UNIVERSITÉ DU QUÉBEC À MONTRÉAL

TYPES ÉNUMÉRATIFS ÉVOLUÉS POUR LANGAGES À OBJETS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

MAXIME MULDER

SEPTEMBRE 2023

REMERCIEMENTS

Arrivant au terme de la rédaction de ce mémoire, je tiens à remercier mon directeur de recherche Étienne Gagnon pour ses précieux conseils, son soutien financier, sa sympathie et sa disponibilité tout au long de ma maîtrise. J'ai particulièrement apprécié nos longs débats sur les langages de programmation qui, malgré d'occasionnels désaccords, furent très intéressants et enrichissants.

Je remercie également les autres membres de l'UQÀM m'ayant permis de réaliser ma maîtrise, dont notamment l'administration de la maîtrise informatique ainsi que mes professeurs : Étienne Gagnon (encore), Wessam Ajib, Sébastien Gambs, Sébastien Mosser, Jude Jacob Nsiempba et Srecko Brlek.

Enfin, je remercie mon école en France, le CESI de Rouen, pour ses enseignements et pour m'avoir permis de réaliser cet échange international et ce double diplôme au Canada.

Plus personnellement, je tiens à remercier tous les amis, camarades et rencontres qui ont fait de mon séjour à Montréal une expérience inoubliable. Je veux notamment mentionner Elian, avec qui j'ai voyagé depuis la France et échangé pendant de si longues heures sur l'informatique et tant d'autres sujets, Éric, Lucile, Camille, et celle qui est aujourd'hui ma bien-aimée, Alice.

Finalement, je remercie profondément ma famille, dont notamment mes parents Paul et Cristina, pour leur confiance, leur générosité, et leur soutien inconditionnel qui furent indispensables à l'achèvement de ma maîtrise à l'étranger.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

RÉSUMÉ

Les langages à objets permettent de représenter le monde grâce à des objets, des conteneurs encapsulant des données, des traitements, et pouvant être sujet à des mutations ou à de l'héritage. Dans ce mémoire, nous divisons les objets en deux genres : les *instances*, des objets locaux et éphémères, et les *constantes*, des objets universels. Nous appelons *énumérations* les classes représentant des ensembles de constantes.

Nous cherchons à intégrer les notions d'énumérations et de constantes à un langage à objets. Pour cela, nous avons conçu Enumlang, un langage à objets original intégrant un système d'énumérations évoluées comprenant notamment : un système de définition d'énumérations permettant de vérifier dynamiquement la validité des constantes grâce à l'évaluation de prédicats ; un système d'héritage énumératif permettant de représenter des relations ensemblistes dans un contexte d'objets grâce aux prédicats précédents ; et un système de formes alternatives permettant d'affiner la représentation d'une constante en fonction du contexte.

Grâce aux énumérations évoluées, Enumlang permet de modéliser et manipuler des constantes de manière ergonomique, sûre et efficace. Les fonctionnalités que nous présentons sont alors implémentées dans l'interpréteur Enumlang de manière souvent paresseuse et mémoïsée.

Mots clés : Langages de programmation, programmation orientée objet, héritage, énumérations, objets, constantes, valeurs

INTRODUCTION

Contexte

Les langages de programmation existent afin de faciliter le développement de programmes en permettant de les spécifier de manière compréhensible pour l'humain. Ces langages peuvent être catégorisés en plusieurs paradigmes dont notamment la programmation impérative, la programmation fonctionnelle et la programmation logique, qui permettent chacun d'utiliser voire de manipuler des valeurs de différentes manières.

La programmation orientée objet est un sous-paradigme de la programmation impérative centrée autour du concept d'objets (Wegner, 1990), des entités existant lors de l'exécution d'un programme et regroupant à la fois des données et les traitements qui leur sont applicables. La programmation impérative permettant la modification de l'état d'un programme, les objets peuvent eux-mêmes avoir un état mutable changeant au cours de leur vie.

Il existe plusieurs implémentations possible pour le concept d'objets. Dans un langage à classes tel que Java, chaque objet est lié à une classe qui définit la structure de l'objet, ses méthodes, et ses relations d'héritage avec d'autres classes ou interfaces.

Java possède des types primitifs sans classes tels que `boolean` et `int` dont les valeurs sont comparées *par valeur*. Hors des types primitifs, la comparaison d'objets avec l'opérateur == se fait toujours *par référence*[1], y compris pour des objets

apparemment immutables tels que ceux de la classe `java.lang.String` de la bibliothèque standard du langage (Arnold *et al.*, 2005).

Cette comparaison peut parfois être contre-intuitive, comme dans l'exemple ci-dessous :

```
String a = "Hello␣World␣!";
Strinb b = "Hello";
if (true) {
    b += "␣World␣!";
}

// Affiche "false"
System.out.println(a == b);
```

Exemple de comparaison de `String` en Java

Cet exemple présente une comparaison de deux chaînes de caractères `a` et `b` en Java. Bien que celles-ci valent toutes les deux `"Hello World !"`, leur comparaison par référence avec l'opérateur == retourne **`false`**. Généralement, la méthode `Object.equals` est à privilégier pour la comparaison de chaînes de caractères.

En dehors des modes de comparaison, les objets en Java sont généralement mutables et peuvent référencer d'autres objets, ce que ne font pas les objets `String`. À partir de ces observations, nous énonçons que les `String` de Java correspondent conceptuellement à des objets globaux et intemporels que nous qualifions d'objets *universels*. Nous décrivons les objets universels plus en détails dans la suite de ce mémoire, qui a pour contexte l'étude des objets universels et de leurs classes dans un langage à objets.

---

1. Plus précisément, la comparaison de deux objets en Java se fait par valeur de référence.

Présentation du sujet

À partir des observations précédentes, nous déclarons qu'il existe des différences fondamentales entre les objets standards (que nous appellerons *instances*) et les objets universels (que nous appellerons *constantes*), le terme *objet* nous servant à désigner à la fois les instances et les constantes. Nous utiliserons aussi les termes de classes *instanciatives* pour parler des classes d'instances, et de classes *énumératives*, ou *énumérations*, pour parler des classes de constantes [2].

Plus précisément, nous définissons une instance comme étant un objet dont l'existence est limitée dans l'espace et le temps, c'est-à-dire qui possède une localité et un cycle de vie avec un début, des mutations éventuelles, et une fin. À l'inverse, nous définissons une constante comme un objet global et ne possédant pas de temporalité et donc pas de cycle de vie.

