

## RAPPORT FINAL

---

# Programmation Récursive Primitive sur les Ensembles Purs

---

Laboratoire i3s  
Sophia Antipolis, Alpes-Maritimes, France  
1 Septembre 2019

*Auteur* : Alexandre Clément  
*Formation* : Science Informatique  
*Parcours* : Architecture Logicielle

*Encadrant* : M. Christophe PAPAŽIAN  
*Co-encadrant* : M. Grégory LAFITTE  
*Tuteur* : M. Igor LITOVSKY

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Avantages et limitations . . . . .	3
1.2	Généralisation aux ensembles . . . . .	3
1.2.1	Classe d'ensemble . . . . .	4
1.2.2	Nombre ordinal . . . . .	4
1.2.3	Ordinal limite . . . . .	5
1.2.4	Propriétés des ensembles . . . . .	5
1.3	Représentation des ensembles . . . . .	6
1.4	Description des jetons . . . . .	7
1.4.1	Fonctions initiales . . . . .	7
1.4.2	Projecteurs . . . . .	7
1.4.3	Opérateurs . . . . .	8
1.4.4	Exemples . . . . .	8
1.4.5	Variantes . . . . .	9
<b>2</b>	<b>Problématiques</b>	<b>9</b>
<b>3</b>	<b>Travail réalisé</b>	<b>10</b>
3.1	Programmes découverts . . . . .	10
3.1.1	Fonctions basiques . . . . .	10
3.1.2	Fonctions booléennes . . . . .	10
3.1.3	Fonctions arithmétiques . . . . .	11
3.1.4	Quantificateurs logique . . . . .	11
3.1.5	Opérations sur les ensembles . . . . .	12
3.1.6	Fonctions caractéristiques . . . . .	12
3.1.7	Manipulations des structures de données . . . . .	13
3.1.8	Forme normale de Cantor . . . . .	13
3.2	Interpréteur . . . . .	15
3.2.1	Jetons . . . . .	15
3.2.2	Programmes usuels . . . . .	15
3.2.3	Constantes . . . . .	15
3.2.4	Sélection des arguments . . . . .	16
3.2.5	Arbre syntaxique . . . . .	16
3.2.6	Interprétation de l'arbre syntaxique . . . . .	17
3.2.7	Fonctions . . . . .	20
3.3	Générateur . . . . .	20
3.4	Benchmark . . . . .	20
3.4.1	Exemples . . . . .	21
3.4.2	Outil de comparaison . . . . .	23
3.5	Programmes générés . . . . .	24
3.5.1	Taille 2 . . . . .	24
3.5.2	Taille 4 . . . . .	25
3.5.3	Taille 5 . . . . .	25
3.5.4	Taille 6 . . . . .	28
3.5.5	Taille 7 . . . . .	29
3.5.6	Taille 8 . . . . .	31
3.5.7	Taille 9 . . . . .	31
3.5.8	Taille 11 . . . . .	31
3.5.9	Taille 12 . . . . .	31
3.5.10	Taille 13 . . . . .	32
<b>4</b>	<b>Planning</b>	<b>33</b>
<b>5</b>	<b>Bilan</b>	<b>34</b>

## Remerciements

Je tiens à remercier toutes les personnes qui ont contribué au succès de mon stage et qui m'ont aidé lors de la rédaction de ce rapport.

Je tiens tout d'abord à remercier vivement mon encadrant de stage, M. Christophe Papazian qui m'a encadré et guidé durant ce stage.

Je remercie également M. Igor Litovsky pour avoir accepté d'être mon tuteur enseignant pour ce stage.

Enfin, je remercie M. Grégory Lafitte qui a apporté son aide au cours de ce stage.

# 1 Introduction

En calculabilité, les fonctions récursives primitives sont une classe de fonctions définies à partir de la composition et d'une récursion primitive. Ce concept a d'abord été imaginé par Richard Dedekind en 1888 avant d'être formalisé par Rózsa Péter en 1934. Les fonctions primitives ont d'abord été définies sur les entiers naturels à partir des axiomes de Peano. Les fonctions basiques étaient données à partir de ces fonctions élémentaires atomiques:

1. **Fonction constante:** la fonction constante notée  $E$  d'arité 0 est récursive primitive.
2. **Fonction successeur:** la fonction d'arité 1 qui renvoie le successeur est récursive primitive. Il s'agit de  $S : k \mapsto k + 1$ .
3. **Projecteur:** Pour tout  $n > 1$  et pour tout  $i$  avec  $1 \leq i \leq n$ , la fonction d'arité  $n$   $P_i^n$ , qui renvoie le  $i^{ieme}$  argument, est récursive primitive. Elle se note:  $P_i^n : p_1, p_2, \dots, p_{i-1}, p_i, p_{i+1}, \dots, p_n \mapsto p_i$ .

Les fonctions primitives récursives plus complexes peuvent être obtenues en appliquant les opérations données par ces opérateurs de construction élémentaires:

4. **Composition:** Étant donné une fonction primitive récursive d'arité  $k$  et  $k$  fonctions primitives récursives  $g_1, \dots, g_k$  d'arité  $m$ , la composition de  $f$  par  $g_1, \dots, g_k$  est la fonction  $h$  telle que  $h(x_1, \dots, x_m) = f(g_1(x_1, \dots, x_m), \dots, g_k(x_1, \dots, x_m))$ . La fonction  $h$  est récursive primitive.
5. **Récursion:** Étant donné une fonction primitive récursive d'arité  $k$ , et  $g$  une fonction primitive récursive d'arité  $k+2$ , la fonction  $h$  d'arité  $k+1$  est la récursion primitive de  $f$  et  $g$  i.e la fonction  $h$  est primitive récursive quand
  - $h(0, x_1, \dots, x_k) = f(x_1, \dots, x_k)$  et
  - $h(S(y), x_1, \dots, x_k) = g(y, h(y, x_1, \dots, x_k), x_1, \dots, x_k)$ .

Une grande partie des fonctions de la théorie des nombres sont récursives primitives. Par exemple, on peut définir l'addition, la division, la factorielle et toutes les fonctions mathématiques usuelles

## 1.1 Avantages et limitations

Par construction de la récursion primitive, on est assuré que tous les programmes se terminent. En revanche, cela nous restreint quant aux fonctions pouvant être calculées. On peut citer l'exemple de la fonction d'Ackermann-Peter qui est définie récursivement, mais qui n'est pas récursive primitive. Toutefois, toute fonction dont la complexité est bornée par une fonction récursive primitive est elle-même récursive primitive. Ainsi, comme la classe des fonctions récursives primitives contient les fonctions arithmétiques usuelles (polynômes, exponentielles, tours d'exponentielles ...), elle contient aussi toutes les classes de complexité usuelles, et en particulier  $P$ ,  $NP$ ,  $EXP$  et  $PSPACE$ .

## 1.2 Généralisation aux ensembles

Dans ce stage, nous allons nous intéresser aux fonctions primitives récursives appliquées aux ensembles purs de la théorie des ensembles de Zermelo-Fraenkel (ZF). Les fonctions primitives de base issues des axiomes de Peano sont ainsi remplacées par les fonctions primitives récursives issues des axiomes de ZF. La fonction projecteur et la composition sont conservées. La récursion primitive va devoir être modifiée pour correspondre aux ensembles [2].

1. **L'ensemble vide:** la fonction constante  $E$  d'arité 0 notée  $E : () \mapsto \emptyset$ .
2. **Fonction union:** la fonction d'arité 2 qui a  $(x, y)$ , renvoie l'union de  $x$  et de  $\{y\}$ . Il s'agit de  $x \cup y = x \cup \{y\}$ .
3. **Fonction d'appartenance:** la fonction d'arité 4 qui a  $(x, y, u, v)$  renvoie  $x$  si  $u \in v$ ,  $y$  sinon. Elle est notée  $if(x, y, u, v) \mapsto x$  si  $u \in v$  sinon  $y$ .

4. **Récursion:** Étant donné une fonction primitive récursive d'arité  $k + 2$ , la fonction  $h$  d'arité  $k + 1$  est la récursion primitive de  $f$  i.e la fonction  $h$  est récursive primitive tel que

$$h(z, x_1, \dots, x_k) = f\left(\bigcup_{u \in z} h(u, x_1, \dots, x_k), z, x_1, \dots, x_k\right)$$

. Cette définition de la fonction de récursion est limitée car le nombre d'appels récursifs est limité par le rang de l'entrée (voir 1.2.4 pour la définition du rang d'un ensemble). Cela a deux conséquences :

- tous les calculs utilisent des ressources finies sur les entrées héréditairement finies;
- sur les ensembles de rang infini, nos calculs sont bornés par les rangs en entrée, ce qui explique que le calcul termine toujours, même si on utilise alors une mémoire infinie et une capacité infinie de parallélisation des calculs : comme nos ensembles sont bien fondés, tous les chemins descendants dans l'ensemble sont finis.

### 1.2.1 Classe d'ensemble

Comme nous manipulons des ensembles dans notre modèle, on utilise le terme de "classe" pour référer à une collection d'objets de notre modèle, cette collection pouvant ne pas faire partie du modèle. [1] On peut alors définir toutes sortes de classes:

- **singleton:** les singletons sont les ensembles ne contenant qu'un seul élément.
- **paire:** les paires sont les ensembles contenant exactement 2 éléments.
- **couple:** les couples sont les ensembles contenant exactement 2 éléments dans un ordre déterminé. Pour représenter un couple sous forme d'ensemble, on utilise la représentation de Kuratowski [3] dans laquelle le couple  $(x, y) = \{\{x\}, \{x, y\}\}$ .
- **tuple:** à partir des couples, on peut construire des tuples de tailles quelconques. On définit les tuples de manière récursive de la façon suivante:  $(e_1, e_2, \dots, e_n) = (e_1, (e_2, \dots, e_n))$ . On transforme ainsi le tuple de taille  $n$   $(e_1, e_2, \dots, e_n)$  en un couple contenant l'élément  $e_1$  et le tuple de taille  $n - 1$   $(e_2, \dots, e_n)$ .
- **liste:** de manière analogue aux tuples, on peut définir les listes. On commence par définir la liste vide  $[] = \emptyset$ . Puis la liste contenant uniquement l'élément  $x$  comme étant le couple contenant  $x$  et la liste vide:  $[x] = (x, []) = (x, \emptyset)$ . La liste  $[x, y]$  se simplifie donc en  $(x, [y]) = (x, (y, [])) = (x, (y, \emptyset))$ . Dans le cas général, la liste  $[x_1, x_2, \dots, x_n] = (x_1, [x_2, \dots, x_n])$ .
- **relation:** les relations sont les ensembles de la forme  $\{(x, y), x \in X, y \in Y\}$  qu'on note plus généralement  $xRy \iff (x, y) \in R$ .
- **fonction:**  $R$  est une fonction si  $\forall x, y, z, xRy \wedge xRz \implies y = z$ .

### 1.2.2 Nombre ordinal

Un nombre ordinal est un ensemble qui vérifie les deux propriétés suivantes:

1. La relation d'appartenance  $\in$  est un bon ordre strict, c'est à dire:
  - **ordre strict:**  $\in$  est antiréflexive et transitive
  - **bon ordre:** toute partie non vide d'un ensemble  $\alpha$  a un plus petit élément
2. Cet ensemble est **transitif**, ce qui s'écrit:

$$\forall x \in \alpha, x \subset \alpha$$

En appliquant la définition précédente, les entiers naturels peuvent être construits de la façon suivante:

$$0 = \emptyset$$

$$n + 1 = n \cup \{n\}$$

Un entiers positif est ainsi identifié à l'aide de ses prédécesseurs:

- $0 = \{\} = \emptyset$
- $1 = \{0\} = \{\{\}\}$
- $2 = \{0, 1\} = \{\{\}, \{\{\}\}\}$
- $3 = \{0, 1, 2\} = \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}$
- $4 = \{0, 1, 2, 3\} = \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}, \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\}\}$
- etc.

Tous ces ordinaux qui correspondent aux entiers naturels sont dits héréditairement finis, cela signifie que tous ces ensembles sont finis et leurs éléments sont également héréditairement finis.

### 1.2.3 Ordinal limite

Un ordinal limite est un ordinal qui n'a pas de prédécesseur. Le premier ordinal limite est noté  $\omega$ . Il correspond à l'ensemble des entiers naturels  $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ . Remarquons que  $\omega$  est également le premier ordinal transfini (voir 1.2.4). L'ordinal qui suit est  $\omega + 1 = \omega \cup \{\omega\}$ . L'ordinal limite suivant est  $\omega + \omega = \omega \times 2$ , suivi par  $\omega \times n$  pour tout entier naturel  $n$ . À partir de la réunion de tous les  $\omega \times n$ , on obtient  $\omega \times \omega = \omega^2$ . Ce procédé peut être itéré pour produire  $\omega^3, \omega^4, \dots, \omega^\omega, \omega^{\omega^\omega}, \dots, \epsilon_0 = \omega^{\omega^{\omega^{\dots}}}, \dots$

### 1.2.4 Propriétés des ensembles

- **Clôture transitive** : La clôture transitive d'un ensemble est le plus petit ensemble transitif contenant l'ensemble de départ.
- **Descendants** : Les descendants d'un ensemble sont les éléments de sa clôture transitive.
- **Rang**: On définit le rang d'un ensemble comme étant sa profondeur, on a ainsi:

- $\text{rang}(\emptyset) = 0$  et
- $\text{rang}(x) = \sup_{u \in x} (\text{rang}(u) + 1)$

Remarquons que dans notre modèle, nos ensembles sont bien-fondés, ce qui signifie qu'ils ne peuvent pas se contenir eux-mêmes.

- **Cardinalité**: On définit la cardinalité d'un ensemble fini comme étant le nombre d'éléments qu'il contient. Plus généralement, le cardinal d'un ensemble est le plus petit ordinal équipotent à celui-ci. Remarquons que comme l'opérateur de récursion est appliqué récursivement sur les éléments d'un ensemble, on peut facilement calculer le rang de celui-ci. En revanche, on ne peut généralement pas déterminer sa cardinalité.
- **Héréditairement fini**: un ensemble est héréditairement fini s'il contient un nombre fini d'éléments et si tous ses éléments sont héréditairement finis.
- **Nombre transfini** : un nombre transfini est un ordinal qui n'est pas fini. Le plus petit nombre transfini est  $\omega$ .

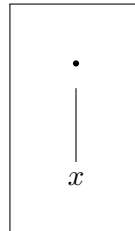
### 1.3 Représentation des ensembles

Nous allons, dans cette partie, définir une représentation sous forme d'arbre pour les ensembles. Cela nous permettra en particulier de mieux comprendre le comportement de certains programmes. Un noeud de l'arbre représente un ensemble et ses enfants représentent les éléments de cet ensemble. Les feuilles de cet arbre sont des ensembles sans éléments, par conséquent, les feuilles représentent l'ensemble vide.

