet un faible couplage, augmentant ainsi la maintenabilité du code. En revanche, on perd le gain de temps assuré par la réusabilité du code promu par l'héritage. C'est là où l'intégration de type intervient.

L'intégration de type permet la réusabilité du code en utilisant la composition. On peut « intégrer » un type en faisant une référence sur celui-ci depuis le type d'accueil. À partir de là, le type intégrant « hérite » des méthode de la classe intégrée. En réalité, le type parent prend les méthodes du même nom qui font appel aux méthodes du type intégré.

Malgré tout, cette propriété ne peut pas être représentée par le système de type de mon language puisqu'il relève plus de la métaprogrammation que de l'inférence de type. En effet, comparé à l'héritage, l'inclusion de type ne provoque pas de sous-typage. Dans le langage Go, il faut passer par les interface pour avoir cette notion de substitution. A ma déception, cette fonctionnalité n'a aucune propriétés intéressantes pour le typage. Cependant il va quand même dans l'idéologie de la simplicité qui a fait de Go un langage prisé par les développeur débutants.

Je profiterai de parler plus loin des autres fonctionnalités intéressantes qui rendrait le langage plus convivial et simple d'emploi pour les utilisateurs de R, mais qui ne présentent aussi très peu d'intérêt au niveau du typage

V. Autres concepts

Dans le cadre de l'évolution de notre langage de programmation, nous exprimons un fort intérêt pour l'ajout de nouvelles fonctionnalités qui, bien que non discutées en détail ici, méritent une attention particulière. Notre focus principal a été sur les tableaux multidimensionnels et le typage des opérations associées, cependant, d'autres extensions potentiellement bénéfiques pourraient également être envisagées. Ces fonctionnalités supplémentaires s'inscriraient parfaitement dans la vision d'un futur projet de développement d'un système de types inspiré de R, reflétant sa philosophie de flexibilité et de puissance dans la manipulation de données. En intégrant ces nouvelles capacités, nous pourrions offrir une robustesse accrue et une expressivité améliorée à notre langage, alignant ainsi notre outil avec les attentes et les besoins des

utilisateurs avancés dans des domaines variés tels que les statistiques, la science des données et l'analyse quantitative.

La communauté R rencontre plusieurs défis dans l'implémentation de la programmation orientée objet (POO), principalement en raison de la diversité des systèmes de POO disponibles. Contrairement à de nombreux langages de programmation qui adoptent un modèle unique et cohérent pour la POO, R offre plusieurs paradigmes distincts : S3, S4, R6 et le récent système basé sur les classes de référence. Cette pluralité de méthodes crée un environnement où les utilisateurs peuvent être confrontés à des choix déroutants, et où les différences entre ces systèmes peuvent conduire à des incompatibilités et à des difficultés d'intégration.

Le système S3, le plus ancien et le plus simple, est apprécié pour sa flexibilité et sa simplicité d'utilisation. Cependant, il manque de rigueur formelle, ce qui peut conduire à des comportements inattendus et à une gestion d'erreurs limitée. En contraste, le système S4 introduit une formalisation stricte des classes et des méthodes, avec une vérification des types plus rigoureuse, ce qui améliore la robustesse et la maintenabilité du code. Malgré ses avantages, S4 est souvent critiqué pour sa complexité et sa courbe d'apprentissage abrupte, ce qui peut dissuader les nouveaux utilisateurs.

Le système R6, introduit pour offrir une approche plus moderne et orientée vers la performance, permet la création de classes avec des champs et des méthodes, similaires aux objets dans des langages comme Python ou Java. Bien que R6 soit plus intuitif pour ceux ayant une expérience avec d'autres langages orientés objet, il n'est pas totalement intégré avec les systèmes S3 et S4, ce qui peut poser des problèmes d'interopérabilité dans des projets complexes qui utilisent plusieurs paradigmes de POO.