Par exemple, une personne peut être représentée par une instance d'une classe hypothétique `Person`. Cette instance a une localité correspondant à son emplacement dans la mémoire d'une machine et une temporalité correspondant à sa création, aux modifications qu'elle subit au cours du temps, et à son oubli une fois qu'elle n'est plus référencée par le programme auquel elle appartient.

À l'inverse, un nombre entier tel que le nombre `1` peut être représenté par une constante d'une énumération hypothétique `Integer`. Cette constante, comme tout nombre entier, existe indépendamment de toute machine ou programme et ne peut pas muter au cours du temps.

---

2. Notre définition d'*énumération* couvre toutes les classes de constantes et est donc plus large que celle de simples types énumérés tels qu'en C ou en Java.

Objectifs

L'objectif de nos recherches est la conception et l'intégration d'un système d'énumérations évoluées à un langage à objets dans le but de permettre la manipulation de constantes de manière expressive et accessible tout en respectant leur nature d'objets. Dans ce mémoire, nous nous sommes notamment concentrés sur les questions de validité de l'état des constantes, de représentations multiples de cet état et d'héritage des énumérations. Notre système permet par exemple de définir des énumérations représentant les ensembles des nombres rationnels et des entiers, de définir plusieurs représentations valides pour leurs constantes et de les lier par héritage.

Contraintes

Nos recherches correspondent à un travail exploratoire au cours duquel nous avons cherché à définir, étudier et implémenter un ensemble de fonctionnalités énumératives cohérent avec nos objectifs. Nous considérons ce travail comme un travail d'informatique appliquée dans lequel l'implémentation, l'expérimentation et l'itération occupent une place centrale, et non comme une étude purement théorique des énumérations.

Nous avons comme contrainte pour nos travaux l'utilisation d'un langage à objets à la fois complet et accessible. Notre langage, que nous appelons *Enumlang*, doit être un langage à objets intégrant notamment des classes, de l'héritage multiple, de la linéarisation, du typage statique, des opérations de vérification de types dynamiques, et des types paramétriques covariants non effacés. Enumlang est notamment inspiré par le langage Nit (Privat, 2015) dont nous avons considéré l'utilisation avant de préférer la conception d'un langage spécialement adapté à nos besoins.

Nos objectifs principaux étant l'expressivité et l'accessibilité de notre système d'énumérations évoluées, nous acceptons parfois de laisser la responsabilité de la validité du code au programmeur si sa validation statique se ferait au détriment de nos objectifs principaux. Par ailleurs, bien que nous présentons certaines techniques d'interprétation réalistes des énumérations, nos travaux ne sont pas des travaux d'optimisation et nous ne nous concentrons donc pas sur leurs performances pratiques dans ce mémoire.

Méthodologie

Afin de mener à bien nos recherches, nous avons choisi de les diviser en plusieurs étapes :

1. Premièrement, nous avons étudié les constantes et énumérations d'un point de vue hypothétique, en cherchant à déterminer leurs propriétés fondamentales et souhaitables à partir de leurs définitions. Cette étape était principalement composée de recherche de littérature et de réflexion sur ce sujet.
2. Ensuite, nous avons écrit une spécification basique pour un langage à objets original implémentant un système d'énumérations évoluées. Ce langage, nommé *Enumlang*, a servi de base à la suite de nos recherches et est largement utilisé dans ce mémoire.
3. Finalement, nous avons développé un interpréteur pour Enumlang et nous en sommes servi pour expérimenter avec ce langage et son système d'énumérations tout en les améliorant itérativement jusqu'à ce que nous soyons satisfaits de leur cohérence et de leur ergonomie.

Ce mémoire présente les résultats de nos recherches dans un ordre indépendant des étapes que nous venons de décrire.

Contributions

Nous pensons que les contributions principales de nos recherches sont les suivantes :

- Une réflexion sur la dichotomie entre les instances et les constantes, et une définition des propriétés de ces dernières.
- L'introduction d'un nouveau modèle d'énumérations évoluées respectant les propriétés des constantes dans un contexte d'objets.
- L'introduction d'une nouvelle approche d'héritage énumératif basé sur l'évaluation de prédicats.
- L'introduction d'une nouvelle approche de représentations multiples d'une même énumération.
- Une implémentation concrète des notions introduites précédemment dans un langage de programmation original.

Structure du mémoire

Afin de présenter nos recherches, nous avons choisi de développer notre mémoire selon le plan suivant :

- Le premier chapitre présente le langage à objets Enumlang sans les énumérations. Enumlang est largement utilisé pour illustrer ce mémoire avec des exemples de code.
- Le deuxième chapitre présente en détails les concepts de constantes et d'énumérations ainsi que leurs propriétés.
- Le troisième chapitre présente les énumérations d'Enumlang introduisant de nouvelles constantes dans un programme ainsi que les choix effectués quant à leur conception.

- Le quatrième chapitre présente notre approche de l'héritage de classes appliqué aux énumérations ainsi que les définitions d'énumérations parentes et enfantes en Enumlang.
- Le cinquième chapitre présente une fonctionnalité d'Enumlang permettant de définir plusieurs représentations distinctes pour une même énumération.
- Le sixième chapitre présente diverses fonctionnalités plus mineures d'Enumlang permettant d'utiliser les constantes de manière ergonomique.
- Le septième chapitre présente un exemple complet de programme utilisant les énumérations et fonctionnalités liées d'Enumlang en montrant les apports de ces dernières par rapport à un système dépourvu d'énumérations évoluées.
- Le huitième chapitre présente différents travaux liés aux énumérations. Nous nous référons à la fois à des langages de programmation répandus et à des articles plus académiques.
- Finalement, la conclusion récapitule les chapitres précédents en synthétisant les travaux effectués et les principales innovations de nos recherches.

En plus de ses chapitres principaux, notre mémoire contient également les annexes suivantes :

- La première annexe présente la grammaire formelle d'Enumlang.
- La deuxième annexe présente une rapide description de l'interpréteur Enumlang.
- La troisième annexe présente des définitions alternatives pour certaines énumérations primitives d'Enumlang.
- La quatrième annexe présente le code complet de l'exemple du septième chapitre.
- Finalement, la bibliographie liste les différents travaux externes référencés dans ce mémoire.

CHAPITRE I

LANGAGE ENUMLANG

Ce chapitre fournit une description du langage Enumlang. Au cours de nos recherches, celui-ci a servi de base à l'implémentation de notre système d'énumérations, avec lequel nous avons pu expérimenter voire itérer afin de tester sa cohérence et améliorer son ergonomie. Dans ce mémoire, Enumlang nous permet également d'illustrer nos recherches avec des exemples concrets.