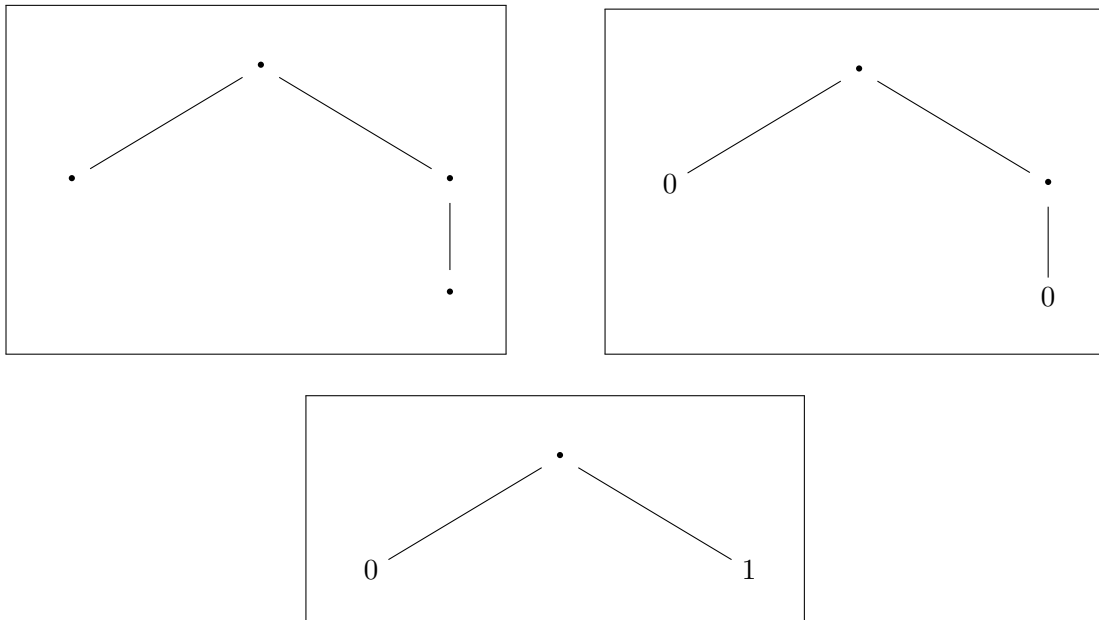
- L'ensemble  $1 = \{0\} = \{\{\}\}$  peut être représenté par



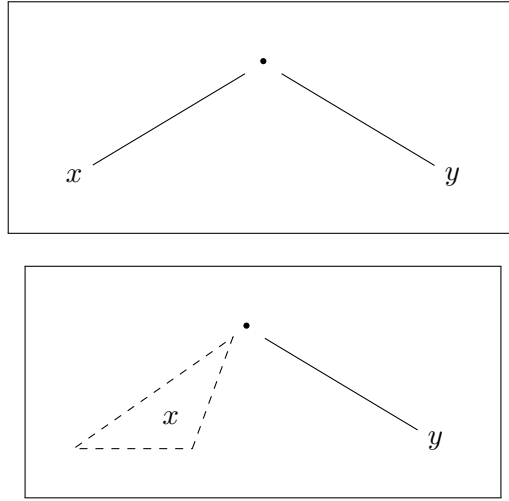
- De manière plus générale, l'ensemble singleton  $\{x\}$  peut être représenté par



- L'ensemble  $2 = \{0, 1\} = \{\{\}, \{\{\}\}\}$  peut être représenté par




- De manière plus générale, l'ensemble paire  $\{x, y\}$  peut être représenté par
- l'ensemble représentant l'opération  $x \cup y$  peut être représenté par  
 $x$  est sous-arbre de l'ensemble  $x \cup y = x \cup \{y\}$  car tous les éléments de  $x$  appartiennent à l'ensemble  $x \cup y$ .



## 1.4 Description des jetons

Les fonctions récursives primitives peuvent être représentées efficacement par des programmes écrits sous la forme de suites de jetons représentant les instructions atomiques de ce modèle de calcul.

On énumère ici les jetons de notre modèle de calcul, par exemple, le jeton  est le programme qui représente la fonction d'arité 0 notée  $E : () \mapsto \emptyset$ .


### 1.4.1 Fonctions initiales

1. Le jeton correspondant à la **Fonction constante E** se note

$$\emptyset [f_0]: () \mapsto \emptyset.$$

2. Pour des raisons pratiques, notre modèle doit comporter un jeton identité

$$\mathbf{I} [f_1]: (x) \mapsto x.$$

3. Le jeton correspondant à la **Fonction successeur** se note   $[f_2]: (x, y) \mapsto x \cup \{y\}$ .

4. On ajoute également un jeton permettant de vérifier l'appartenance d'un élément à un autre ensemble.

$$\in [f_4]: (x, y, u, v) \mapsto \text{si } u \in v \text{ alors } x \text{ sinon } y$$














### 1.4.2 Projecteurs

Le **Projecteur** est représenté par les deux jetons suivants.

$$5. \leftarrow [f_n \rightarrow f_{n+1}]: \leftarrow \mathbf{f} (x, \bar{y}) \mapsto \mathbf{f} (\bar{y})$$

$$6. \rightarrow [f_n \rightarrow f_{n+1}]: \rightarrow \mathbf{f} (\bar{y}, x) \mapsto \mathbf{f} (\bar{y})$$

**Exemple:**

-  :  $(a, b) \mapsto b$
-   :  $(a, b, c) \mapsto b$
-    :  $(a, b, c, d) \mapsto c$
-    :  $(a, b, c, d) \mapsto a$



Plus généralement, on peut construire le programme projecteur qui pour  $n$  arguments renvoie le  $i^{ieme}$ .

$$\Pi_n^i : (a_1, a_2, \dots, a_i, \dots, a_n) \mapsto a_i = \underbrace{\langle \cdot \rangle \dots \langle \cdot \rangle}_{i-1} \underbrace{\langle \cdot \rangle \dots \langle \cdot \rangle}_{n-i-1} \mathbf{I}.$$

### 1.4.3 Opérateurs

7. L'opérateur de **Composition** est représenté par le jeton suivant.

$$\mathbb{O} [f_{n+1}, f_p \times (n+1) \rightarrow f_p]$$

$$\mathbb{O} \mathbf{f} g_1 g_2 \dots g_n : (\bar{x}) \mapsto \mathbf{f} (g_1(\bar{x}), g_2(\bar{x}), \dots, g_n(\bar{x}))$$

8. L'opérateur de **Récursion** est représenté par le jeton suivant.

$$\mathbf{R} [f_{n+2} \rightarrow f_{n+1}]$$

$$\mathbf{R} \mathbf{f} : (x, \bar{y}) \mapsto \mathbf{f} \left( \bigcup_{z \in x} \mathbf{R} \mathbf{f} (z, \bar{y}), x, \bar{y} \right)$$

### 1.4.4 Exemples

1. On s'intéresse au programme d'arité 1  $\mathbb{O} \mathbb{U} \langle \cdot \rangle \emptyset \mathbf{I}$  :

$$\begin{aligned} \mathbb{O} \mathbb{U} \langle \cdot \rangle \emptyset \mathbf{I} (\emptyset) &= \mathbb{U} (\langle \cdot \rangle \emptyset (\emptyset), \mathbf{I} (\emptyset)) \\ &= \mathbb{U} (\emptyset (), \emptyset) \\ &= \mathbb{U} (\emptyset, \emptyset) \\ &= \emptyset \cup 1 \\ &= 1 \\ \mathbb{O} \mathbb{U} \langle \cdot \rangle \emptyset \mathbf{I} (1) &= \mathbb{U} (\langle \cdot \rangle \emptyset (1), \mathbf{I} (1)) \\ &= \mathbb{U} (\emptyset (), 1) \\ &= \mathbb{U} (\emptyset, 1) \\ &= \emptyset \cup \{1\} \\ &= \{1\} \\ \mathbb{O} \mathbb{U} \langle \cdot \rangle \emptyset \mathbf{I} (2) &= \mathbb{U} (\langle \cdot \rangle \emptyset (2), \mathbf{I} (2)) \\ &= \mathbb{U} (\emptyset (), 2) \\ &= \mathbb{U} (\emptyset, 2) \\ &= \emptyset \cup \{2\} \\ &= \{2\} \end{aligned}$$

On déduit que  $\mathbb{O} \mathbb{U} \langle \cdot \rangle \emptyset \mathbf{I} : z \mapsto \{z\}$

On s'intéresse au programme d'arité 1  $\mathbf{R} \cup$  :

$$\begin{aligned}
\mathbf{R} \cup (\emptyset) &= \cup (\bigcup_{u \in \emptyset} \mathbf{R} \cup (u), \emptyset) \\
&= 1 \\
\mathbf{R} \cup (1) &= \cup (\bigcup_{u \in 1} \mathbf{R} \cup (u), 1) \\
&= \cup (\mathbf{R} \cup (\emptyset), 1) \\
&= \cup (1, 1) \\
&= 2 \\
\mathbf{R} \cup (\{1\}) &= \cup (\bigcup_{u \in \{1\}} \mathbf{R} \cup (u), \{1\}) \\
&= \cup (\mathbf{R} \cup (1), \{1\}) \\
&= \cup (2, \{1\}) \\
&= \{\emptyset, 1, \{1\}\}
\end{aligned}$$

Pour chaque récursion, le programme  $\mathbf{R} \cup$  ajoute l'élément  $z$  à l'union des appels du programme  $\mathbf{R} \cup$  sur les éléments de  $z$ . Ainsi, on en déduit que le  $\mathbf{R} \cup$  construit récursivement un ensemble contenant tous les éléments de  $z$  et de ses descendants. Cette fonction permet donc de calculer la clôture transitive d'un ensemble (voir 1.2.4).

$\mathbf{R} \cup : (x) \mapsto \mathbf{TC}(\{x\})$  où  $\mathbf{TC}$  est la clôture transitive.

#### 1.4.5 Variantes

Certains des jetons définis peuvent être dérivé sous différentes formes. On peut par exemple nous intéresser au cas du jeton  $\in$  qui peut être transformé sous forme d'opérateur tel que

$$\in \mathbf{f} \mathbf{g} : (\bar{x}, u, v) \mapsto \mathbf{f}(\bar{x}, u, v) \text{ si } u \in v \text{ sinon } \mathbf{g}(\bar{x}, u, v)$$

## 2 Problématiques

L'objet de ce stage est donc l'étude pratique et théorique de l'ensemble des programmes récursifs primitifs sur les ensembles purs par les suites de jetons. Une première piste de recherche est d'exhiber concrètement les programmes de calcul arithmétique ou de reconnaissance de propriétés particulières (comme la reconnaissance de la finitude d'une entrée). Certains de ces programmes ont leur existence démontrée dans la littérature mais n'ont jamais été explicitement écrits et simulés. Un des autres objectifs est de mieux comprendre les limitations de ce modèle de calcul, en particulier en le comparant avec les modèles Turing-Complet ou les machines de Turing à temps infini. En effet, par exemple sur les ordinaux, selon la manière dont on choisit leur codage, la reconnaissance ou l'écriture des ordinaux peut avoir différentes limites.

### 3 Travail réalisé

#### 3.1 Programmes découverts

##### 3.1.1 Fonctions basiques

On commence par définir les programmes de bases permettant de manipuler les ensembles de notre modèle.

1. Le programme qui construit le singleton de l'entrée.

$$\{\cdot\}: z \mapsto \{z\} = \text{⊞} \text{⊔} \text{▶} \text{∅} \text{I}$$

2. Le programme qui construit l'ensemble contenant les deux éléments en entrée.

$$\{\cdot, \cdot\}: x, y \mapsto \{x, y\} = \text{⊞} \text{⊔} \text{▶} \{\cdot\} \text{◀} \text{I}$$

3. Le programme qui construit l'ensemble contenant le couple d'éléments en entrée.

$$(\cdot, \cdot): x, y \mapsto (x, y) = \{\{x\}, \{x, y\}\} = \text{⊞} \{\cdot\} \text{▶} \{\cdot\} \{\cdot\}$$

4. Le programme "map" prend un programme  $\mathbf{p}$  en paramètre et transforme chaque élément de l'entrée en leurs appliquant le programme  $\mathbf{p}$ .

$$\mathbf{M} \mathbf{p} : z \mapsto \{\mathbf{p}(u), u \in z\} \text{ i.e } \mathbf{M} \text{ applique le programme } \mathbf{p} \text{ sur chaque élément de } z.$$

$$= \text{⊞} \mathbf{R} \text{⊞} \text{⊞} \text{⊞} \text{◀} \text{▶} \text{⊞} \{\cdot\} \mathbf{p} \text{▶} \text{▶} \text{I} \text{◀} \text{▶} \text{I} \text{◀} \text{◀} \text{I} \text{I} \text{I}$$

5. Le programme "filter" prend un programme  $\mathbf{p}$  en paramètre et pour chaque élément de l'entrée, l'élément est conservé si le programme  $\mathbf{p}$  renvoie 1 pour cet élément.

$$\mathbf{F} \mathbf{p} : z \mapsto \{u, u \in z \wedge \mathbf{p}(u)\} \text{ i.e } \mathbf{F} \text{ conserve les éléments de } z \text{ qui sont vrais par } \mathbf{p}.$$

$$= \text{⊞} \mathbf{R} \text{⊞} \text{⊞} \text{⊞} \text{◀} \text{▶} \text{⊞} \text{⊞} \{\cdot\} \text{◀} \text{∅} \text{◀} \text{∅} \mathbf{p} \text{▶} \text{▶} \text{I} \text{◀} \text{▶} \text{I} \text{◀} \text{◀} \text{I} \text{I} \text{I}$$

6. Le programme qui construit l'union des deux ensembles en entrée.

$$\cup: a, b \mapsto a \cup b = \text{⊞} \mathbf{R} \text{⊞} \text{⊞} \text{⊞} \text{⊞} \text{⊔} \text{◀} \text{◀} \text{◀} \text{I} \text{◀} \text{▶} \text{I} \text{▶} \text{▶} \text{I} \text{◀} \text{▶} \text{▶} \text{I} \text{◀} \text{◀} \text{I} \text{I} \text{I}$$

7. Forme généralisée: construit l'union des éléments d'un ensemble.

$$\bigcup: z \mapsto \bigcup z = \text{⊞} \mathbf{R} \text{⊞} \text{⊞} \text{⊞} \text{◀} \text{▶} \text{I} \text{▶} \text{▶} \text{I} \text{◀} \text{▶} \text{I} \text{◀} \text{◀} \text{I} \text{I} \text{I}$$

8. Le programme qui construit l'intersection des deux ensembles en entrée.

$$\cap: a, b \mapsto a \cap b = \text{⊞} \mathbf{R} \text{⊞} \text{⊞} \text{⊞} \text{⊞} \text{⊞} \text{◀} \text{▶} \text{▶} \text{I} \text{◀} \text{◀} \text{◀} \text{◀} \text{∅} \text{◀} \text{▶} \text{▶} \text{I} \text{◀} \text{◀} \text{◀} \text{I} \text{I} \text{I}$$

9. Forme généralisée: construit l'intersection des éléments d'un ensemble.

$$\bigcap : z \mapsto \bigcap z$$

$$= \text{⊞} \mathbf{F} \text{⊞} \text{⊞} \text{⊞} \text{◀} \text{◀} \text{∅} \text{◀} \text{◀} \text{1} \text{◀} \text{◀} \text{∅} \text{⊞} \mathbf{M} \text{⊞}$$

##### 3.1.2 Fonctions booléennes

On code un booléen avec  $\emptyset = \text{False}$  et  $1 = \text{True}$ . On peut alors définir les fonctions usuelles sur les booléens.