Cette multiplicité de systèmes de POO dans R peut non seulement semer la confusion parmi les utilisateurs, mais aussi engendrer des discordes sur la « meilleure » approche à adopter. Les développeurs peuvent avoir des préférences fortes basées sur leur familiarité avec un système particulier, conduisant à des débats au sein de la communauté sur les pratiques optimales. De plus, la documentation et les ressources pédagogiques peuvent être fragmentées, rendant l'apprentissage et la maîtrise de la POO en R plus ardus pour les nouveaux venus.

Ces fonctionnalité pourraient amener une meilleur implémentation de la programmation orientée objet ainsi que plusieurs avantage intéressantes dans la gestion de données.

V.1. L'appel de fonction uniforme

L'appel de fonction uniforme, ou « uniform function call syntax » (UFCS), est un concept en programmation qui permet d'invoquer des fonctions de manière uniforme, indépendamment de leur définition en tant que méthodes d'une classe ou fonctions libres. Cela signifie que les fonctions peuvent être appelées de la même manière, qu'elles soient définies à l'intérieur d'une classe ou en dehors, améliorant ainsi la lisibilité et la flexibilité du code.

Par exemple, en utilisant l'UFCS, une fonction qui agit sur un objet peut être appelée comme une méthode de cet objet, même si elle n'est pas définie à l'intérieur de la classe de cet objet. Supposons qu'on ait une fonction libre `length(x)` et un objet `vec`. Avec l'UFCS, on pourrait appeler `vec.length()`, traitant `length` comme une méthode de `vec`. Cela permet

aux développeurs de choisir l'appel le plus naturel ou le plus lisible pour leur contexte, sans se soucier de la localisation de la définition de la fonction.

L'UFCS est particulièrement utile dans les langages de programmation qui favorisent la composition fonctionnelle et la modularité. Par exemple, en C++, D, et Rust, l'UFCS permet d'appeler des fonctions de manière fluide et cohérente, réduisant la confusion entre les fonctions membres et les fonctions non membres. Cela encourage également la réutilisation du code, car les fonctions libres peuvent être facilement intégrées dans des chaînes d'appels de méthodes.

En R, l'adoption de l'UFCS pourrait potentiellement unifier les diverses méthodes de manipulation des objets, rendant le code plus intuitif et cohérent. Actuellement, la pluralité des systèmes de programmation orientée objet (POO) en R (comme S3, S4, et R6) entraîne des différences dans la manière dont les fonctions sont appelées et utilisées. L'introduction de l'UFCS pourrait atténuer certaines de ces divergences en permettant un style d'appel homogène, simplifiant ainsi l'interaction avec les objets et les fonctions.

V.2. Typage nominal/structurel

Le typage nominal et le typage structurel sont deux approches distinctes pour la vérification des types dans les langages de programmation, chacune ayant ses avantages et inconvénients. Le typage nominal repose sur les noms des types pour déterminer la compatibilité entre eux. Deux types sont considérés comme compatibles si et seulement si ils portent le même nom ou si l'un est explicitement déclaré comme étant une sous-classe de l'autre. Ce système est courant dans des langages comme Java et C#. L'avantage principal du typage nominal est la clarté et la sécurité qu'il procure : les relations de type sont explicites et faciles à comprendre, réduisant les risques d'erreurs de typage accidentelles. Cependant, il peut être rigide, car il nécessite souvent des déclarations répétitives et ne permet pas la compatibilité entre types structurellement similaires mais nominalement différents, limitant ainsi la flexibilité.

En revanche, le typage structurel se base sur la forme ou la structure des types pour vérifier leur compatibilité. Deux types sont compatibles si leurs structures internes (comme les champs et les méthodes) correspondent, indépendamment de leurs noms. Cette approche est utilisée dans des langages comme TypeScript et Go. Le principal avantage du typage structurel est sa flexibilité : il permet de traiter des objets de types différents mais ayant des structures similaires de manière interchangeable, facilitant ainsi la réutilisation du code et l'intégration de composants. Toutefois, le typage structurel peut introduire des ambiguïtés et des erreurs difficiles à diagnostiquer, car des types accidentellement similaires peuvent être considérés comme compatibles, menant potentiellement à des comportements inattendus.