Ce chapitre présente le langage Enumlang en allant du plus haut-niveau au plus bas-niveau, en commençant par une description d'ensemble du langage pour finir par les différents genres d'expressions. Nous n'abordons pas dans ce chapitre les fonctionnalités liées aux énumérations, celles-ci étant présentées dans le reste de ce mémoire. Le langage Enumlang ne possédant actuellement pas de spécification complète, ce mémoire peut servir de référence quant à son fonctionnement.

La grammaire complète d'Enumlang est présentée dans l'annexe A et une courte description de l'interpréteur est faite dans l'annexe B.

1.1 Présentation

Enumlang est un langage de programmation impératif, statiquement typé, orienté objet, et intégrant un système d'énumérations évoluées.

Afin de créer des objets, Enumlang permet la définition de classes, qui spécifient la structure de l'objet, ses méthodes, et ses relations d'héritage. Enumlang possède divers genres de classes, dont les énumérations, qui permettent la définition de relations d'héritage multiple.

Le système de typage statique d'Enumlang est utilisé pour valider statiquement la conformité d'un programme Enumlang. Il intègre notamment des fonctionnalités de sous-typage, de typage paramétrique, de vérifications de types dynamiques et d'inférence simple de types.

L'aspect original d'Enumlang est son intégration avancée de diverses fonctionnalités liées au concept d'énumérations. Grâce à celle-ci, Enumlang permet de mieux raisonner à propos des énumérations dans un contexte d'objets.

1.2 Fichiers et modules

Un programme Enumlang est composé d'un ensemble de fichiers de code Enumlang, eux-mêmes divisés en deux parties :

- La première partie spécifie un ensemble de déclarations statiques composant la structure du programme.
- La deuxième partie spécifie une séquence d'instructions à exécuter lors du lancement du programme.

Chacune de ces parties peut également être vide.

L'exemple 1.1 présente un fichier Enumlang contenant une déclaration statique sous forme de la déclaration d'une fonction `hello_world` et une instruction correspondant à un appel de cette fonction.

```
fun hello_world() {
    print("Hello World !");
}

hello_world();
```

FIGURE 1.1 – Exemple de fichier Enumlang

En Enumlang, un module est une déclaration statique permettant de regrouper des déclarations statiques, y compris d'autres modules, dans un espace nommé afin de mieux structurer un programme. L'emplacement et l'ordre d'apparition des déclarations au sein des fichiers et des modules n'a pas d'influence sur la validité d'un programme.

Un module peut être déclaré avec le mot-clé **`module`** suivi de son nom et d'un corps contenant ses déclarations statiques. L'accès à une déclaration d'un module se fait alors avec l'opérateur `.`.

```
module Math {
    fun get_pi(): Float {
        return 3.1415926536;
    }
}

print(Math.get_pi());
```

FIGURE 1.2 – Exemple de module

L'exemple 1.2 présente la déclaration d'un module `Math` contenant une fonction `get_pi`, suivi d'un appel de cette fonction à l'extérieur du module.

Les instructions situées à la fin d'un fichier Enumlang sont exécutées lors du lancement du programme, à la manière d'une fonction `main` dans certains langages.

Un programme pouvant contenir plusieurs fichiers, ces séquences d'instructions sont exécutées fichier par fichier dans l'ordre dans lequel ceux-ci sont fournis.

1.3 Fonctions

Une fonction est une déclaration statique encapsulant une séquence d'instructions pouvant avoir des paramètres ou retourner un objet. En Enumlang, les paramètres et le type de retour d'une fonction sont typés explicitement, bien que ce dernier peut être omis et défini à `Void` par défaut.

En Enumlang, une fonction peut être déclarée avec le mot-clé **`fun`** suivi de son nom, de ses paramètres, de son type de retour, et de son corps. Le mot-clé **`primitive`** peut être utilisé pour annoter une fonction, signalant que son comportement est défini directement par le langage, et que son corps peut donc être omis.

Une fonction peut être appelée en lui fournissant des arguments dont le nombre, l'ordre, et les types correspondent à ses paramètres. L'appel d'une fonction se termine lors de l'exécution d'une instruction **`return`**, qui peut éventuellement retourner un objet, ou à la fin du corps de la fonction, sans retourner d'objet.

```
primitive fun print_string(string: String);

fun print(object: Object) {
    print_string(object.to_string());
}

print(1);
```

FIGURE 1.3 – Exemple de fonctions

L'exemple 1.3 présente les déclarations des fonctions `print_string` et `print`. La fonction `print` est appelée avec le nombre `1` comme argument.

1.4 Classes

Une classe est une déclaration statique décrivant un ensemble d'objets. Il existe plusieurs genres de classes en Enumlang, correspondant chacun à des genres d'objets : une classe instanciative définit des instances, une classe énumérative définit des constantes, et une interface s'applique à tout genre d'objets.

Une classe peut être déclarée avec le mot-clé **`class`**, **`enum`** ou **`interface`**, selon son genre, suivi de son nom et d'un corps contenant ses membres.

Une classe peut par ailleurs être concrète ou abstraite, selon qu'elle permette ou non de créer directement des objets de son type. Toutes les interfaces sont abstraites, et le mot-clé **`abstract`** permet d'annoter une classe instanciative comme étant abstraite.

Le mot-clé **`super`** suivi d'un parent permet de déclarer une relation d'héritage entre des classes. Une classe hérite non seulement des parents qu'elle déclare, mais aussi récursivement de tous leurs parents respectifs. Enumlang supporte l'héritage multiple, permettant à une classe d'hériter directement de plusieurs autres classes. Toute classe n'est héritée qu'une seule fois, même si elle est accessible par plusieurs liens d'héritage dans une classe donnée.

Il existe plusieurs restrictions sur l'héritage. Toutes les classes peuvent hériter d'interfaces, mais une classe instanciative ne peut pas hériter d'une classe énumérative et inversement. L'héritage énumératif est également sujet à d'autres contraintes décrites dans le chapitre 4.

Une classe peut déclarer des méthodes, avec une syntaxe similaire à celle des fonctions. Elle hérite également de toutes les méthodes de ses parents voire peut en redéfinir en annotant une de ses méthodes avec le mot-clé **redef** si sa signature correspond à celle de la méthode héritée. Une classe abstraite peut aussi déclarer une méthode abstraite en l'annotant avec le mot-clé **abstract**, indiquant que cette méthode doit être redéfinie dans les classes concrètes qui en héritent.

Toute classe doit posséder une version strictement plus spécifique de chacune de ses méthodes, empêchant toute classe d'hériter de plusieurs versions redéfinies d'une même méthode, à moins qu'elle ne la redéfinisse elle-même. Cette version plus spécifique est alors appelée à chaque appel de la méthode sur un objet de cette classe.