1. **not**:  $z \mapsto 0$  si  $z$  sinon  $1$  =  $\text{⊞} \text{⊞} \text{◀} \text{∅} \text{◀} \text{1} \text{◀} \text{∅} \text{I}$

2. **and**:  $a, b \mapsto a \wedge b = \text{⊞} \text{⊞} \text{◀} \text{⊞} \text{⊞} \text{◀} \text{1} \text{◀} \text{∅} \text{◀} \text{∅} \text{I} \text{◀} \text{◀} \text{∅} \text{◀} \text{◀} \text{∅} \text{▶} \text{I}$

3. **∧**:  $z \mapsto \wedge z = \text{⊞} \text{⊞} \text{◀} \text{∅} \text{⊞} \text{⊞} \text{◀} \text{1} \text{◀} \text{∅} \text{◀} \text{1} \text{I} \text{◀} \text{∅} \text{I}$

4.  $\text{ou} : a, b \mapsto a \vee b =$
5.  $\text{v} : z \mapsto \forall z =$

### 3.1.3 Fonctions arithmétiques

Le programme qui construit le successeur de l'entrée dans le cas où celle-ci est un ordinal.

$$\text{S} : z \mapsto z \cup \{z\} =$$

En appliquant le programme  $\text{S}$ , on peut construire les ordinaux finis de la façon suivante :

- $0 = \{\}$
- $n + 1 = \text{S}(n)$

On peut alors définir un certain nombre de fonctions arithmétiques utilisant les ordinaux, en particulier les opérations arithmétiques mais aussi le rang d'un ensemble. Notons bien que ces fonctions arithmétiques doivent être aussi bien valides dans le cas héréditairement fini que dans le cas transfini.

6.  $\text{P} : z \mapsto z - 1 =$

7.  $\text{+} : \alpha, \beta \mapsto \alpha + \beta = \sup_{\gamma < \beta} (\alpha + \gamma + 1) \text{ si } \beta > 0 \text{ sinon } \alpha$   
 $=$

8.  $\text{-} : \alpha, \beta \mapsto \alpha - \beta = \sup_{\gamma \leq \alpha} (\beta + \gamma \leq \alpha)$   
 $=$

9.  $\text{x} : \alpha, \beta \mapsto \alpha \times \beta = \sup_{\gamma < \beta} (\alpha \times \gamma + \alpha)$   
 $=$

10.  $\text{/} : \alpha, \beta \mapsto \alpha / \beta = \sup_{\gamma \leq \alpha} (\beta \times \gamma \leq \alpha)$   
 $=$

11.  $\alpha^\beta : \alpha, \beta \mapsto \alpha^\beta = \sup_{\gamma < \beta} (\alpha^\gamma \times \alpha) \text{ si } \beta > 0 \text{ sinon } 1$   
 $=$

12.  $\text{log} : \alpha, \beta \mapsto \log_\beta(\alpha) = \sup_{\gamma \leq \alpha} (\beta^\gamma \leq \alpha)$   
 $=$

### 3.1.4 Quantificateurs logique

On peut rajouter deux nouveaux opérateurs correspondant aux deux opérateurs logique classique.

- Le programme qui prend en paramètre un programme  $\text{P}$  et qui renvoie vrai si  $\text{P}$  renvoie vrai pour tout les éléments de l'ensemble d'entrée.

$$\forall \text{P} : z \mapsto \forall u \in z, \text{P}(u) \text{ is True} =$$

- Le programme qui prend en paramètre un programme  $\mathbf{P}$  et qui renvoie vrai si  $\mathbf{P}$  renvoie vrai pour au moins un élément de l'ensemble d'entrée.

$$\mathbf{\exists P} : z \mapsto \exists u \in z, \mathbf{P}(u) \text{ is True} = \mathbf{\omega} \mathbf{\forall} \mathbf{M} \mathbf{P}$$

### 3.1.5 Opérations sur les ensembles

1. Le programme qui renvoie vrai si le premier ensemble contient le second.

$$\mathbf{\in} : a, b \mapsto a \in b = \mathbf{\omega} \mathbf{\in} \mathbf{\leftarrow} \mathbf{\leftarrow} \mathbf{1} \mathbf{\leftarrow} \mathbf{\leftarrow} \mathbf{\emptyset} \mathbf{\rightarrow} \mathbf{I} \mathbf{\leftarrow} \mathbf{I}$$

2. Le programme qui renvoie vrai si le premier ensemble ne contient pas le second.

$$\mathbf{\notin} : a, b \mapsto a \notin b = \mathbf{\omega} \mathbf{R} \mathbf{\in} \mathbf{\leftarrow} \mathbf{\leftarrow} \mathbf{1} \mathbf{\rightarrow} \mathbf{I} \mathbf{\leftarrow} \mathbf{I}$$

3. Le programme qui renvoie vrai si le premier ensemble est inclut dans le second.

$$\mathbf{\subseteq} : a, b \mapsto a \subseteq b = \mathbf{\forall} \mathbf{\in}$$

4. Le programme qui renvoie vrai si le premier ensemble est égal au second.

$$\mathbf{=} : a, b \mapsto a = b = \mathbf{\omega} \mathbf{\in} \mathbf{\leftarrow} \mathbf{\leftarrow} \mathbf{1} \mathbf{\leftarrow} \mathbf{\leftarrow} \mathbf{\emptyset} \mathbf{\rightarrow} \mathbf{I} \mathbf{\cup}$$

5. Le programme qui renvoie vrai si le premier ensemble n'est pas égal au second.

$$\mathbf{\neq} : a, b \mapsto a \neq b = \mathbf{\omega} \mathbf{R} \mathbf{\in} \mathbf{\leftarrow} \mathbf{\leftarrow} \mathbf{1} \mathbf{\leftarrow} \mathbf{I} \mathbf{\cup}$$

6. Le programme qui construit la différence des deux ensembles en entrée.

$$\mathbf{-} : a, b \mapsto a - b = \mathbf{F} \mathbf{\notin}$$

7. Le programme qui construit la différence symétrique des deux ensembles en entrée.

$$\begin{aligned} \mathbf{\Delta} & : a, b \mapsto a \Delta b \\ &= \mathbf{\omega} \mathbf{-} \mathbf{\cup} \mathbf{\cap} \\ &= \mathbf{\omega} \mathbf{F} \mathbf{\omega} \mathbf{ou} \mathbf{\omega} \mathbf{\notin} \mathbf{\leftarrow} \mathbf{\rightarrow} \mathbf{I} \mathbf{\rightarrow} \mathbf{\rightarrow} \mathbf{I} \mathbf{\omega} \mathbf{\notin} \mathbf{\leftarrow} \mathbf{\leftarrow} \mathbf{I} \mathbf{\rightarrow} \mathbf{\rightarrow} \mathbf{I} \mathbf{\cup} \mathbf{\rightarrow} \mathbf{I} \mathbf{\leftarrow} \mathbf{I} \end{aligned}$$

8. Le programme qui supprime le deuxième ensemble du premier ensemble donné en entrée.

$$\mathbf{\setminus} : a, b \mapsto a \setminus b = \mathbf{F} \mathbf{\neq}$$

9. Le programme qui construit l'ordinal égal au rang de l'ensemble en entrée.

$$\mathbf{K} : z \mapsto \text{rang}(z) = \mathbf{\omega} \mathbf{P} \mathbf{R} \mathbf{\rightarrow} \mathbf{R} \mathbf{\cup}$$

### 3.1.6 Fonctions caractéristiques

Pour la suite, nous allons avoir besoin de fonctions caractéristiques reconnaissant des propriétés sur les ensembles.

1. Programme qui renvoie vrai si l'ensemble en entrée est transitif.

$$\mathbf{T} : z \mapsto z \text{ est transitif} \iff z \mapsto \forall u \in z, u \subseteq z = \mathbf{\omega} \mathbf{\forall} \mathbf{\subseteq} \mathbf{I} \mathbf{I}$$

2. Programme qui renvoie vrai si l'ensemble en entrée est un ordinal.

$$\mathbf{O} : z \mapsto z \text{ est un ordinal} \iff z \mapsto \forall u \in z, u \text{ est transitif et } u \text{ est un ordinal} = \mathbf{R} \mathbf{\omega} \mathbf{\omega} \mathbf{\wedge} \mathbf{\{..\}} \mathbf{\rightarrow} \mathbf{I} \mathbf{\leftarrow} \mathbf{T}$$

3. Programme qui renvoie vrai si l'ensemble en entrée est un singleton.

$$\mathbf{\{..\}} : z \mapsto z \text{ est un singleton} = \mathbf{\omega} \mathbf{\wedge} \mathbf{\omega} \mathbf{M} \mathbf{\omega} \mathbf{\wedge} \mathbf{\omega} \mathbf{M} \mathbf{=} \mathbf{\leftarrow} \mathbf{I} \mathbf{\rightarrow} \mathbf{I} \mathbf{I} \mathbf{I}$$

4. Programme qui renvoie vrai si l'ensemble en entrée est une paire.

$$\{ \cdot \} : z \mapsto z \text{ est un singleton ou } z \text{ est une paire} = \omega \wedge \omega M \omega \wedge M \omega \in \leftarrow \leftarrow 1$$

$$\omega \wedge M \omega \omega R \omega \{ \cdot \} \omega = \rightarrow \rightarrow I \leftarrow \rightarrow I \omega = \rightarrow \rightarrow I \leftarrow \leftarrow I \leftarrow \rightarrow I \leftarrow$$

$$\leftarrow I \rightarrow \rightarrow I \leftarrow \leftarrow \leftarrow \emptyset \omega = \rightarrow \rightarrow I \leftarrow \leftarrow I \leftarrow \rightarrow I \leftarrow \leftarrow I \rightarrow \rightarrow I I I I$$

5. Programme qui renvoie vrai si l'ensemble en entrée est un couple.

$$\{ \cdot \} : z \mapsto z \text{ est un couple} = \omega \text{ and } \omega \text{ and } \omega \text{ and } \{ \cdot \} \omega \in \cup I \exists \{ \cdot \} \omega \in \leftarrow \emptyset \leftarrow 1 \leftarrow$$

$$\emptyset I$$

6. Programme qui renvoie vrai si l'ensemble en entrée est une relation.

$$z \mapsto z \text{ est une relation} = \forall \{ \cdot \}$$

7. Programme qui renvoie vrai si l'ensemble en entrée est une fonction.

$$z \mapsto z \text{ est une fonction} = \text{and } \forall \{ \cdot \} \omega \forall \omega \notin \rightarrow \downarrow \omega \cup \omega M \downarrow \omega \setminus \leftarrow I \rightarrow I I$$

$$I$$

8. Programme qui renvoie vrai si l'ensemble en entrée est un ordinal limite.

$$\infty? : z \mapsto z \text{ est un ordinal limite} = \omega \wedge \omega \{ \cdot \} O \omega \wedge \omega \{ \cdot \} \omega \neq I \leftarrow \emptyset \omega \forall \omega \neq$$

$$\rightarrow S \leftarrow I I I$$

9. Programme qui renvoie vrai si l'ensemble en entrée est  $\omega$ .

$$\omega? : z \mapsto z = \omega = \omega \wedge \omega \{ \cdot \} \forall \omega \text{ not } \infty? \infty?$$

10. Programme qui renvoie  $\omega$  si l'ensemble en entrée contient  $\omega$ , sinon renvoie 0.

$$\downarrow \omega : z \mapsto \omega \text{ si } \omega \in z \text{ sinon } 0 = \omega \in I \omega \cup F \omega? \leftarrow \emptyset \omega?$$

Remarquons que sur les ordinaux, cela équivaut à  $z \mapsto \omega$  si  $z \geq \omega$  sinon 0.

11. Programme qui renvoie le  $\log$  en base  $\omega$  de l'ordinal en entrée.

$$\log_{\omega} : z \mapsto \log_{\omega}(z) = \omega \omega \in \omega \log \rightarrow I \leftarrow I \leftarrow \leftarrow \emptyset \leftarrow \leftarrow \emptyset \leftarrow I I \downarrow \omega$$

### 3.1.7 Manipulations des structures de données

1. Programme renvoie le premier élément du couple en entrée.

$$\downarrow \cdot : z = (a, b) \mapsto a = \omega \cup \omega \cup F \{ \cdot \}$$

2. Programme renvoie le second élément du couple en entrée.

$$\downarrow \cdot : z = (a, b) \mapsto b = \omega \omega \in \leftarrow I \omega \setminus \rightarrow \cup \leftarrow I \leftarrow \leftarrow \emptyset \rightarrow \{ \cdot \} I \downarrow \cdot$$

### 3.1.8 Forme normale de Cantor

Tout ordinal peut s'écrire de manière unique sous la forme  $\omega^{\beta_1} \times \gamma_1 + \dots + \omega^{\beta_k} \times \gamma_k$  où  $k$  est un entier naturel,  $\gamma_1, \dots, \gamma_k$  sont des entiers positifs et  $\beta_1 > \beta_2 > \dots > \beta_k \geq 0$  sont des ordinaux. Cette décomposition se nomme la forme normale de Cantor.

1.  $z \mapsto \mathbf{P}(z, z, \omega)$  si  $z \geq \omega$  sinon  $\{(0, z)\}$ .

$$fn_1 \mathbf{P} = \omega \omega \in \mathbf{P} \omega \{ \cdot \} \omega ( \cdot ) \leftarrow \leftarrow \leftarrow \emptyset \leftarrow \rightarrow I \leftarrow \leftarrow \leftarrow \emptyset \leftarrow \leftarrow I I I \downarrow \omega$$

Ce programme permet d'extraire  $\omega$  de l'entrée si celle-ci contient ou est égale à  $\omega$ . Cela nous permettra par la suite de calculer le  $\log$  en base  $\omega$ . Si l'entrée est finis, alors on ne peut pas extraire  $\omega$  de celle-ci mais on connaît la forme normale de Cantor des ordinaux finis:  $\alpha = \omega^0 \times \alpha$ . On peut donc renvoyer le couple  $(0, \alpha)$  dans le cas où  $\alpha < \omega$ .