Comme le langage R est un langage conçu pour être flexible, il aura un système de type qui favorisera le typage structurel avec l'usage de types comme les tableaux, les tuples ou les records (qui sont mentionnés après). Il y aura tout de même l'opportunité de passer au typage nominal à l'aide du concept de named tuple.

Records

Par défaut, le langage R n'utilise pas de classe, mais émule un système similaire à l'aide de label. Le langage que nous formons n'implémentera pas de classe avec des méthodes mais

permettra de créer des types avec des fonctions dédiées. L'un des types les plus proche des classes sont les records. Ils n'ont pas le pouvoir d'héritages comme les classes mais offrent d'autres pouvoir plus intéressants.

Les records et les classes sont des concepts fondamentaux en programmation orientée objet et en gestion de données structurées, chacun ayant des avantages distincts selon le contexte d'utilisation. Les records, ou structures, sont des types de données simples qui regroupent un ensemble de champs ou de valeurs sous un même nom. Ils sont souvent utilisés pour représenter des données immuables et peuvent être comparés par leurs valeurs. Un avantage majeur des records est leur simplicité et leur efficacité pour représenter des données de manière directe et sans comportement associé. Cela les rend particulièrement adaptés pour la représentation de données de configuration, de résultats de calculs ou d'états immuables.

En revanche, les classes sont des structures plus complexes qui combinent à la fois des données (champs) et des comportements (méthodes). Elles permettent de définir des objets avec un état interne modifiable et des opérations spécifiques qui peuvent manipuler cet état. Les classes offrent une encapsulation des données et un haut niveau d'abstraction, facilitant ainsi la modélisation de concepts complexes et la gestion de l'état mutable. Un avantage clé des classes est leur capacité à encapsuler le comportement avec les données, promouvant ainsi la réutilisation du code et la modularité.

Cependant, les classes peuvent aussi introduire de la complexité, notamment en matière d'héritage et de gestion de la hiérarchie des classes. L'héritage multiple, par exemple, peut poser des défis de conception et de maintenance en introduisant des dépendances et des relations complexes entre les classes. De plus, la gestion de l'état mutable peut parfois conduire à des erreurs difficiles à diagnostiquer, telles que les problèmes de concurrence dans les environnements multi-threadés.

En comparaison, les records sont souvent plus simples à utiliser et à raisonner, mais peuvent être limités lorsque des comportements complexes doivent être associés aux données. Le choix entre records et classes dépend donc largement des besoins spécifiques du projet : les records sont souvent préférés pour la représentation de données simples et immuables, tandis que les classes sont utilisées pour modéliser des entités avec un état interne mutable et des comportements associés. En résumé, bien que les records et les classes aient des rôles distincts, ils offrent chacun des avantages significatifs en fonction du contexte d'utilisation et des exigences du projet.

Sous-typage

Le sous-typage est une part du polymorphism et a souvent été utilisé en programmation orienté objet à l'aide de l'héritage. Avec notre langage qui a une orientation fonctionnelle comme R, nous laissons ce context au profit d'autres méthodes de sous typage.

On peut mentionner l'usage d'interfaces qui permettrons à toutes les types les implémentant d'être considérés de « sous-type » car ils respectent les contrats imposés par par les dits interfaces. Ici les interfaces ne seront pas aussi puissant que dans le langage Rust mais permettrons d'implémenter des fonctions par défaut. Ils ne se baseront pas sur des génériques mais pourront être combinés pour créer de plus grandes interfaces. Nous aurons ainsi un peu plus de puissance et de flexibilité sans apporter trop de complexité.