Finalement, une classe instanciative peut définir des attributs, déclarés chacun avec le mot-clé **var** suivi de son nom et de son type. Similairement aux méthodes, une classe instanciative hérite de tous les attributs de ses parents et peut elle-même définir de nouveaux attributs.

```
interface Stringable {
    abstract fun stringify(): String;
}

# Déclaration d'une classe instanciative
class Person {
    super Stringable;

    var name: String;

    redef fun stringify(): String {
        return self.name;
    }
}
```

FIGURE 1.4 – Exemple de classes

L'exemple 1.4 présente la déclaration d'une interface `Stringable` contenant une méthode abstraite `stringify`, ainsi que la déclaration d'une classe `Person` héritant de cette interface, contenant un attribut `name` et redéfinissant la méthode `stringify`.

1.5 Déclarations paramétriques

Les classes, les fonctions et les méthodes sont des déclarations paramétriques, c'est-à-dire des déclarations statiques qui peuvent posséder des paramètres génériques. Ces paramètres permettent aux déclarations paramétriques d'être définies en faisant abstraction des types précis qui leurs sont substitués lors de leurs diverses utilisations. Chaque paramètre générique possède une borne, valant `Object` par défaut, qui indique les relations de typage qu'il doit satisfaire.

```
# Classe paramétrique
class List[T] {
    # ...
}

# Fonction paramétrique
fun new_list[T](): List[T] {
    return new List[T];
}

var integers: List[Integer] = new_list[Integer]();
```

FIGURE 1.5 – Exemple de déclarations paramétrique

L'exemple 1.5 présente des déclarations et usages d'une classe `List` et d'une fonction `new_list` ayant toutes deux un paramètre générique `T`.

Enumlang est un langage avec de la covariance générique, c'est-à-dire dont les relations de sous-typage s'étendent aux paramètres génériques, signifiant par exemple

que `List[String]` est un sous-type de `List[Object]`, puisque `String` est un sous-type de `Object`.

La covariance générique a pour avantage de simplifier l'écriture de code paramétrique. Cependant, elle peut également causer des problèmes de sûreté de types, qui doivent être détectés et rapportés lors de l'exécution du programme.

```
fun append(list: List[Object], object: Object) {
    list.append(object);
}

var integers = new List[Integer];
append(integers, "Hello World !");
```

FIGURE 1.6 – Exemple d'erreur dynamique de typage paramétrique

Dans l'exemple 1.6, l'appel de la fonction `append` tente d'insérer un objet de type `String` dans une liste de type `List[Integer]`. Enumlang détecte dynamiquement cette incohérence et soulève une erreur.

Finalement, le typage paramétrique possède également quelques restrictions statiques. Notamment, un paramètre générique ne peut pas être utilisé comme parent d'une classe ou pour instancier une instance.

1.6 Linéarisation

En Enumlang, dans une méthode redéfinie, l'expression **`super`** permet d'appeler la méthode parente et d'obtenir l'objet qu'elle retourne. Cet appel est effectué avec les mêmes arguments que ceux envoyés à la méthode dans laquelle il apparaît. Évidemment, **`super`** ne peut pas être utilisé dans une méthode sans méthode parente ou dans laquelle la méthode parente est abstraite.

En présence d'héritage multiple, une méthode peut avoir plusieurs parents dont les spécificités sont égales, rendant la méthode à appeler par **`super`** ambigüe. Pour résoudre ce problème, Enumlang procède à la linéarisation de la hiérarchie de classes afin de définir un ordre strict de ses classes. Cet ordre permet alors à **`super`** de sélectionner une méthode à appeler selon sa priorité dans la classe du receveur.

L'algorithme de linéarisation d'Enumlang cherche à satisfaire les trois contraintes suivantes :

— **Cohérence** : Toute classe parente doit apparaître avant ses enfants dans une linéarisation.
— **Consistance** : Toute classe apparaissant avant une autre classe dans une linéarisation doit aussi apparaître avant cette classe dans toute autre linéarisation.
— **Complétude** : Toute hiérarchie de classe doit avoir une linéarisation.
— **Unicité** : Toute hiérarchie de classe ne peut avoir plus d'une linéarisation.

Afin de régler les conflits de spécificité lors de la linéarisation, Enumlang utilise l'approche de Nit consistant à comparer l'ordre de déclaration des classes dans le programme (Privat, 2015). Bien que relativement arbitraire, cette approche permet de satisfaire l'ensemble des contraintes énoncées précédemment contrairement à certains autres algorithmes tels que C3 (Barrett *et al.*, 1996).

L'exemple 1.7 présente la linéarisation d'une hiérarchie d'héritage en losange contenant les classes `A`, `B`, `C` et `D`, qui sont linéarisées dans ce même ordre.

```
class A {
    fun get(): String {
        return "A";
    }
}

class B {
    super A;

    redef fun get(): String {
        return super + "B";
    }
}

class C {
    super A;

    redef fun get(): String {
        return super + "C";
    }
}

class D {
    super B;
    super C;

    redef fun get(): String {
        return super + "D";
    }
}

# Affiche "ABCD"
print((new D).get());
```

FIGURE 1.7 – Exemple de linéarisation

1.7 Instructions

En plus des déclarations statiques, un programme Enumlang est également composé de séquences d'instructions exécutables.

1.7.1 Instructions basiques

Les instructions basiques d'Enumlang désignent ses instructions se terminant par un point-virgule :

- La déclaration de variable est formée du mot-clé **`var`** suivi du nom de la variable et éventuellement de son type, qui vaut `Object` par défaut.
- L'assignement est formé d'une référence (une déclaration de variable, un nom de variable, ou un accès à un attribut) suivi de l'opérateur d'assignement = et d'une expression. Si la référence est une déclaration de variable, son type par défaut est celui de l'expression assignée.
- Le retour est formé du mot-clé **`return`** suivi éventuellement d'une expression correspondant à l'objet à retourner.
- L'expression est formée d'une expression.

```
# Déclaration de variable
var name: String;
# Assignement
name = "John";
# Expression
print(name);
# Saut
return 0;
```

FIGURE 1.8 – Exemple d'instructions basiques

1.7.2 Instructions de contrôle

Les instructions de contrôle d'Enumlang désignent ses instructions se terminant par un bloc, qui peut lui-même contenir d'autres instructions :

- L'instruction de bloc permet de regrouper plusieurs instructions en une tout en limitant la portée de leurs variables.
- L'instruction conditionnelle **`if`** permet d'exécuter un bloc d'instructions si une condition booléenne est vraie. L'instruction conditionnelle peut également être terminée par un **`else`** afin d'exécuter un autre bloc d'instructions si la condition est fausse.
- L'instruction de boucle **`loop`** permet de répéter l'exécution d'un bloc d'instructions.
- L'instruction de boucle **`while`** permet de répéter l'exécution d'un bloc d'instructions tant qu'une condition booléenne est vraie.
- L'instruction de boucle **`for`** permet de répéter l'exécution d'un bloc d'instructions tout en itérant sur un objet avec une variable.