2.  $(x, y, z, \omega) \mapsto \mathbf{p}(z, \omega)$  si  $y = \emptyset$  sinon  $\mathbf{q}(x, \omega)$ .

$$fn_2 \mathbf{p} \mathbf{q} = \omega \in \triangleright \triangleright \mathbf{p} \omega \mathbf{q} \triangleright \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \mathbf{I} \triangleleft \triangleleft \triangleleft \triangleleft \emptyset \triangleleft \triangleright \triangleright \{\cdot\}$$

Ce programme s'inscrit dans un opérateur de récursion et il sert à appeler le programme  $\mathbf{p}$  lors du premier appel récursif puis le programme  $\mathbf{q}$  pour les autres appels. Le programme  $\mathbf{p}$  sert alors de programme d'initialisation pour la récursion.

3.  $(z, \omega) \mapsto (\{(\beta_1, \gamma_1)\}, R_1)$  où  $\beta_1 = \log_\omega(z)$ ,  $\gamma_1 = \frac{z}{\omega^{\beta_1}}$  et  $R_1 = z - \omega^{\beta_1} \times \gamma_1$ .

$$fn_3 = \omega \omega \omega \omega (\cdot\cdot) \omega \{\cdot\} \omega (\cdot\cdot) \triangleleft \triangleleft \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \triangleleft \mathbf{I} \omega - \triangleright \triangleright \triangleright \triangleright \mathbf{I} \omega \\ \times \omega \triangleleft \triangleleft \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \triangleleft \mathbf{I} \triangleright \triangleright \triangleright \mathbf{I} \triangleleft \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \mathbf{I} \omega \\ / \triangleright \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \mathbf{I} \triangleright \triangleright \mathbf{I} \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \mathbf{I} \omega \alpha^\beta \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \mathbf{I} \triangleright \mathbf{I} \triangleleft \\ \mathbf{I} \log$$

Ce programme permet d'initialiser le calcul de la forme normale de Cantor en calculant le premier terme et en renvoie un couple contenant la solution partielle ainsi que le reste partiel.

4.  $(x, \omega) \mapsto (\{(\beta_1, \gamma_1), \dots, (\beta_{k+1}, \gamma_{k+1})\}, R_{k+1})$  où  $x$  est de la forme  $(\{(\beta_1, \gamma_1), \dots, (\beta_k, \gamma_k)\}, R_k)$  et  $\beta_{k+1} = \log_\omega(R_k)$ ,  $\gamma_1 = \frac{R_k}{\omega^{\beta_{k+1}}}$  et  $R_{k+1} = R_k - \omega^{\beta_{k+1}} \times \gamma_{k+1}$ .

$$fn_4 = \omega \omega \omega \omega \omega (\cdot\cdot) \omega \cup \triangleright \triangleright \triangleright \triangleright \triangleright \mathbf{I} \omega (\cdot\cdot) \triangleleft \triangleleft \triangleleft \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \triangleleft \\ \mathbf{I} \omega - \triangleleft \triangleright \triangleright \triangleright \triangleright \triangleright \mathbf{I} \omega \times \omega \triangleleft \triangleleft \triangleleft \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \triangleleft \mathbf{I} \triangleleft \triangleright \triangleright \triangleright \mathbf{I} \\ \triangleleft \triangleright \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \triangleleft \mathbf{I} \omega / \triangleleft \triangleright \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \\ \triangleleft \triangleleft \mathbf{I} \triangleright \triangleright \mathbf{I} \triangleleft \triangleright \triangleright \mathbf{I} \triangleleft \triangleleft \mathbf{I} \triangleleft \triangleleft \mathbf{I} \triangleleft \triangleleft \mathbf{I} \omega \alpha^\beta \triangleleft \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \mathbf{I} \\ \triangleright \triangleright \mathbf{I} \triangleleft \triangleright \mathbf{I} \triangleleft \triangleleft \mathbf{I} \triangleleft \log \triangleright (\cdot\cdot) \triangleright (\cdot\cdot) \triangleleft \mathbf{I}$$

Ce programme utilise le reste partiel précédemment calculé afin de calculer le terme suivant de la forme normale de Cantor. Ce terme sera ajouté à la solution partielle et un nouveau reste sera calculé.

5. Programme calculant la forme normale de Cantor.

$$\mathbf{FN} : \alpha \mapsto \{(\beta_1, \gamma_1), \dots, (\beta_k, \gamma_k)\} \text{ tel que } \alpha = \omega^{\beta_1} \times \gamma_1 + \dots + \omega^{\beta_k} \times \gamma_k \\ = fn_1 \omega (\cdot\cdot) \mathbf{R} fn_2 fn_3 fn_4$$

## 3.2 Interpréteur

L'interpréteur a été développé pour nous aider à vérifier la validité d'un programme, mais aussi à visualiser le comportement de ceux-ci. Ce point devient important lorsque l'on va étudier un grand nombre de programmes générés, l'interpréteur facilitera l'étude de ces programmes.

L'interpréteur se décompose en deux grandes parties:

- une première étape de parsing et de construire de l'arbre syntaxique,
- une seconde étape d'interprétation de l'arbre syntaxique.

### 3.2.1 Jetons

Notre interpréteur utilise une grammaire simple de seulement 8 <sup>1</sup> jetons définis en 1.4:










								 <sup>a</sup>
E	I	+	?	<	>	o	R	!

Table 1: Table de correspondance entre les jetons et leur caractère ASCII associé.






<sup>a</sup>Ce jeton n'est pas nécessaire dans notre modèle, il a été ajouté pour fournir un exemple étudiable de variante de jeton composant notre modèle.

### 3.2.2 Programmes usuels

Pour faciliter l'expressivité de l'utilisateur, cette grammaire a été enrichie avec des instructions plus complexes pouvant être décomposées récursivement jusqu'à obtenir une suite de jetons.

Par exemple, on peut définir l'instruction

singleton

qui représente la fonction  $(x) \mapsto \{x\}$ , et comme étant égale à     . Ainsi, l'interpréteur transformera automatiquement le mot "singleton" en la séquence de jetons correspondante.

On peut de même définir le mot

pair

qui représente la fonction  $(x, y) \mapsto \{x, y\}$ , et comme étant égale à

o+> singleton <I

Après simplification du mot singleton, l'interpréteur lira          .



### 3.2.3 Constantes

De la même façon que l'on a défini des programmes usuels, nous avons ajouté les ordinaux finis au langage. L'interpréteur est donc capable de lire

o singleton 2

comme étant le programme  $P : () \mapsto \{2\}$  et qui équivaut à

o singleton o successeur o successeur E

<sup>1</sup>La grammaire dispose actuellement de 9 jetons car nous avons ajouté le jeton opérateur  dérivant du jeton .



qui se simplifie en . Par la suite, on le notera simplement .

D'une manière générale, on a  $n = \underbrace{\text{composition } S \dots \text{composition } S}_n \text{ empty set}$ .

### 3.2.4 Sélection des arguments

Pour simplifier l'écriture de programme dont l'arité peut être grande, on ajoute également un opérateur de sélection des arguments. Sa syntaxe est la suivante :

`select  $a_i$  among  $n$  for  $P$`

où  $a_i$  est le paramètre voulu,  $n$  est le nombre d'arguments attendus par le programme final et  $P$  est le programme d'arité 1 qui prend  $a_i$  en argument.

Par exemple, l'entrée

`select 2 among 4 for singleton`

est transformé en , et elle équivaut au programme  $(x_0, x_1, x_2, x_3) \mapsto \{x_2\}$ . Cet opérateur peut être étendue pour sélectionner plusieurs paramètres. On peut donc écrire

`select 0 2 among 4 for +`

qui se transforme en .

Notons que l'arité du programme cible correspond toujours au nombre d'arguments sélectionnés.

Pour finir, dans le cas où l'on désire sélectionner une plage d'argument, par exemple, nous voulons tous les arguments sauf le deuxième, alors on peut écrire

`select 0 2 ... among 5 for ?`

est équivalent à

`select 0 2 3 4 among 5 for ?`

et devient .

Un dernier cas particulier a été ajouté: il est possible de faire

`select none among  $n$  for  $P$`

pour ne sélectionner aucun argument. Le programme  $P$  est alors obligatoirement d'arité 0.

### 3.2.5 Arbre syntaxique

Chaque jeton est converti en un noeud qui lui est associé: les fonctions initiales , , et sont respectivement converties en un noeud EmptySet, Identity, UnionPlus et IfThenElse. Les opérateurs et sont représentés par un noeud Projection. Ce noeud projection contient un autre noeud enfant ainsi que deux valeurs `left` et `right` représentant le nombre d'opérations et à appliquer aux paramètres. Ainsi, si le noeud enfant est d'arité  $n$ , alors le noeud Projection parent est d'arité  $n + \text{left} + \text{right}$ . Représenter les jetons et par un seul noeud permet d'accélérer l'interprétation par rapport à deux noeuds Left et Right distincts. En effet, dans le deuxième cas, pour chaque jeton et , on ajoutera un appel dans la pile d'exécution tandis que dans le cas du noeud Projection, un seul appel devra être empiler. Ce choix se justifie également par la volonté de mettre en cache les noeuds de l'arbre syntaxique pour accélérer davantage l'interprétation. Or ce choix permet également de réduire l'espace mémoire nécessaire pour représenter les jetons et . Le jeton est représenté par le noeud Composition qui contient un noeud enfant  $f$  et  $n$  noeuds nommés

*compounds*, où  $n$  est l'arité du noeud  $f$ . L'arité du noeud Composition parent est donc la même que l'arité de chacune composantes. Pour finir, le jeton **R** est représenté par le noeud Recursion qui contient un noeud enfant  $f$ . Si  $f$  est d'arité  $n$  alors le noeud Recursion parent est d'arité  $n - 1$ .

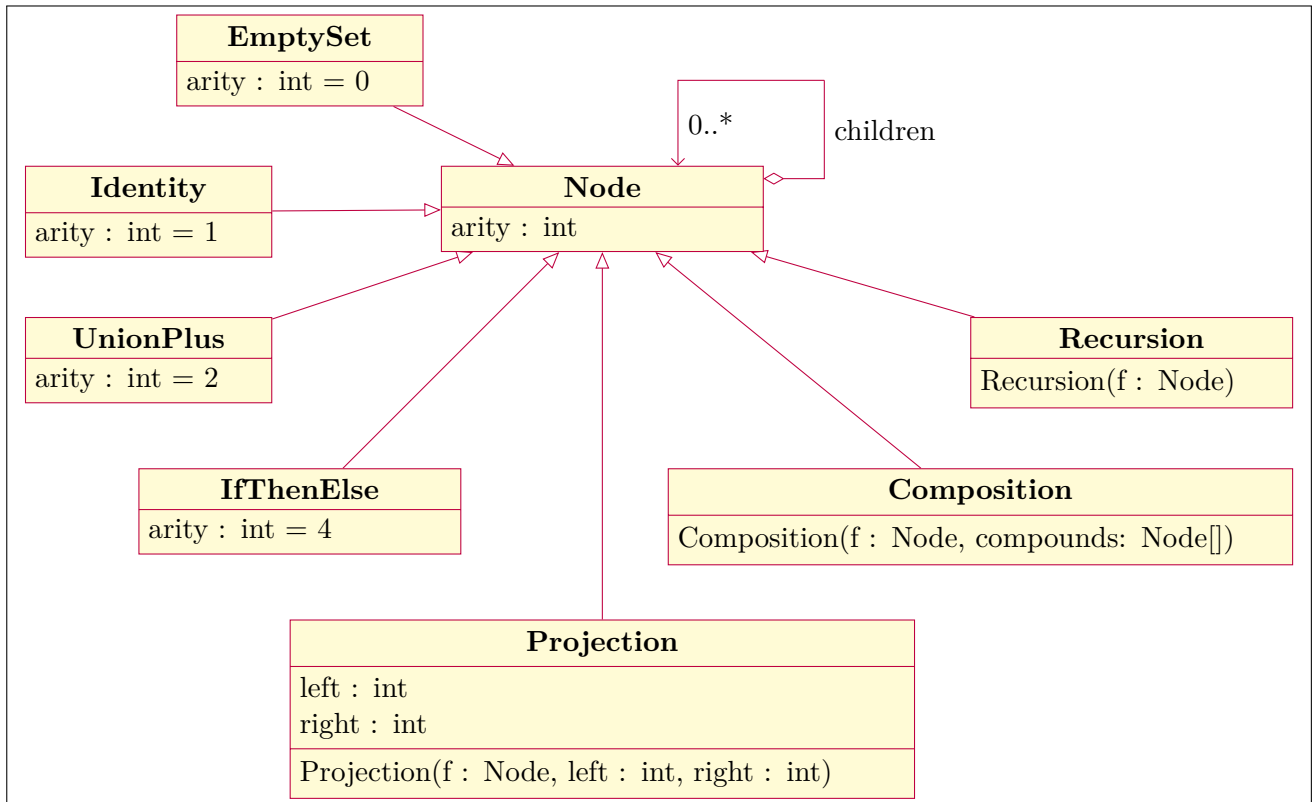


Figure 2: Représentation UML des noeuds

Remarquons qu'à partir de l'arbre syntaxique généré, on peut déjà vérifier la validité du programme en contrôlant les arités de chaque noeud de l'arbre.

### 3.2.6 Interprétation de l'arbre syntaxique

Pour pouvoir interpréter un programme, nous allons devoir parcourir l'arbre syntaxique qui lui est associé. Pour cela, nous allons mettre en place un patron de conception visiteur.