Dans le contexte du sous-typage, un record peut aussi être considéré comme un sous-type d'un autre s'il possède tous les champs de ce dernier, éventuellement avec quelques champs supplémentaires. Par exemple, si nous avons un record `Person` avec des champs `name` et `age`, et un record `Employee` qui ajoute un champ `id` à ceux de `Person`, alors `Employee` est un sous-type de `Person`.

Le polymorphisme de ligne, quant à lui, permet de définir des types en termes de lignes de champs plutôt qu'en termes de noms de types fixes. Cela offre une flexibilité significative dans la définition et l'utilisation des types de données, car les types peuvent être étendus dynamiquement avec de nouveaux champs sans nécessiter de redéfinition explicite du type. Par exemple, un type polymorphe pourrait être défini comme ayant au moins certains champs spécifiques, mais la présence d'autres champs est autorisée et traitée de manière flexible.

L'interaction entre le sous-typage et le polymorphisme de ligne permet donc de créer des structures de données qui peuvent être étendues tout en maintenant les relations de sous-typage. Cela signifie qu'un code qui s'attend à manipuler un type plus général peut également interagir avec des instances de types plus spécifiques qui ajoutent des champs supplémentaires. Cette approche favorise la réutilisation du code et la gestion dynamique des données, car elle permet de traiter les types de manière plus générique tout en prenant en charge des variations spécifiques au sein de ces types, améliorant ainsi la flexibilité et l'expressivité du système de types.

Dataframe

Les dataframes sont des structures de données cruciales en sciences des données, particulièrement dans le langage de programmation R, en raison de plusieurs caractéristiques et fonctionnalités qui répondent aux besoins spécifiques de l'analyse de données.

Tout d'abord, les dataframes permettent de stocker et d'organiser des données tabulaires sous forme de lignes et de colonnes, où chaque colonne peut représenter une variable différente et chaque ligne correspond à une observation ou un enregistrement unique. Cette organisation est essentielle pour traiter et manipuler des ensembles de données complexes et hétérogènes provenant de diverses sources, comme des fichiers CSV, des bases de données ou des résultats d'expérimentations.

Ensuite, R est spécialement conçu pour le traitement statistique et graphique des données. Les dataframes offrent une structure de données optimisée pour les opérations statistiques courantes telles que le calcul de moyennes, de médianes, de corrélations, et la création de graphiques. Ils facilitent également la manipulation des données grâce à une large gamme de fonctions intégrées et de packages spécialisés dédiés à la manipulation, à la transformation et à l'analyse de dataframes.

De plus, les dataframes en R sont conçus pour être compatibles avec les autres outils et méthodes statistiques de la langue. Ils s'intègrent parfaitement aux fonctions de modélisation statistique, aux tests d'hypothèses, à l'analyse de variance, à la régression linéaire et non linéaire, ainsi qu'à la visualisation de données avancées. Cette intégration transparente permet aux scientifiques des données et aux analystes de travailler efficacement avec des données volumineuses tout en conservant la capacité d'appliquer des techniques statistiques sophistiquées.

Enfin, les dataframes facilitent la collaboration et le partage de données au sein d'une équipe de travail. Leur structure tabulaire standardisée et leur manipulation aisée permettent à différents analystes et chercheurs de travailler sur les mêmes données, en utilisant des méthodes cohérentes et reproductibles. Cela contribue à la transparence des analyses et à la vérifiabilité des résultats, des aspects essentiels dans le domaine scientifique et académique.

Si nous représentons les dataframes sous la forme de record de tableau (à 1 dimension) alors nous ouvrons la porte à une nouvelle façon de définir les opérations sur ces éléments. À l'aide du polymorphisme de rang discuté plus tôt, nous somme en mesure de developper des fonctions agissantes sur des dataframes aillants des colonnes spécifique et d'un type spécifique, facilitant la création de package sécurisés.