Les expressions des instructions **`if`** et **`while`** doivent être de type `Boolean`, et celle de l'instruction **`for`** doit être de type `Iterable[T]`, `T` servant alors de type à la variable associée à cette instruction.

Le fonctionnement de l'instruction **`for`** est basé sur les classes `Iterable`, `Iterator` et `Maybe` de la bibliothèque standard d'Enumlang. Lors de son exécution, l'expression itérable est évaluée afin d'appeler la méthode `Iterable.iter` sur l'objet produit pour obtenir un itérateur. À chaque itération, la méthode `Iterator.next` est appelée sur l'itérateur, si l'objet optionnel retourné contient un objet, alors ce dernier est assigné à la variable et le bloc est exécuté, sinon, l'exécution de l'instruction **`for`** se termine.

1.8 Expressions

Les instructions d'Enumlang sont souvent composées d'expressions, pouvant elles-même être composées d'autres expressions. Toute expression a un type et produit un objet de ce type lors de son évaluation.

1.8.1 Littéraux

Enumlang permet de produire des objets primitifs à l'aide d'expressions littérales :

- Un booléen peut être spécifié en utilisant un mot-clé **`true`** ou **`false`**.
- Un nombre entier peut être spécifié en écrivant le nombre en question en base 10.
- Une chaîne de caractères peut être spécifiée en écrivant cette dernière délimitée par des guillemets.

```
var boolean = true;
var string = "Hello World !";
var integer = 0;
```

FIGURE 1.9 – Exemple d'objets littéraux

1.8.2 Coercitions

Les expressions de coercitions de type permettent d'effectuer des opérations sur un objet ayant rapport à son type :

- L'expression **`isa`** produit un booléen permettant de vérifier si un objet est d'un type donné.
- L'expression **`as`** permet de convertir un objet à un type donné. Une erreur est soulevée si l'objet n'est pas de ce type.

Les expressions **isa** et **as** fonctionnent également avec des types génériques.

1.8.3 Opérateurs

Enumlang possède divers opérateurs binaires infixes :

- Les opérateurs logiques avec court-circuitage `&&`, `||`.
- Les opérateurs de comparaison == et `!=`.
- Les opérateurs de comparaison d'ordre <, >, <= et >=.
- Les opérateurs arithmétiques +, −, ⋆, / et %.

Enumlang possède également quelques opérateurs unaires préfixes :

- Les opérateurs arithmétiques + et −.
- L'opérateur logique de négation `!`.

1.8.4 Divers

Enumlang possède finalement d'autres expressions diverses :

- Le mot-clé **new** permet de créer une nouvelle instance du type donné.
- Le mot-clé **self** permet de référencer le receveur de la méthode en cours d'exécution.
- Le mot-clé **super** permet d'appeler la méthode parente de celle en cours d'exécution.
- Un identifiant permet de référencer une variable ou une fonction.
- Une fonction ou une méthode peut être appelée en lui fournissant une liste d'arguments entre parenthèses.
- Un attribut peut être référencé en faisant suivre le receveur d'un point et du nom de l'attribut.

1.9 Conclusion

Ce chapitre a présenté Enumlang, un langage de programmation impératif, statiquement typé et orienté objet implémentant un système d'énumérations évoluées.

Ce chapitre a présenté les bases du langage Enumlang ainsi que son interpréteur. Un programme Enumlang est composé de fichiers et de modules, contenant chacun diverses déclarations statiques, dont notamment des classes et des fonctions. Un programme Enumlang contient également des séquences d'instructions et des expressions, telles que des structures de contrôle, des appels de fonctions ou de méthodes, des conversions de types ou d'autres opérations diverses.

Le système de typage statique d'Enumlang intègre des fonctionnalités de sous-typage, de typage paramétrique, d'inférence basique et de vérification dynamique. Ces types sont définis par le biais de classes qui supportent l'héritage multiple en utilisant notamment un système de linéarisation.

CHAPITRE II

PRÉSENTATION DES ÉNUMÉRATIONS

Ce chapitre présente les notions de constantes et d'énumérations sur lesquelles est fondé Enumlang, sans toutefois rentrer dans le détail quant à leur utilisation dans ce langage.

Dans la première section, nous comparons les notions d'instances et de constantes avec notamment quelques exemples concrets mettant en avant leurs différences.

Dans la deuxième section, nous présentons la notion d'énumérations, qui sont des classes énumératives représentant des ensembles de constantes.

Dans la troisième section, nous listons quelques propriétés ensemblistes des énumérations.

2.1 Instances et constantes

2.1.1 Définitions

Nos recherches ont pour contexte la catégorisation des objets en deux genres : les *instances* et les *constantes*. Sous leur forme la plus simple, nous définissons ces deux notions comme suit :

— Une instance est un objet local et temporel.

— Une constante est un objet global et intemporel.

À partir de ces définitions, nous déterminons les propriétés suivantes pour les instances et les constantes :

— Une instance étant locale, elle peut n'être accessible que dans un nombre limité de contextes.
— Une instance étant temporelle, elle peut avoir un cycle de vie avec un début et une fin, voire subir des mutations au cours de celui-ci.
— Une constante étant globale, elle est accessible dans n'importe quel contexte.
— Une constante étant intemporelle, elle n'a pas de cycle de vie avec un début et une fin, et ne peut pas subir de mutations, ce qui fait d'elle un objet immutable.

À la vue de ces propriétés, nous considérons que les instances s'apparentent généralement aux objets « classiques » de langages tels que Java ou C#, définis dans des classes, instanciés avec l'opérateur **`new`**, et dont la mutabilité n'est par défaut pas restreinte par le langage[1]. À l'inverse, les constantes correspondent plus aux valeurs primitives de ces langages, telles que celles de type `boolean`, `int` ou `float`.