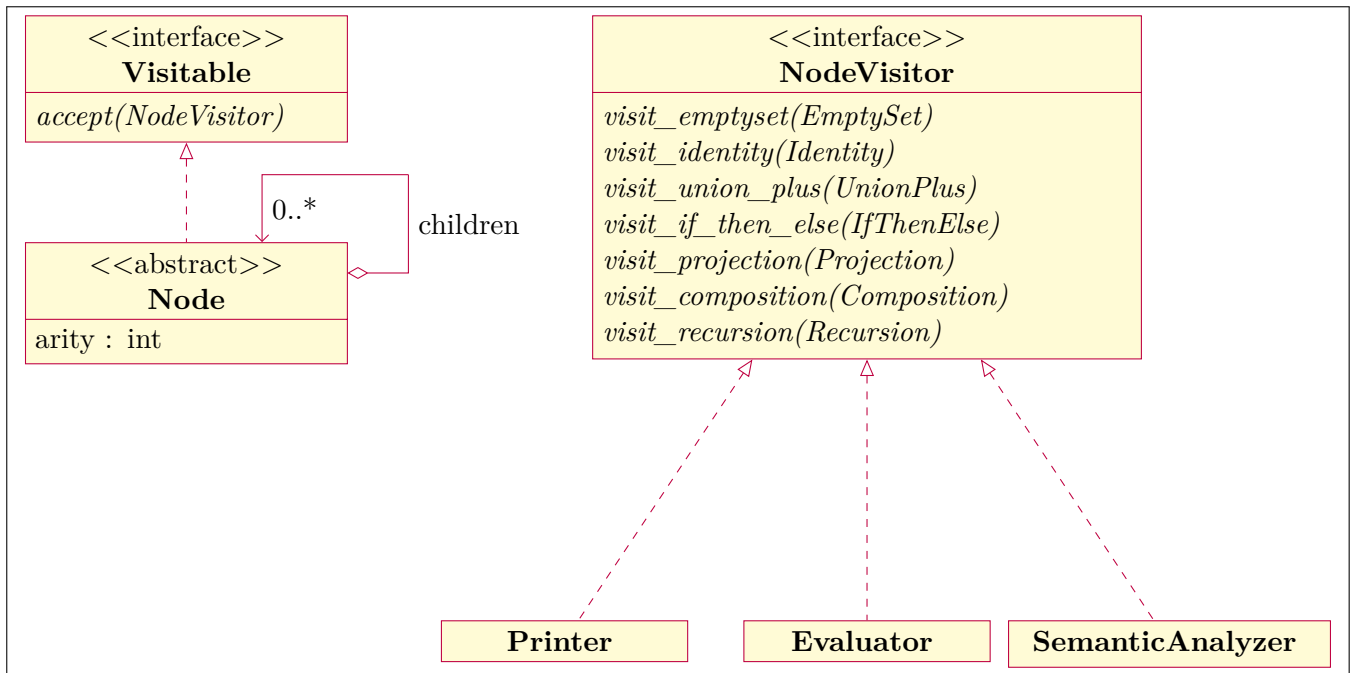


Figure 3: UML patron de conception visiteur

Il est important de remarquer que ce patron de conception nous permet également de pouvoir représenter l'arbre sous diverses formes ou bien d'effectuer son analyse sémantique sans avoir à modifier la base de code actuelle.

Pour pouvoir évaluer chaque noeud, nous avons ajouté la notion d'expression qui permet d'ajouter l'appel d'un noeud avec ses arguments à la pile d'exécution.

On a choisi d'utiliser des expressions paresseuses (voir figure 4) par souci de performances. Ainsi, on évalue une expression seulement si le résultat est nécessaire pour continuer l'exécution du programme. Les expressions paresseuses sont mutables et sont closes lorsqu'elles sont évaluées. Ainsi, le résultat de l'évaluation d'une expression est propagé dans toutes les expressions qui la contenaient. Lors de l'appel initial de l'interpréteur, celui-ci crée des expressions qui sont forcement closes puisqu'elles contiennent les valeurs des paramètres fournis par l'utilisateur.

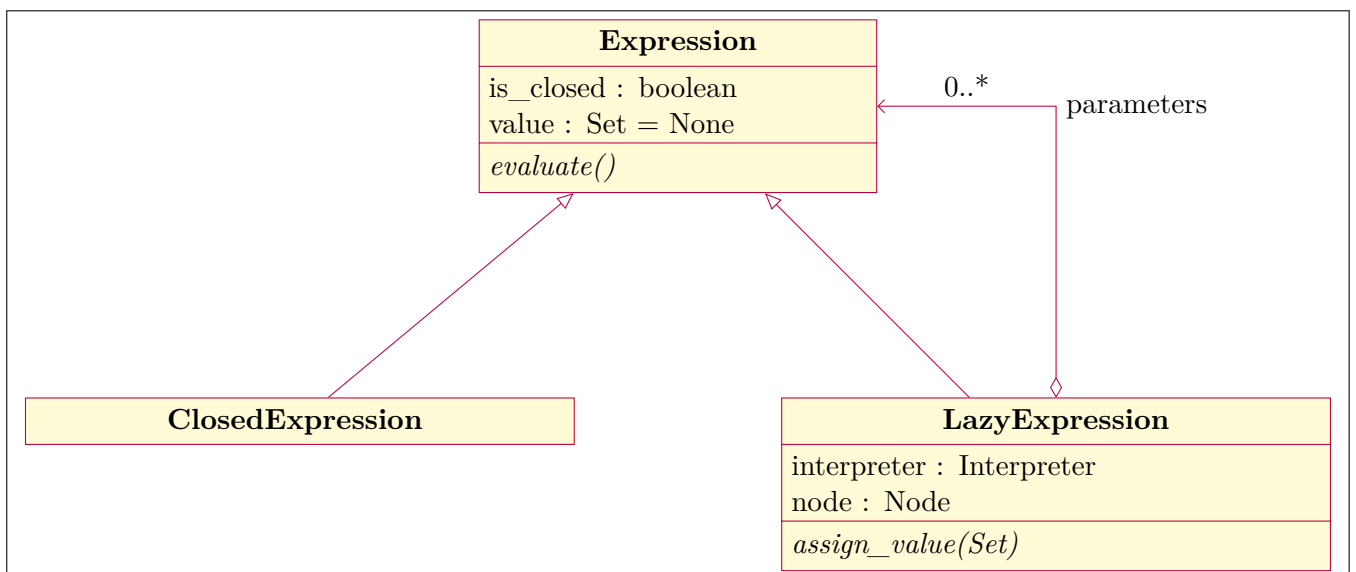


Figure 4: Représentation UML des expressions

Lorsqu'une expression doit être évaluée, l'interpreteur fait appel au visiteur Evaluator pour évaluer le résultat d'un noeud sur les paramètres de l'expression parente.

Par exemple, prenons l'expression E qui contient le noeud Identity noté I et une expression en paramètre P. Pour évaluer E, nous avons deux possibilités:

- soit P est close et dans ce cas, on peut clore E en lui assignant la valeur de P;
- soit P n'est pas close et dans ce cas, on l'ajoute à la pile d'exécution. P sera alors évaluée jusqu'à être close, puis l'expression E sera de nouveau évaluée et la valeur de P lui sera assignée.

Il en est de même pour les noeuds EmptySet, UnionPlus, IfThenElse et Projection. Le noeud Composition est un peu particulier car il ajoute une nouvelle expression sur la pile qui possède le noeud f et n nouvelles expressions en paramètres correspondants au n composantes.

Enfin, le noeud Recursion nécessite d'ajouter un noeud qui permet d'effectuer l'opération  $\bigcup_{u \in x} f(u)$  uniquement si l'évaluation de cette union est nécessaire. Pour cela, nous avons besoin d'un noeud noté Union qui évaluera chacun des  $f(u)$  et un autre noeud noté Merge qui construira l'union des  $f(u)$ .

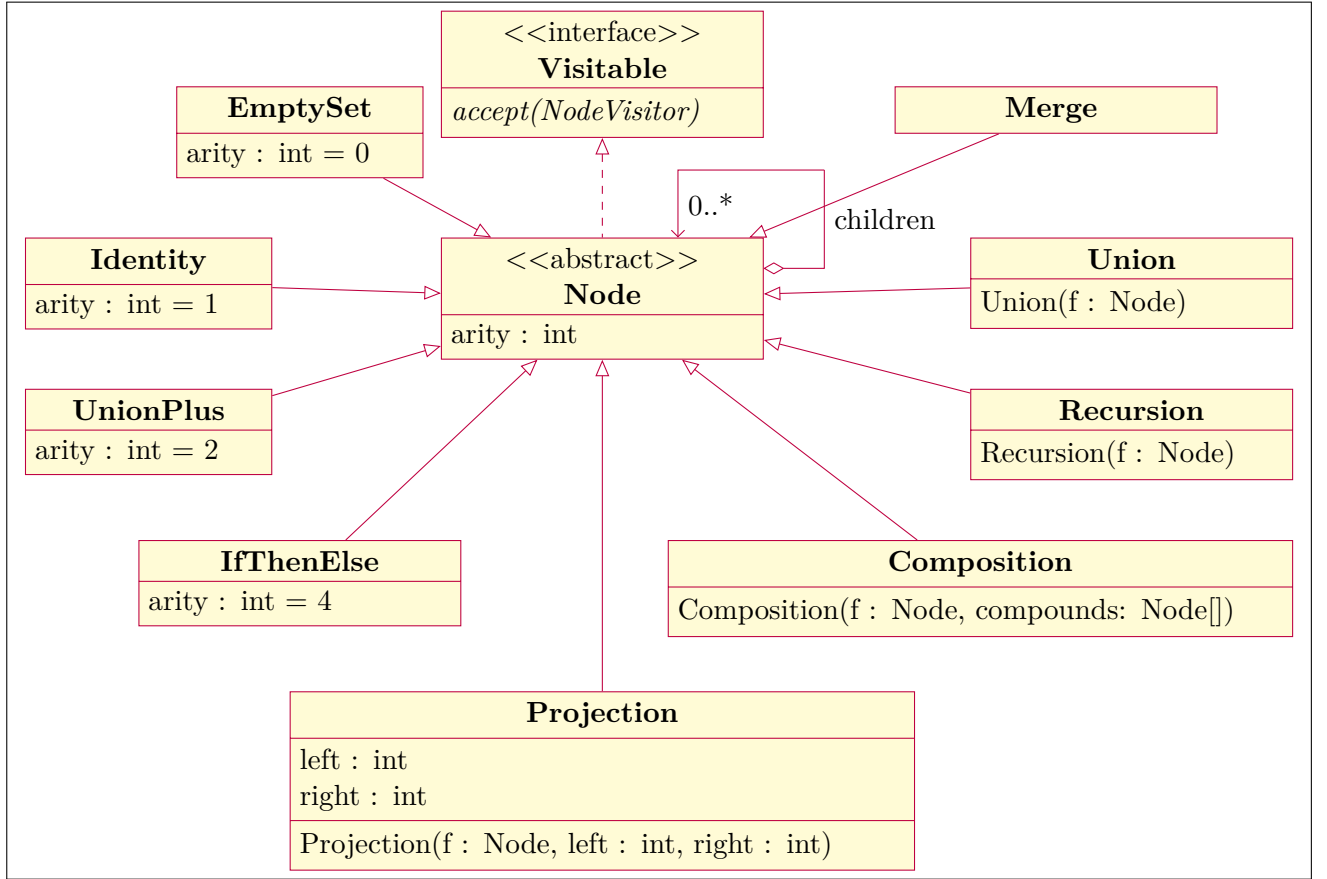


Figure 5: Représentation UML des noeuds avec Union et Merge

Plus tôt, dans la partie 3.2.5, nous avons expliqué la mise en cache des noeuds mais cette explication était incomplète. En effet, la mise en cache des noeuds devient très intéressante pour l'évaluateur, puisque lorsqu'une expression sera évaluée, on met en cache son résultat pour accélérer le calcul des autres expressions (par exemple dans le cas d'un calcul de la fonction de Fibonacci). Cette mise en cache nécessite de comparer les expressions deux à deux et cette comparaison nécessite de comparer les noeuds. Sans mise en cache, comparer deux noeuds peut être long car nous devons comparer tous les descendants de chaque noeud, or avec la mise en cache, il suffit de comparer les adresses mémoires pour comparer les noeuds.

Pour finir, l'interpréteur utilise des ensembles immuable comme élément du modèle. L'immuabilité permet de factoriser la taille en mémoire des grands ensembles, en particulier pour les ordinaux. Par exemple, l'ordinal  $n$  contient  $n$  ordinaux de 0 à  $n - 1$ . Au total, l'ordinal  $n$ , représenté sous forme d'arbre, contient  $2^n$  sommets, mais seulement  $n + 1$  éléments distincts. On peut donc gagner beaucoup de mémoire en réutilisant les ensembles précédemment construits.

### 3.2.7 Fonctions

Pour compléter l'interpréteur, nous avons ajouté un type de noeuds qui contient son propre comportement (au lieu que celui-ci soit inclus dans l'évaluateur). Cela nous permet de définir des fonctions qui remplacent certains programmes.

**Exemple** Le programme  $\text{R} \blacktriangleright \text{I}$  est le programme  $(x) \mapsto 0$ . Ce programme n'est pas très intéressant, d'autant plus que le programme  $\blacktriangleright \emptyset$  est équivalent et plus court. Sauf que le programme  $\text{R} \blacktriangleright \text{I}$  nécessite  $n$  étapes de calculs où  $n$  est le rang de  $x$ . On se trouve donc face à un programme coûteux à exécuter pour finalement aboutir à un résultat simple à calculer. D'autres exemples existent comme  $\text{R} \text{R} \in$  qui est encore plus long à évaluer et qui calcul  $(x, y) \mapsto 0$ .

Pour ce type de programme, les fonctions permettent de précompiler leurs comportements.

### 3.3 Générateur

Le générateur de programmes a pour but de nous aider à identifier et déterminer le comportement des programmes de petite taille. Le travail d'analyse des programmes devra être manuel, mais il permettra de fortement améliorer les performances de l'interpréteur en précompilant le comportement des petits programmes.

Le générateur est une fonction  $G$  à 2 arguments: la taille  $n$  du programme voulu et son arité  $a$ . Par conséquent,  $G(n, a)$  renvoie l'ensemble des programmes de taille  $n$  et d'arité  $a$ .

Pour l'implémentation, nous avons construit une fonction  $G$  qui prend en argument 3 paramètres: la taille  $n$ , l'arité  $a$  et un troisième paramètre permettant de ne pas inclure de jetons  $\blacktriangleleft$  ou  $\blacktriangleright$  au résultat. Ce choix vient du fait que l'on veut minimiser la génération de programme existant sous une forme plus courte. Dans le cas de la composition, si on a  $\text{W} \blacktriangleleft \text{f } g_1 \dots g_n$  alors on peut construire un programme plus court équivalent:  $\text{W} \text{f } g_2 \dots g_n$ . De même avec  $\text{W} \blacktriangleright \text{f } g_1 \dots g_n$ , on aurait  $\text{W} \text{f } g_1 \dots g_{n-1}$  qui serait plus court et équivalent. On élimine également le cas inverse dans lequel toutes les composantes commencent par  $\blacktriangleleft$  ou  $\blacktriangleright$  car  $\text{W} \text{f } \blacktriangleright g_1 \dots \blacktriangleright g_n = \blacktriangleright \text{W} \text{f } g_1 \dots g_n$  qui est plus court dès que  $n > 1$ . Pour la récursion, on a  $\text{R} \blacktriangleleft \text{p}$  qui est équivalent à  $\text{p}$  et, dans le cas où  $\text{p}$  est d'arité  $n > 1$ , on a  $\blacktriangleright \text{R} \text{p}$  qui équivaut à  $\text{R} \blacktriangleright \text{p}$ .