V.3. Gestion d'erreur

La gestion des erreurs est un aspect crucial de la programmation moderne, essentiel pour assurer la fiabilité, la sécurité et la robustesse des applications logicielles. Les erreurs peuvent survenir pour diverses raisons : des données d'entrée invalides, des opérations non valides, des erreurs réseau, ou encore des bugs logiciels. Traiter ces erreurs de manière efficace est nécessaire pour garantir que l'application continue de fonctionner de manière prévisible et stable, même face à des conditions imprévues.

Les sum types (ou types somme) et les union types sont des concepts fondamentaux en programmation qui permettent de représenter des valeurs qui peuvent être de différents types. Ils offrent une façon élégante et expressive de modéliser des situations où une valeur peut prendre plusieurs formes possibles. Par exemple, un type union peut représenter une valeur numérique valide ou une valeur NaN pour indiquer un calcul incorrect.

Dans le contexte de la gestion des erreurs, les union types offrent une flexibilité particulière en permettant de représenter explicitement les cas où une opération peut échouer ou retourner un résultat indéterminé comme NaN. Cela permet aux développeurs de définir des fonctions et des structures de données capables de traiter ces situations de manière cohérente et prévisible. Par exemple, au lieu de retourner une valeur incorrecte ou de provoquer une exception non contrôlée, une fonction peut retourner un type union qui indique clairement si le calcul a réussi ou a produit une valeur non valide.

Cette approche renforce également la documentation du code et l'interopérabilité avec d'autres systèmes, car elle clarifie les attentes quant aux résultats des fonctions et aux données manipulées. En outre, les union types permettent de gérer de manière efficace les valeurs manquantes ou non valides dans les analyses de données, un aspect crucial dans les applications de science des données où des données manquantes ou incorrectes sont fréquentes et doivent être gérées avec soin pour éviter des résultats incorrects ou biaisés.

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:

Expression	E	::=	$\text{let}(x, \text{E1}, \text{E2})$	**let**
		|	$\text{func}\left(\overline{a}, \overline{[x, T]}, T, E\right)$	**func**
		|	$\text{if}(\text{E1}, \text{E2}, \text{E3})$	**if**
		|	E1 op E2	**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`	::=	E1 and E2	**and**
		|	E1 or E2	**or**
		|	E1 + E2	**plus**
		|	$\text{append}(\text{E1}, \text{E2})$	**concat**
		|	E1 == E2	**equal**
		|	E1 < E2	**lower**
		|	E1 ≤ E2	**lower or equal**
		|	E1 > E2	**greater**
		|	E1 ≥ 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**

VI.1. Exemples

Exemple d'évaluation. Reprendre les exemples développés jusquà présent

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 une

librairie de réseaux de neuronnes et voir comment notre système de type peut nous aider à créer ce type de construction.

Dans son article, Joyes Xu 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 $M_{\{NxO\}}$ 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 43 —

```
let NNLayer = 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 44 —

```
let p_sigmoid 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 45 —

```
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. Vous pourrez voir le Chapitre V. la notion d'appel de fonction uniforme qui nous permettrai d'avoir une représentation plus lisible à l'oeil humain à l'aide de tuyaux:

Exemple 46 —

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

Pour les personnes plus adepte du passage par message:

Exemple 47 —

```
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.

VII.3. Réseaux de neurones (backpropagation)

Dans la phase d'entrainement, les poids et les biais sont changé à chaque fois grâce aux fonctions de perte et à la backpropagation. Les fonctions de perte calculent la différence entre la valeur cible y et la valeurs sortie par le réseau y' et retourne un nombre qui évalue la distance entre la réponse obtenue et la réponse attendue. La backpropagation est réalisée un faisant un caclule des gradiants des poids et des bias de chaque couche.

L'avantage du calcul du gradient vient du fait que les poids et les biais peuvent être modifié de façon indépendantes.

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. Références

X. 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. Alexis 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.