2.1.2 Localité

L'un des aspects différenciant les instances des constantes est leur localité : une instance ne peut être accédée que par le biais d'une référence, dont la disponibilité varie en fonction du contexte. Dans l'exemple 2.1, la variable `rocky` se fait

1. L'ajout des mots-clés `final` en Java et `readonly` en C# permettent de restreindre la mutabilité d'attributs ou de variables.

assigner une nouvelle instance du type `Dog`. Ensuite, la fonction `do_something` est appelée. En l'absence de référence passée en argument à cette fonction, celle-ci ne peut pas accéder à l'instance de la variable `rocky`, malgré que cette instance soit encore valide dans d'autres endroits du programme.

```
var rocky = new Dog;
rocky.name = "Rocky";
do_something();
```

FIGURE 2.1 – Exemple de localité d'une instance

Contrairement aux instances, les constantes sont globales : elles sont accessibles dans n'importe quel contexte donné. Dans l'exemple 2.2, la variable `seven` se voit assigner la constante `7`. Ensuite, la méthode `do_something`, qui n'a pas accès à la variable précédente, réobtient tout de même cette constante par le biais d'un calcul et l'assigne à une nouvelle variable.

```
fun do_something() {
    var seven_bis = 3 + 4;
}

var seven = 7;
do_something();
```

FIGURE 2.2 – Exemple de globalité d'une constante

Conceptuellement, la différence de localité entre les constantes et les instances s'étend au-delà du périmètre d'un seul programme. Là où une instance ne peut exister que dans un unique programme donné [2], une constante peut exister dans de

multiples programmes voire machines simultanément, la constante `2` représentant par exemple le même nombre indépendamment de son contexte d'exécution.

2.1.3 Temporalité

Bien que très lié à la localité, un autre aspect différenciant les instances des constantes est leur temporalité, puisqu'une instance possède notamment un cycle de vie :

1. La vie d'une instance possède un début correspondant à son instant de création. Dans l'exemple 2.3, l'exécution de l'expression **`new`** `Dog` crée une nouvelle instance assignée à la variable `luna`.
2. Ensuite, une instance peut éventuellement subir des mutations. Dans l'exemple 2.3, l'exécution de l'expression `luna.name = "Luna"` fait muter l'instance de la variable `luna` en modifiant l'un de ses attributs.
3. Finalement, une instance est oubliée lorsque son programme ne lui fait plus référence [3]. Dans l'exemple 2.3, l'instance de la variable `luna` devient inaccessible après l'exécution du bloc de code dans lequel elle est créée. Il n'est alors plus possible de la retrouver même en créant une nouvelle instance identique.

Contrairement à une instance, une constante n'a pas de cycle de vie. Dans l'exemple 2.4 la constante `2` existe avant son assignation à la variable `two`, ne

---

2. Nous pouvons mentionner certaines fonctionnalités telles que la sérialisation et la programmation distribuée permettant d'accéder à une instance depuis des contextes éloignés. Ces accès n'étant cependant pas universels et étant soumis aux aléas de leurs systèmes et réseaux hôtes, nous considérons qu'ils ne sont que des proxys et que toute instance existe uniquement en un point correspondant à son emplacement en mémoire.

3. En théorie, l'exécution de certains programmes peut ne pas avoir de fin, et donc garder des instances en vie indéfiniment. Nous considérons cependant qu'en pratique tout programme se termine, soit en arrivant au terme d'une exécution normale, soit par une intervention extérieure.

```
{
     var luna = new Dog;
     luna.name = "Luna";
}

var luna_bis = new Dog;
```

FIGURE 2.3 – Exemple de temporalité d'une instance

peut pas muter, et peut être réobtenue même si son programme ne lui fait plus référence. L'exécution de l'expression `1 + 1` permet justement de retrouver la constante `2` bien que la variable `two` ne soit plus valide à cet instant.

```
{
     var two = 2;
     calculate(two);
}

var two_bis = 1 + 1;
```

FIGURE 2.4 – Exemple d'intemporalité d'une constante

2.1.4 Instanciation et obtention

La création d'une instance, qui marque le début de son cycle de vie, est nommée *instanciation*. Dans un langage tel que Java ou C#, cela correspond à l'exécution d'une expression **new**.

Une constante étant intemporelle, elle n'est jamais réellement instanciée ou créée. Nous utilisons donc le terme d'*obtention* pour parler de l'acquisition d'une constante par un programme.

2.2 Énumérations

En programmation orientée objet, une classe est une déclaration permettant de définir des objets et les propriétés qui leur sont liées. Il peut exister plusieurs genres de classes dans un langage telles que les classes concrètes, abstraites ou les interfaces en Java (Arnold *et al.*, 2005).

Nous appelons classe *instanciative* une classe permettant de définir des instances, et classe *énumérative*, ou plus simplement *énumération*, une classe permettant de définir des constantes. Nous utilisons le terme *interface* pour désigner une classe pouvant couvrir à la fois des instances et des constantes.

2.3 Propriétés des énumérations

2.3.1 Ensemblisme

Une classe peut être considérée comme un ensemble, auquel chaque objet de cette classe appartient. Nous insistons alors sur la nature ensemblistes des énumérations, qui permettent de définir des ensembles de constantes, et sont donc sujettes aux diverses propriétés et relations des ensembles.

Par exemple, nous utilisons les relations de sur-ensembles et sous-ensembles entre énumérations pour définir de l'héritage énumératif dans le chapitre 4.

Par ailleurs, nous pouvons noter qu'une énumération est un ensemble fixe, puisque toute constante appartient ou non à une énumération, et que cette appartenance ne change pas au cours du temps.

2.3.2 Cardinalité

La cardinalité d'un ensemble est une propriété désignant le nombre d'éléments de cet ensemble, qui peut être fini comme infini. Une énumération représentant un ensemble de constantes, toute énumération possède une cardinalité correspondant à son nombre total de constantes, indépendamment de leur utilisation ou non dans un programme.

Chaque énumération définissant ses constantes de manière différente, la cardinalité d'une énumération doit être calculée individuellement pour chaque énumération. Les calculs de cardinalité des constantes sont présentés pour chaque genre d'énumérations dans les chapitres 3 et 4.

La connaissance de la cardinalité d'une énumération est principalement utilisée pour effectuer des analyses statiques, comme notamment la vérification d'exhaustivité des aiguillages énumératifs décrite dans la section 6.3.

Finalement, en considérant une classe instanciative comme un ensemble, nous pouvons noter que toute classe instanciative a une cardinalité infinie puisque qu'il est possible de créer un nombre non borné d'instances, chaque instanciation créant une nouvelle instance différente des précédentes.

2.4 Conclusion

Ce chapitre a présenté les notions de constantes et d'énumérations essentielles à la compréhension d'Enumlang. Les constantes sont des objet universels et immutables qui s'opposent aux instances, des objets locaux et éphémères.