### 3.4 Benchmark

Lors de l'énumération des programmes, nous avons découvert des programmes de même taille calculant la même fonction. Un des points que nous souhaitions analyser est le nombre d'étape de calcul de chaque programme sur certaines entrées. Pour ce faire, nous avons développé un outil de benchmark qui compte le nombre d'étape de calcul atomiques: les fonctions initiales correspondant aux jetons  $\emptyset$   $\cup$   $\in$ . Les résultats sont sauvegardés sous forme de tableau puis le coefficient directeur de la courbe associé est calculé.

Le développement du benchmark a nécessité de revoir l'implémentation de la mise en cache des expressions. La mise en cache des expressions fait partie intégrante de l'interpréteur et le nombre d'étapes sans mise en cache devient tellement élevé que les mesures sur certains programmes deviennent impossibles. Nous devons donc utiliser la mise en cache des expressions durant les mesures mais tout en réinitialisant le cache entre chaque calcul pour éviter de fausser les données. Nous avons alors deux choix possibles:

- forcer une réinitialisation du cache entre chaque calcul lors des mesures sans modifier le comportement actuel de la mise en cache des expressions.
- contextualiser la mise en cache des expressions à chaque nouveau calcul. C'est-à-dire que chaque calcul dispose de son propre cache.

La première solution utilise plus de mémoire vive car tous les calculs sont conservés mais lorsque l'on effectue deux fois le même calcul, la seconde fois ne nécessite qu'un appel au cache. La seconde



solution utilise un cache pour les expressions uniquement durant le calcul de celle-ci. L'usage de la mémoire vive est donc grandement diminué et on a l'avantage/inconvénient qu'effectuer deux fois le même calcul nécessitera le même nombre d'étape de calcul. Ce deuxième point est nuancé car on effectue rarement deux fois le même calcul dans la pratique. De plus, ne pas avoir de cache partagé est justement nécessaire pour effectuer des mesures pertinentes pour notre benchmark.

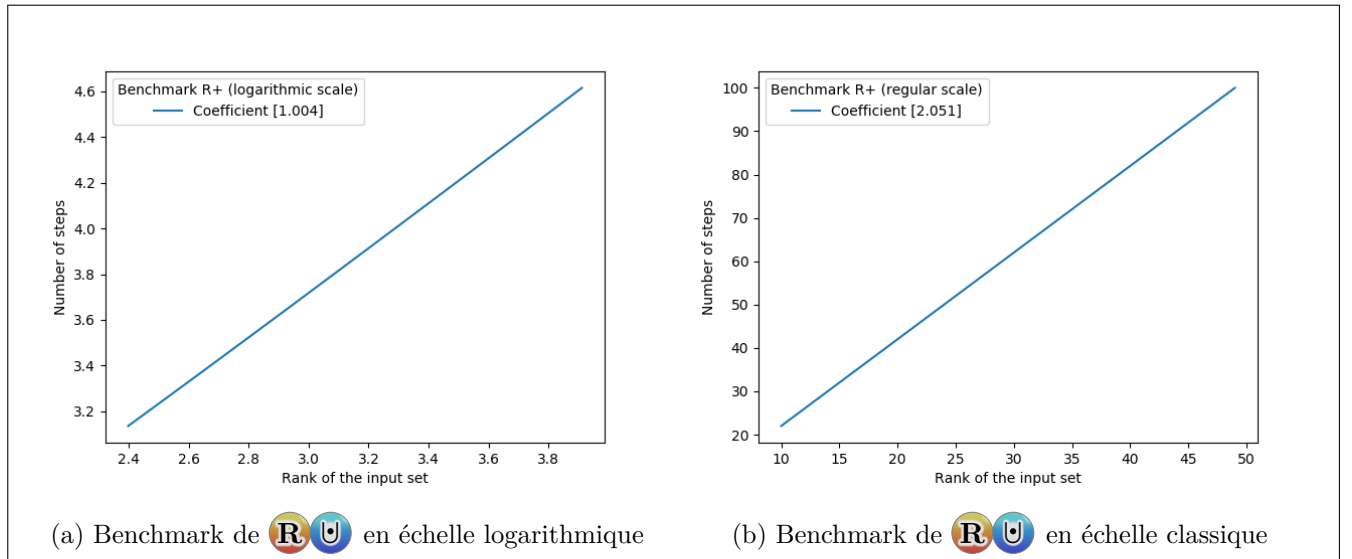
Pour finir, dans le premier cas, il faut garder une trace des calculs qui auraient pu être interrompus par l'utilisateur. Ces calculs, à cause de l'intervention de l'utilisateur, restent sans résultats mais sont pourtant conservés en mémoire. Ils font donc garder une trace des ajouts dans le cache lors d'un calcul pour les supprimer en cas de problème. Ce problème couplé à l'importante charge que subit la mémoire vive pose de gros problèmes de performance en cas d'interruption de l'interpréteur.



Pour toutes ces raisons, nous avons décidé de modifier le comportement actuel de la mise en cache des expressions en optant pour la deuxième solution: contextualiser la mise en cache des expressions à chaque nouveau calcul.

Remarquons que les autres caches (celui des ensembles et celui des noeuds) ne nécessitent aucune modification. Le benchmark compte le nombre d'appels aux jetons atomiques, ces deux autres caches n'ont donc aucun impact sur le résultat.

### 3.4.1 Exemples

**clôture transitive** On s'intéresse ici à l'étude du programme  . Le nombre d'étapes étant linéaire dans ce cas, nous affichons également un graphique sans échelle logarithmique.



À partir de la figure 6a, on déduit que le programme   est linéaire par rapport au rang de l'entrée. Pour l'exemple, on décide de tracer la courbe en échelle classique pour visualiser le coefficient de la fonction affine obtenue en figure 6b.

On peut illustrer l'impact du cache en affichant la courbe obtenue lorsque la mise en cache des expressions est désactivée.

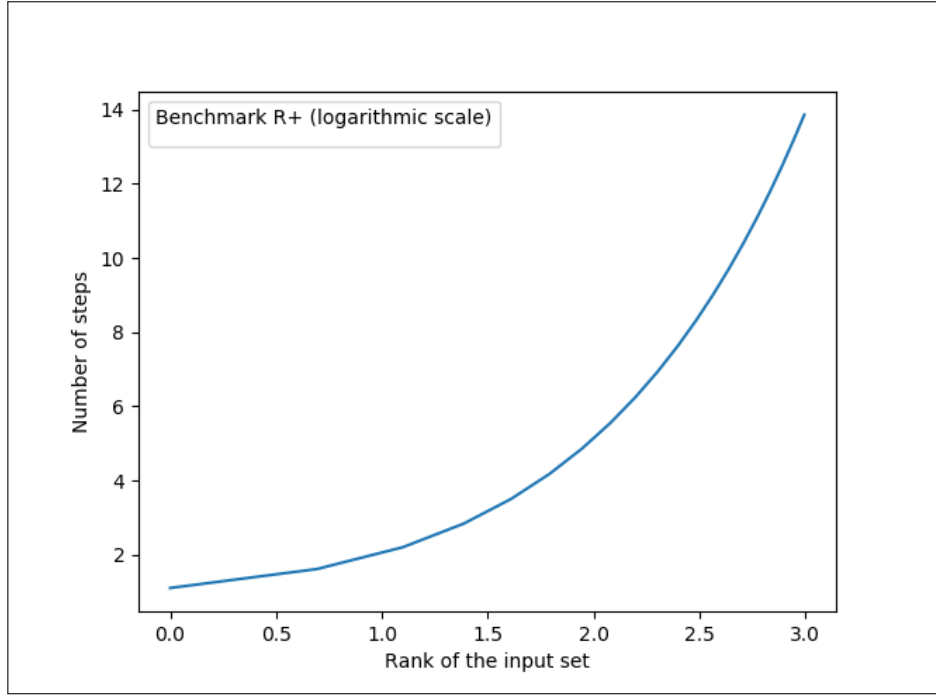



Figure 7: Benchmark de  en échelle logarithmique sans mise en cache

Sans la mise en cache, il devient difficile d'effectuer des mesures, même pour un programme aussi simple que la clôture transitive. Le nombre d'étapes nécessaire est exponentiel, et remarquons qu'avec le cache, nous pouvions faire des mesures sur des ensembles de rang 50 en moins de 100 étapes tandis que sans la mise en cache, la clôture transitive nécessite plus d'un million d'étapes de calcul dès le rang 20.

**Multiplication** On s'intéresse ici à l'étude du programme .

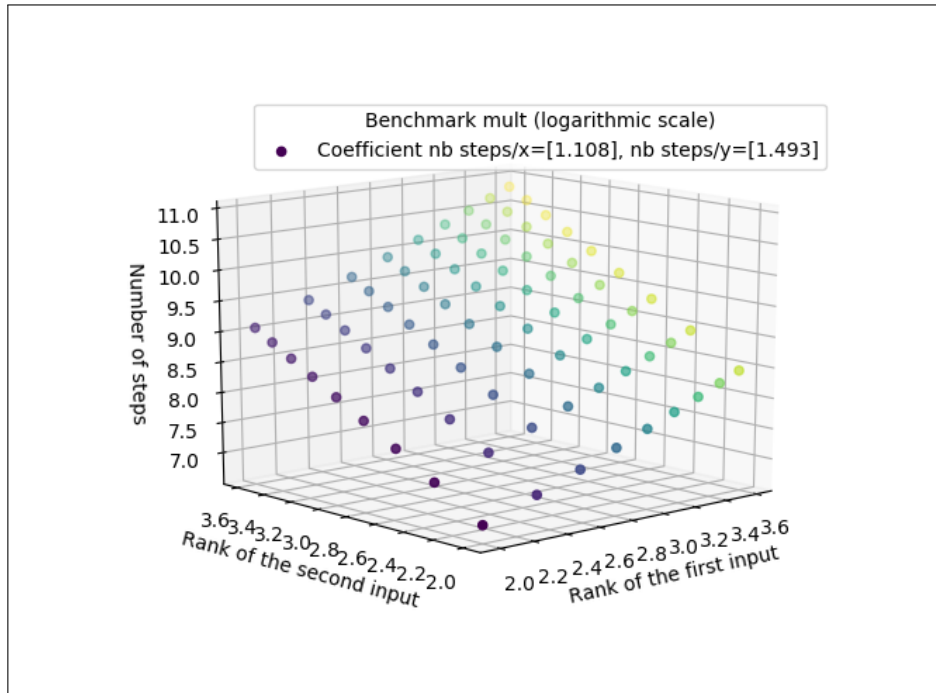


Figure 8: Benchmark de  en échelle logarithmique

La multiplication est un programme d'arité 2, en conséquence, on s'intéresse à l'évolution du nombre d'étapes de calcul en fonction de chacun des paramètres. Dans cet exemple, le coefficient du nombre d'étapes en fonction de  $x$  (le premier paramètre) est de 1.108 tandis qu'il est de 1.493 par rapport à  $y$  (le deuxième paramètre).

### 3.4.2 Outil de comparaison

Pour faciliter la visualisation des données, on propose de réunir les résultats de plusieurs programmes sur la même figure.

On s'intéresse ici à la fonction successeur sur les ordinaux  $\mathbf{S} = \textcircled{\omega} \textcircled{\cup} \textcircled{\mathbf{I}} \textcircled{\mathbf{I}}$ . On verra dans la section 3.2 que la clôture transitive  $\textcircled{\mathbf{R}} \textcircled{\cup}$  se comporte comme la fonction successeur  $\mathbf{S}$ . On cherche donc à comparer la complexité de ces deux programmes pour déterminer le plus performant.

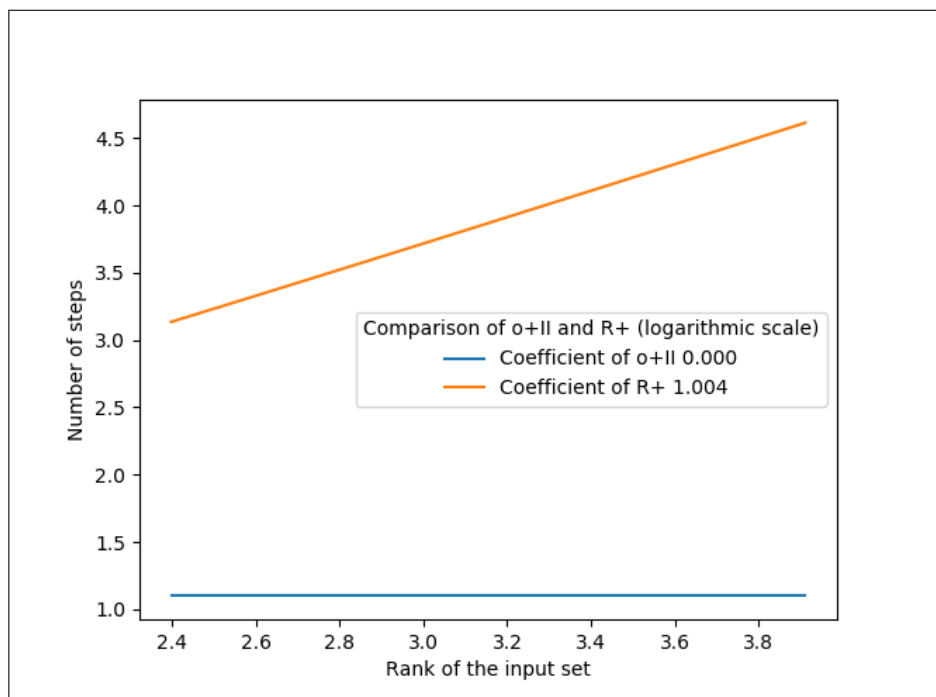


Figure 9: Comparatif de  $\textcircled{\omega} \textcircled{\cup} \textcircled{\mathbf{I}} \textcircled{\mathbf{I}}$  et  $\textcircled{\mathbf{R}} \textcircled{\cup}$  en échelle logarithmique

Comme le programme  $\textcircled{\omega} \textcircled{\cup} \textcircled{\mathbf{I}} \textcircled{\mathbf{I}}$  ne comporte aucun jeton  $\textcircled{\mathbf{R}}$ , le nombre d'étapes de calcul de ce programme est constant par rapport à l'entrée. Cela correspond au coefficient 0 de la courbe en échelle logarithmique.

En revanche, le programme  $\textcircled{\mathbf{R}} \textcircled{\cup}$  quant à lui opère une récursion sur l'entrée. Ce programme effectue donc un nombre d'étapes qui dépend de la taille de l'entrée. En l'occurrence, comme le coefficient en échelle logarithmique est de 1, le nombre d'étapes est linéairement proportionnel à la taille de l'entrée.

Cet outil nous permet ainsi de visualiser rapidement les fonctions donc la complexité est trop grande pour être analysé via les outils développés.

Un exemple de ce genre de programme est  $\textcircled{\mathbf{R}} \blacktriangleright \textcircled{\mathbf{R}} \blacktriangleright \textcircled{\mathbf{R}} \textcircled{\omega} \textcircled{\cup} \textcircled{\cup} \textcircled{\cup}$ :





**Preuve.** Soit  $\alpha$  un ensemble transitif, alors  $\bigcup_{u \in \alpha} \mathbf{R} \mathbf{U} (u) \subset \alpha$ , d'après le lemme 3.1. Réciproquement,  $\forall u \in \alpha, u \in \mathbf{R} \mathbf{U} (u)$  donc on a  $\alpha \subset \bigcup_{u \in \alpha} \mathbf{R} \mathbf{U} (u)$ . On en déduit que  $\bigcup_{u \in \alpha} \mathbf{R} \mathbf{U} (u) = \alpha$ . Ce qui nous donne:

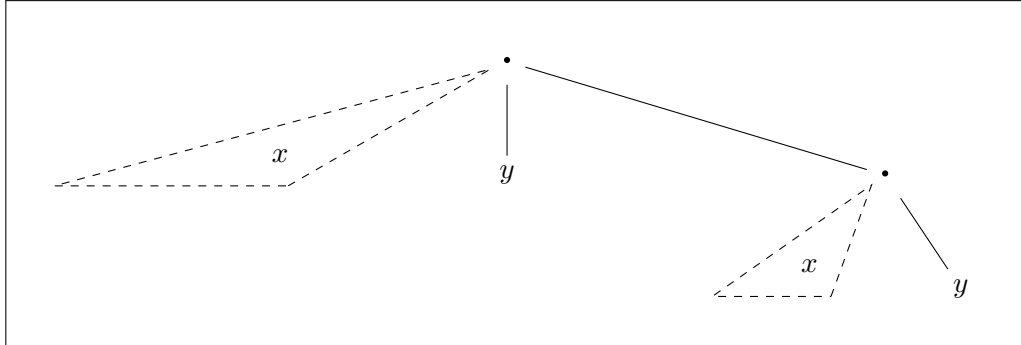
$$\begin{aligned} \mathbf{R} \mathbf{U} (\alpha) &= \bigcup_{u \in \alpha} \mathbf{R} \mathbf{U} (u) \cup \{\alpha\} \\ &= \mathbf{S} (\alpha) \end{aligned}$$

■

2.  $\mathbf{R} \mathbf{E} : (x, y, z) \mapsto \emptyset$  si  $y \in z$  sinon  $x$

### 3.5.2 Taille 4

1.  $\mathbf{O} \mathbf{U} \mathbf{O} \mathbf{O} : () \mapsto 1$
2.  $\mathbf{R} \mathbf{P} \mathbf{R} \mathbf{U} : (x) \mapsto \text{rang}(x) + 1$
3.  $\mathbf{O} \mathbf{U} \mathbf{I} \mathbf{I} = \mathbf{S} : (x) \mapsto x \cup \{x\} = S(x)$
4.  $\mathbf{O} \mathbf{U} \mathbf{U} \mathbf{U} : (x, y) \mapsto x \cup \{y, x \cup \{y\}\}$



5.  $\mathbf{O} \mathbf{R} \mathbf{U} \mathbf{U} : (x, y) \mapsto TC(\{x \cup \{y\}\})$
6.  $\mathbf{O} \mathbf{U} \mathbf{E} \mathbf{E} : (x, y, u, v) \mapsto S(x)$  si  $u \in v$  sinon  $S(y)$
7.  $\mathbf{O} \mathbf{R} \mathbf{U} \mathbf{E} : (x, y, u, v) \mapsto TC(x)$  si  $u \in v$  sinon  $TC(y)$

### 3.5.3 Taille 5

1.  $\mathbf{R} \mathbf{O} \mathbf{U} \mathbf{U} \mathbf{U} = \mathbf{R} \mathbf{O} \mathbf{R} \mathbf{U} \mathbf{U} = \mathbf{++} : (x) \mapsto TC(\{\bigcup_{u \in x} (\mathbf{++}(u)) \cup \{x\}\})$

**Preuve de  $\mathbf{R} \mathbf{O} \mathbf{U} \mathbf{U} \mathbf{U} = \mathbf{R} \mathbf{O} \mathbf{R} \mathbf{U} \mathbf{U}$  par récurrence transfinie:**  
 Pour cette preuve, nous allons également vérifier que pour tout ensemble  $\alpha$ ,

$$\mathbf{R} \mathbf{O} \mathbf{U} \mathbf{U} \mathbf{U} (\alpha) = \mathbf{R} \mathbf{O} \mathbf{R} \mathbf{U} \mathbf{U} (\alpha) \text{ est un ensemble transitif.}$$

- $\mathbf{R} \mathbf{O} \mathbf{U} \mathbf{U} \mathbf{U} (0) = \mathbf{R} \mathbf{O} \mathbf{R} \mathbf{U} \mathbf{U} (0) = 2$ , et comme 2 est un ordinal, il est transitif.



## Étude de complexité

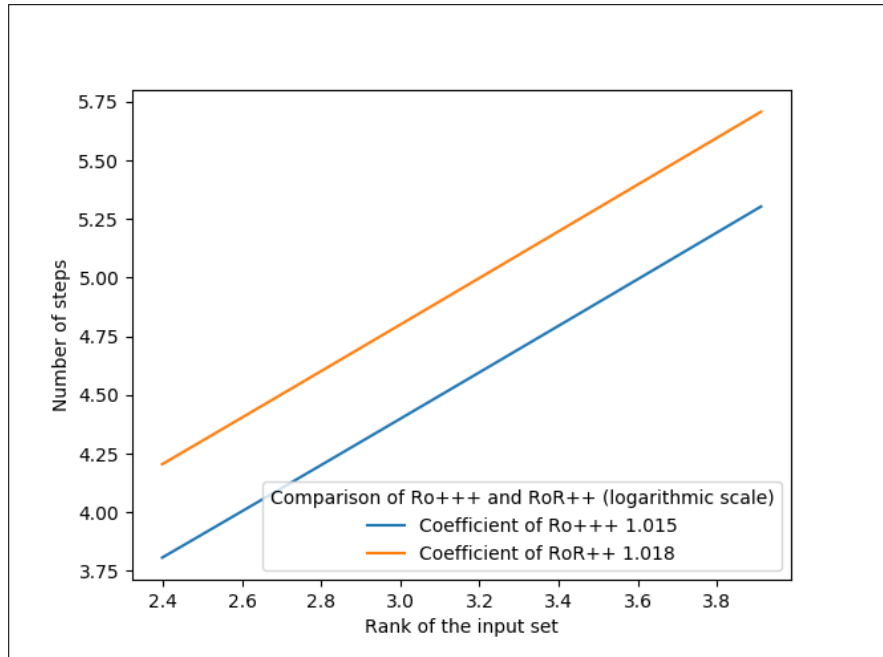


Figure 11: Comparaison de  $\mathbf{R} \otimes \mathbf{U} \mathbf{U} \mathbf{U}$  et de  $\mathbf{R} \otimes \mathbf{R} \mathbf{U} \mathbf{U}$  en échelle logarithmique

On peut observer sur ce graphique que les deux droites sont parallèles (même coefficient directeur), ce qui indique que les deux programmes sont de même complexité. Le coefficient étant de 1, cette complexité est linéaire. Cependant, l'écart entre les courbes indique qu'il existe un facteur entre le nombre d'étapes nécessaires pour chaque programme. En l'occurrence, le programme  $\mathbf{R} \otimes \mathbf{R} \mathbf{U} \mathbf{U}$  nécessite 1.5 fois plus d'étapes que le programme  $\mathbf{R} \otimes \mathbf{U} \mathbf{U} \mathbf{U}$ .

De ce fait, comme ces deux programmes calculent la même fonction, par la suite, nous utiliserons le programme  $\mathbf{R} \otimes \mathbf{U} \mathbf{U} \mathbf{U}$ . (Ce qui signifie que le programme  $\mathbf{R} \triangleright \mathbf{R} \otimes \mathbf{U} \mathbf{U} \mathbf{U}$  sera généré mais pas le programme  $\mathbf{R} \triangleright \mathbf{R} \otimes \mathbf{R} \mathbf{U} \mathbf{U}$ )

2.  $\mathbf{U} \mathbf{U} \mathbf{I} \mathbf{L} \mathbf{U} : (x) \mapsto x \cup \{0\}$
3.  $\mathbf{U} \mathbf{U} \mathbf{I} \mathbf{R} \mathbf{U} : (x) \mapsto x \cup \{TC(\{x\})\}$
4.  $\mathbf{U} \mathbf{U} \mathbf{L} \mathbf{U} \mathbf{I} = \mathbf{U} : (x) \mapsto \{x\}$
5.  $\mathbf{U} \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U} : (x) \mapsto TC(\{TC(\{x\})\}) = S(TC(\{x\}))$
6.  $\mathbf{U} \mathbf{U} \mathbf{U} \mathbf{R} \mathbf{I} : (x, y) \mapsto x \cup \{x, y\} = S(x) \cup \{y\}$
7.  $\mathbf{U} \mathbf{U} \mathbf{L} \mathbf{I} \mathbf{U} : (x, y) \mapsto y \cup \{x \cup \{y\}\}$
8.  $\mathbf{U} \mathbf{U} \mathbf{R} \mathbf{I} \mathbf{U} : (x, y) \mapsto x \cup \{x \cup \{y\}\}$
9.  $\mathbf{R} \otimes \mathbf{U} \in \in : (x, u, v) \mapsto rang(x) + 1 \text{ si } u \in v \text{ sinon } S(x)$
10.  $\mathbf{R} \otimes \mathbf{R} \mathbf{U} \in : (x, u, v) \mapsto rang(x) + 1 \text{ si } u \in v \text{ sinon } TC(x)$
11.  $\mathbf{U} \mathbf{R} \mathbf{U} \mathbf{R} \in : (x, u, v) \mapsto 1 \text{ si } u \in v \text{ sinon } TC(x)$

### 3.5.4 Taille 6

$$1. \textcircled{R} \blacktriangleright \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} = \textcircled{R} \blacktriangleright \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} : (x) \mapsto \text{rang}(x) + 1$$

*Preuve de  $\textcircled{R} \blacktriangleright \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} = \textcircled{R} \blacktriangleright \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} = \text{rang} + 1$  par récurrence transfinie:*

- $\textcircled{R} \blacktriangleright \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} (0) = \textcircled{R} \blacktriangleright \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} (0) = \text{rang}(0) + 1 = 1$
- Soit  $\alpha$  un ensemble, on suppose que pour tout  $u \in \alpha$ , on a:

$$\textcircled{R} \blacktriangleright \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} (u) = \textcircled{R} \blacktriangleright \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} (u) = \text{rang}(u) + 1$$

Alors

$$\begin{aligned} \textcircled{R} \blacktriangleright \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} (\alpha) &= \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} \left( \bigcup_{u \in \alpha} \textcircled{R} \blacktriangleright \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} (u) \right) \\ &= \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} \left( \bigcup_{u \in \alpha} \textcircled{R} \blacktriangleright \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} (u) \right) \\ &= \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} \left( \bigcup_{u \in \alpha} \text{rang}(u) + 1 \right) \quad \text{Par hypothèse de récurrence} \\ &= \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} (\text{rang}(\alpha)) \\ &= \text{rang}(\text{rang}(\alpha)) + 1 \quad (\text{car } \textcircled{R} \blacktriangleright \textcircled{R} \textcircled{\cup} = \text{rang} + 1) \\ &= \text{rang}(\alpha) + 1 \end{aligned}$$

$$\begin{aligned} \textcircled{R} \blacktriangleright \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} (\alpha) &= \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} \left( \bigcup_{u \in \alpha} \textcircled{R} \blacktriangleright \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} (u) \right) \\ &= \textcircled{S} \left( \bigcup_{u \in \alpha} \textcircled{R} \blacktriangleright \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} (u) \right) \quad (\text{car } \textcircled{\omega} \textcircled{\cup} \textcircled{I} \textcircled{I} = \textcircled{S}) \\ &= \textcircled{S} \left( \bigcup_{u \in \alpha} \text{rang}(u) + 1 \right) \quad \text{Par hypothèse de récurrence} \\ &= \textcircled{S} (\text{rang}(\alpha)) \\ &= \text{rang}(\alpha) + 1 \end{aligned}$$

■

## Étude de complexité

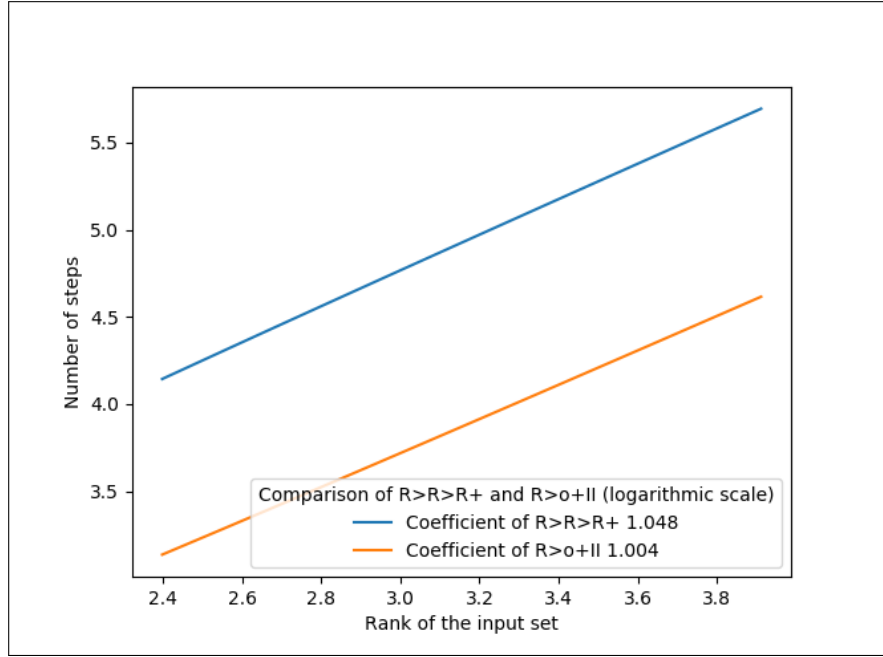


Figure 12: Comparaison de  $\mathbf{R} \triangleright \mathbf{R} \triangleright \mathbf{R} \cup$  et de  $\mathbf{R} \triangleright \omega \cup \mathbf{I} \mathbf{I}$  en échelle logarithmique

Les programmes ont tous deux une complexité linéaire par rapport à l'entrée mais le programme  $\mathbf{R} \triangleright \omega \cup \mathbf{I} \mathbf{I}$  nécessite 2 fois moins d'étapes que le programme  $\mathbf{R} \triangleright \mathbf{R} \triangleright \mathbf{R} \cup$ .