Les instances sont regroupées dans des classes instanciatives et les constantes dans des énumérations. Ces énumérations représentent des ensembles et sont donc sujettes à diverses propriétés et relations ensemblistes telles que la cardinalité.

CHAPITRE III

ÉNUMÉRATIONS INTRODUCTRICES

Ce chapitre présente les énumérations introductrices, c'est-à-dire les énumérations définissant de nouvelles constantes au sein d'un programme. Enumlang possède plusieurs genres d'énumérations introductrices afin de permettre de représenter expressivement la diversité des constantes.

Dans la première section, nous présentons la notion d'énumération introductrice.

Dans la deuxième section, nous présentons les différentes structures d'énumérations introductrices ainsi que leurs syntaxes de déclaration et d'obtention de constantes.

Dans la troisième section, nous présentons les différents choix que nous avons effectués quant à la conception des énumérations décrites dans ce chapitre.

3.1 Introduction de constantes

Toute énumération représente un ensemble de constantes. L'un des fondements d'Enumlang est l'idée qu'une constante peut appartenir à plusieurs énumérations. Pour cela, nous avons choisi de diviser les énumérations d'Enumlang en deux catégories :

— Une énumération *introductrice* est une énumération introduisant de nouvelles constantes ainsi que leur structure dans un programme.
— Une énumération *non introductrice* est une énumération définie par héritage et réutilisant les constantes et structures d'autres énumérations afin de définir son ensemble de constantes.

Ces deux genres d'énumérations sont mutuellement exclusifs. Ainsi, une énumération ne peut pas introduire de nouvelles constantes et utiliser des constantes d'autres énumérations, ce qui signifie qu'une énumération introductrice ne peut pas hériter d'une autre énumération introductrice.

Toute constante est donc introduite par une unique énumération introductrice, qui définit notamment l'agencement de ses données par le biais de sa structure. Une structure ne peut pas être vide et doit donc permettre la représentation d'au moins une constante.

3.2 Structure des énumérations

3.2.1 Énumérations listées

Une énumération listée est une énumération introductrice composée de constantes nommées listées exhaustivement. La cardinalité d'une énumération listée correspond au compte de ses constantes nommées.

Comme toute énumération introductrice en Enumlang, une énumération listée est déclarée avec le mot-clé **`enum`** suivi de son nom et de son corps. Une énumération listée est alors une énumération comportant dans son corps une une ou plusieurs constantes nommées déclarées avec le mot-clé **`case`** suivi du nom de la constante.

```
enum Suit {
    case clubs;
    case diamonds;
    case hearts;
    case spades;
}
```

FIGURE 3.1 – Exemple de déclaration d'une énumération listée

L'exemple 3.1 présente une énumération listée `Suit` comportant les quatre constantes `clubs`, `diamonds`, `hearts` et `spades`.

Une constante nommée peut être obtenue en tant que membre de son énumération, avec la même syntaxe que l'accès à un membre de module.

```
var suit = Suit.clubs;
```

FIGURE 3.2 – Exemple d'obtention d'une constante nommée

L'exemple 3.2 présente l'obtention de la constante nommée `clubs`, de l'énumération `Suit`.

3.2.2 Énumérations primitives simples

Une énumération primitive est une énumération introductrice dont la structure est intégrée au langage. Enumlang possède trois énumérations primitives non paramétriques :

— L'énumération `Boolean` représente un booléen et comporte donc les deux constantes **`true`** et **`false`**. Cette énumération pourrait également être définie comme une énumération listée comme décrit dans l'annexe C.1.

— L'énumération `Integer` représente l'ensemble des nombres entiers. Les constantes de type `Integer` sont signées, non bornées, et intérieurement représentées avec un *big integer*[1].

— L'énumération `String` représente l'ensemble des chaînes de caractères. Les constantes de type `String` sont intérieurement représentées avec des caractères Unicode.

Les cardinalités des énumérations primitives simples d'Enumlang sont décrites dans le tableau 3.1.

Énumération	**Cardinalité**
`Boolean`	2
`Integer`	∞
`String`	∞

TABLEAU 3.1 – Tableau des cardinalités des énumérations primitives simples

Les constantes des énumérations primitives simples peuvent être obtenues statiquement grâce aux expressions littérales décrites dans la section 1.8.1. Certains types primitifs communs tels que `Character` et `Float` ne sont actuellement pas implémentés dans Enumlang car nous ne les jugeons pas essentiels à la réalisation nos recherches.

3.2.3 Énumérations à attributs

Une énumération à attributs est une énumération introductrice structurée avec des attributs. Sa cardinalité correspond au produit des cardinalités de ses attributs, ce qui fait d'une énumération à attributs un type produit (Pierce, 2002).

---

1. Un *big integer* est un entier non borné. Dans l'interpréteur Enumlang, nous utilisons la classe `java.math.BigInteger` à cet effet.

Un attribut énumératif est déclaré avec le mot-clé **var**, similairement à un attribut d'instance. Toute énumération comportant un ou plusieurs attributs est une énumération à attributs. Un attribut énumératif ne peut être typé que par une énumération ou un paramètre générique.

```
enum Point {
    var x: Integer;
    var y: Integer;
}
```

FIGURE 3.3 – Exemple de déclaration d'une énumération à attributs

L'exemple 3.3 présente une énumération à attributs `Point` comportant deux attributs `x` et `y`.

Une constante à attributs peut être obtenue par le biais d'une fonction d'obtention liée à son énumération. Les paramètres de cette fonction correspondent alors aux attributs de l'énumération, et sont dans le même ordre et ont les mêmes types que ces derniers.

```
var origin = Point(0, 0);
```

FIGURE 3.4 – Exemple d'obtention d'une constante à attributs

L'exemple 3.4 présente l'obtention d'une constante de l'énumération `Point`, dont les deux attributs sont égaux à `0`.

3.2.4 Énumérations à prédicat

Une énumération à prédicat est une énumération dont la définition est affinée par un test booléen nommé *prédicat* vérifiant dynamiquement l'appartenance de

chaque constante à cette énumération. Ce prédicat est alors un code évalué lors de toute obtention de constante de cette énumération et soulevant une erreur en cas de non-appartenance de la constante.

Un prédicat est déclaré avec le mot-clé **predicate** suivi d'un corps. Dans ce corps, le mot-clé **self** réfère à la constante receveuse et l'instruction **return** permet de retourner le booléen indiquant ou non l'appartenance de la constante à l'énumération à prédicat.

```
enum Fraction {
    var numerator: Integer;
    var denominator: Integer;

    predicate {
        return self.denominator != 0;
    }
}
```

FIGURE 3.5 – Exemple d'énumération à prédicat

L'exemple 3.5 présente une énumération à attributs `Fraction` possédant un prédicat s'assurant que le dénominateur de ses constantes soit différent de 0.