On choisit donc de conserver le programme  $\mathbf{R} \triangleright \omega \cup \mathbf{I} \mathbf{I}$  et d'ignorer le programme  $\mathbf{R} \triangleright \mathbf{R} \triangleright \mathbf{R} \cup$  pour la génération de plus grands programmes.

2.  $\omega \cup \mathbf{I} \mathbf{I} \mathbf{R} \cup$  :  $(x) \mapsto \{TC(\{x\})\}$
3.  $\omega \cup \mathbf{I} \mathbf{I} \mathbf{R} \triangleright \mathbf{I} \mathbf{I}$  :  $(x) \mapsto (x, y, z) \mapsto y \cup \{z\} \cup \{x \cup \{y\}\}$
4.  $\mathbf{R} \omega \mathbf{R} \cup \mathbf{R} \in$  :  $(x, y) \mapsto rang(x') + 1$ , où  $x'$  est l'ensemble  $x$  dans lequel tous les éléments appartenant à  $y$  sont remplacés par 0

### 3.5.5 Taille 7

1.  $\mathbf{R} \triangleright \mathbf{R} \omega \cup \cup \cup$  :  $(x) \mapsto 2 \times rang(x) + 2$   
 $= \mathbf{R} \triangleright \omega \mathbf{R} \cup \mathbf{R} \cup$

*Preuve de 1 par récurrence transfinie:*

•

$$\begin{aligned}
 2 \times rang(0) + 2 &= 2 \\
 &= \mathbf{R} \triangleright \mathbf{R} \omega \cup \cup \cup (0) \\
 &= \mathbf{R} \triangleright \omega \mathbf{R} \cup \mathbf{R} \cup (0)
 \end{aligned}$$

- Soit  $\alpha$  un ensemble, on suppose que pour tout  $u \in \alpha$ , on a:

$$\begin{aligned} 2 \times \text{rang}(u) + 2 &= \mathbf{R} \blacktriangleright \mathbf{R} \omega \mathbf{U} \mathbf{U} \mathbf{U} (u) \\ &= \mathbf{R} \blacktriangleright \omega \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U} (u) \end{aligned}$$

Alors

$$\begin{aligned} \mathbf{R} \blacktriangleright \mathbf{R} \omega \mathbf{U} \mathbf{U} \mathbf{U} (\alpha) &= \mathbf{R} \omega \mathbf{U} \mathbf{U} \mathbf{U} \left( \bigcup_{u \in \alpha} \mathbf{R} \blacktriangleright \mathbf{R} \omega \mathbf{U} \mathbf{U} \mathbf{U} (u) \right) \\ &= \mathbf{R} \omega \mathbf{U} \mathbf{U} \mathbf{U} \left( \bigcup_{u \in \alpha} 2 \times \text{rang}(u) + 2 \right) \quad \text{Par H.R} \\ &= \mathbf{R} \omega \mathbf{U} \mathbf{U} \mathbf{U} (2 \times \text{rang}(\alpha)) \\ &= 2 \times \text{rang}(\alpha) + 2 \end{aligned}$$

$$\begin{aligned} \mathbf{R} \blacktriangleright \omega \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U} (\alpha) &= \omega \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U} \left( \bigcup_{u \in \alpha} \mathbf{R} \blacktriangleright \omega \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U} (u) \right) \\ &= \omega \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U} \left( \bigcup_{u \in \alpha} 2 \times \text{rang}(u) + 2 \right) \quad \text{Par H.R} \\ &= \omega \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U} (2 \times \text{rang}(\alpha)) \\ &= \omega \mathbf{S} \mathbf{S} (2 \times \text{rang}(\alpha)) \quad (\text{Voir lemme 3.2 : } \mathbf{R} \mathbf{U} = \mathbf{S}) \\ &= 2 \times \text{rang}(\alpha) + 2 \end{aligned}$$

■

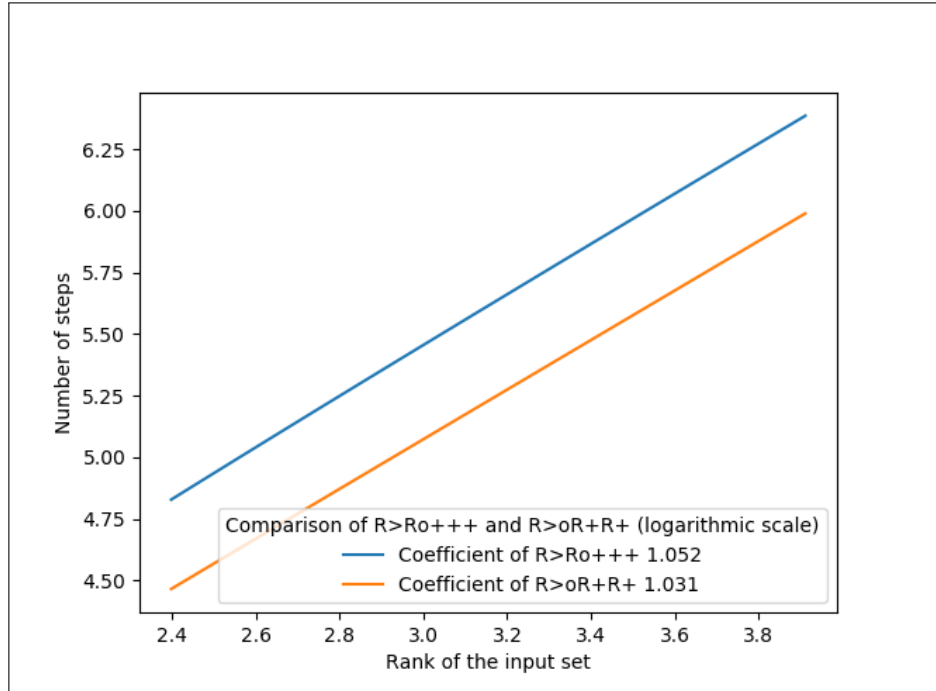























Figure 13: Comparaison de  $\mathbf{R} \blacktriangleright \mathbf{R} \omega \mathbf{U} \mathbf{U} \mathbf{U}$  et de  $\mathbf{R} \blacktriangleright \omega \mathbf{R} \mathbf{U} \mathbf{R} \mathbf{U}$  en échelle logarithmique

Les deux programmes ont la même complexité mais le programme        nécessite 1.5 fois plus d'étapes que le programme       .

On choisit donc de conserver le programme       et d'ignorer le programme     pour la génération de plus grands programmes.

2.        :  $(x) \mapsto x \cup \{rang(x) + 1\}$
3.        :  $(x) \mapsto x \cup \{S(x)\}$
4.         :  $(x) \mapsto (rang(x) + 1) \cup \{x\}$
5.        :  $(x) \mapsto TC\{S(x)\}$
6.         :  $(x) \mapsto 0 \text{ si } 0 \in x \text{ sinon } x$
7.        :  $(x) \mapsto x \cup \{x \cup \{x\}\}$

### 3.5.6 Taille 8


1.  :  $(x) \mapsto P(rang(x))$  avec  $P$  un programme de taille  $n > 8$
2.  :  $(x) \mapsto Q(rang(x))$  avec  $Q$  un programme de taille  $n > 8$

Remarque: Lors de nos recherches, nous n'avons pas découvert les programmes  $P$  et  $Q$  définis précédemment.


### 3.5.7 Taille 9


1. :  $(x) \mapsto 0$  si  $x = y$  sinon  $x + 1$
2. :  $(x) \mapsto x'$  où  $x'$  est l'ensemble  $x$  dans lequel tous les éléments appartenant à  $y$  sont remplacés par 0

### 3.5.8 Taille 11

1. 
- |     |                                |                                 |                |
|-----|--------------------------------|---------------------------------|----------------|
|     | si $x \cup \{y\} \in z$        | alors                           | $x \cup \{y\}$ |
| $:$ | $(x, y, z) \mapsto$            | sinon si $y \in z$              | alors 0        |
|     | sinon                          |                                 | $x$            |
|     |                                | si $y < x < z$                  | alors $x$      |
| $:$ | $(x, y, z)$ ordinaux $\mapsto$ | sinon si $x = y$ et $x + 1 < z$ | alors $x + 1$  |
|     |                                | sinon si $y < z$                | alors 0        |
|     |                                | sinon                           | $x$            |

### 3.5.9 Taille 12



- 

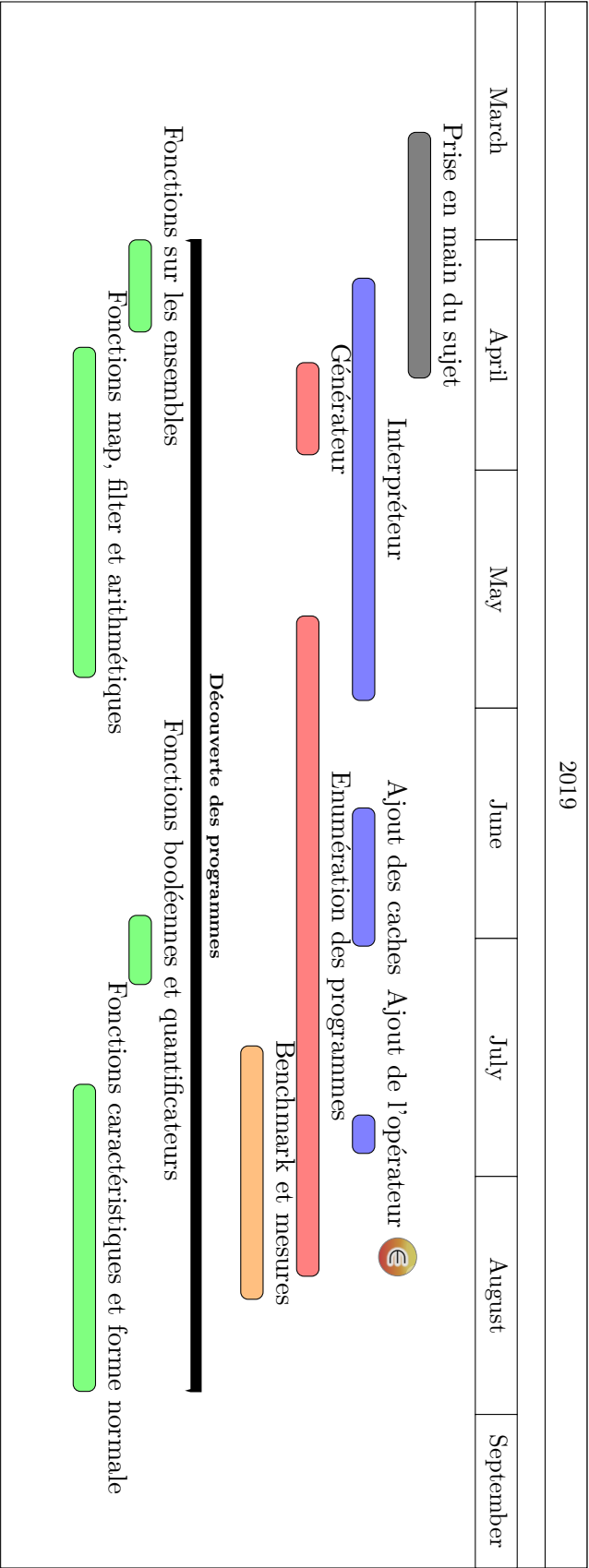
$$\begin{array}{ll}
 : (x, y) \text{ ordinaux} \mapsto & \begin{array}{ll} \text{si } x + 1 < y & \text{alors } x + 1 \\ \text{sinon si } x + 1 = y & \text{alors } 0 \\ \text{sinon} & y - 1 \end{array}
 \end{array}$$
- 

$$\begin{array}{ll}
 : (x, y) \text{ ordinaux} \mapsto & \begin{array}{ll} \text{si } x + 1 < y & \text{alors } x + 1 \\ \text{sinon} & x \end{array}
 \end{array}$$



3.5.10 Taille 13

1.  :  $(x) \text{ ordinal} \mapsto \bigcup_{u \in x} (u + 1 \text{ si } u + 1 < x) = x - 1$   
= 



## 5 Bilan

Pour conclure, nous avons découvert toutes les fonctions arithmétiques, ainsi qu'un grand nombre de fonctions caractéristiques. Nous avons également pu exhiber un certain nombre de fonctions permettant de manipuler les ensembles et les structures de données, comme les couples, que l'on peut construire par dessus. Toutes ces fonctions nous ont permis de construire la forme normale de Cantor qui est un premier pas vers l'ajout de jeton comme  $\omega$  à notre modèle.

L'énumération des programmes est un travail laborieux mais très intéressant: elle donne un bon aperçu des fonctions simples et naturelles pour notre modèle, même si celles-ci sont difficiles à exprimer via les outils mathématiques actuels à notre disposition.

Il reste encore beaucoup de travail, que ce soit sur la découverte des programmes avec la découverte du programme vérifiant la finitude d'un ensemble, l'énumération des programmes ou l'ajout de jetons supplémentaires à notre modèle. Même dans le cas le plus simple où uniquement  $\omega$  est ajouté au modèle, de nombreuses questions sont en suspens, par exemple, la possibilité de simuler une boucle `while` avec la récursion transfinie sur  $\omega$ . Nous pouvons également étudier l'impact sur notre modèle si on enlève ou si on modifie certains jetons. Il existe des modèles de calcul rudimentaire plus faible que les fonctions récursives primitives, mais peut-on construire des modèles de calculs intermédiaires, et quelles propriétés auront-ils ?

## References

- [1] Keith J. Devlin. Thomas jech. set theory. pure and applied mathematics. academic press, new york, san francisco, and london, 1978, xi 621 pp. *Journal of Symbolic Logic*, 46(4), 1981.
- [2] Ronald B Jensen and Carol Karp. Primitive recursive set functions. In *Axiomatic set theory*, volume 13, pages 143–176. Amer. Math. Soc, 1971.
- [3] Casimir Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta mathematicae*, 2(1):161–171, 1921.

## Résumé

Dans ce rapport de stage, nous allons vous présenter l'étude pratique et théorique de l'ensemble des programmes rékursifs primitifs sur les ensembles purs grâce à un modèle de calcul représentant les fonctions rékursives primitives sous forme de suites de jetons. Pour ce faire, nous avons développé et optimisé un interpréteur ainsi qu'un générateur de programme. Ces outils nous ont permis d'exhiber des programmes usuels comme les fonctions arithmétiques ou certaines fonctions caractéristiques.

Nous avons également énuméré et analysé tous les programmes de petite taille à la recherche de comportement intéressant et inattendu.

## Abstract

In this internship report, we will present the practical and theoretical study of the set of primitive recursive programs on pure sets through a computational model representing the primitive recursive functions as a series of tokens. To do this, we have developed and optimised an interpreter as well as a program generator. These tools allowed us to exhibit usual programs like arithmetic functions or certain characteristic functions. We also listed and analysed all small programs, looking for interesting and unexpected behaviour.