Puisque **self** réfère potentiellement à une constante invalide pendant l'exécution du prédicat, dû à la possibilité que cette exécution se termine par le retour de la valeur **false**, l'utilisation de **self** est restreinte dans le corps du prédicat au seul accès aux attributs. Par exemple, l'appel d'une fonction ou d'une méthode avec **self** ainsi que son assignation dans une variable sont statiquement rejetés [2]. Ceci assure qu'une référence à une constante potentiellement invalide ne peut d'aucune façon se propager hors du corps du prédicat.

---

2. Par manque de temps, ces restrictions n'ont pas été implémentées dans l'interpréteur Enumlang.

Toute énumération représentant un ensemble fixe, le statut d'acceptation retourné par un prédicat pour une constante donnée doit également être fixe. Un prédicat ayant pour seul paramètre sa constante receveuse, un prédicat pur comme décrit dans la section 3.3.4 répond à cette contrainte d'invariabilité. Nous appelons *énumération évaluative* toute énumération demandant l'évaluation de code pour déterminer l'appartenance d'une constante à cette dernière.

La cardinalité d'une énumération évaluative dépend du nombre de constantes acceptées par son prédicat. En l'incapacité de déterminer statiquement ce nombre, bien qu'il soit borné par la cardinalité de cette énumération en l'absence de son prédicat, Enumlang considère que toute énumération évaluative a une cardinalité inconnue.

L'obtention d'une constante d'une énumération à prédicat utilise la même fonction d'obtention que les énumérations à attributs. Cependant, toute nouvelle constante est accompagnée par une évaluation du prédicat de l'énumération qui soulève une erreur en cas de refus de la constante.

```
var one = Fraction(1, 1); # Constante acceptée
var four = Fraction(4, 0); # Constante refusée
```

FIGURE 3.6 – Exemple d'obtentions de constantes avec prédicats

L'exemple 3.6 présente l'obtention de deux constantes d'une énumération à prédicat `Fraction`. La première constante est acceptée, mais la deuxième, représentant une fraction dont le dénominateur est égale à 0, soulève une erreur lors de l'exécution du programme.

3.2.5 Énumérations paramétriques

Une énumération paramétrique est une énumération ayant un ou plusieurs paramètres génériques. La déclaration d'une énumération paramétrique est similaire à toute déclaration de classe paramétrique.

```
enum Pair[T] {
    var first: T;
    var second: T;
}
```

FIGURE 3.7 – Exemple d'énumération paramétrique

L'ensemble de constantes représenté par une énumération dépend notamment de sa structure, et donc de ses attributs dans le cas d'une énumération à attributs. Un paramètre générique pouvant être utilisé pour typer un attribut, l'ensemble représenté par une énumération peut changer selon les arguments génériques qui lui sont fournis. Dans l'exemple 3.7, les ensembles représentés par les énumérations `Pair[String]` et `Pair[Integer]` sont notamment différents.

Un attribut énumératif ne pouvant être typé que par une énumération, un paramètre générique ne peut pas avoir de classe instanciative en borne générique. Nous autorisons cependant l'utilisation d'une interface en tant que borne générique tant que celle-ci est ensuite substituée par une énumération. La borne par défaut des paramètres génériques des énumérations est par ailleurs, comme pour les autres genres de classes, l'interface `Object`.

En Enumlang, similairement aux instances, les arguments génériques d'une constante sont inclusent dans cette dernière. Ainsi, une constante obtenue avec `Sequence[String]` ne peut par exemple pas être égale à une constante obtenue avec `Sequence[Integer]`.

```
enum Pair[T] {
    var first: T;
    var second: T;
}

var a = Pair[Odd](1, 1);
var b = Pair[Integer](1, 1);

# Affiche "true"
print(a isa Pair[Odd]);
# Affiche "true"
print(a isa Pair[Integer]);
# Affiche "false"
print(b isa Pair[Odd]);
# Affiche "true"
print(b isa Pair[Integer]);
```

FIGURE 3.8 – Exemple de constantes paramétriques

En considérant une énumération `Odd` héritant de `Integer` comme défini dans le chapitre 4, l'exemple 3.8 présente deux constantes `a` et `b` respectivement de type `Pair[Odd]` et `Pair[Integer]`. Les deux constantes appartiennent à l'énumération `Pair[Integer]` mais contrairement à `a`, `b` n'appartient pas à l'énumération `Pair[Odd]` puisque son type n'est pas un sous-type de cette dernière.

Une constante dépendant de ses arguments génériques et le nombre de sous-énumérations d'une énumération donnée étant non borné (chapitre 4), la cardinalité d'une énumération paramétrique est considérée comme infinie.

3.2.6 Énumérations primitives paramétriques

Enumlang possède deux énumérations primitives paramétriques :

— L'énumération `Maybe[T]` représente l'ensemble des constantes appartenant à l'énumération `T` ou leur absence. Elle permet de remplacer la valeur **`null`** qui existe dans divers langages (Chalin et Rioux, 2005; Amin et Tate, 2016).
— L'énumération `Sequence[T]` représente l'ensemble des séquences de constantes appartenant à l'énumération `T`.

Les constantes des énumérations primitives paramétriques `Sequence[T]` et `Maybe[T]` peuvent être obtenues grâce à plusieurs fonctions d'obtention primitives :

— La fonction variadique `Sequence[T](..T): Sequence[T]` permet d'obtenir une constante de type `Sequence[T]` composée des constantes qui lui sont envoyées en arguments dans le même ordre.
— La fonction `some[T](T): Maybe[T]` permet d'obtenir une constante de type `Maybe[T]` composée de la constante qui lui est envoyée en argument.
— La fonction `none[T](): Maybe[T]` permet d'obtenir une constante de type `Maybe[T]` ne contenant aucune constante `T`.

```
var integers = Sequence[Integer](1, 2, 3);
var integer_some = some[Integer](0);
var integer_none = none[Integer]();
```

FIGURE 3.9 – Exemple d'obtentions de constantes primitives paramétriques

L'exemple 3.9 présente une série d'obtentions de constantes d'énumérations primitives paramétriques d'Enumlang.

Les énumérations primitives paramétriques d'Enumlang pourraient également être définies avec des combinaisons d'énumérations non primitives comme décrit dans les annexes C.2 et C.3. Nous avons choisi d'en faire des énumérations primitives dans un but de simplicité et d'optimisation.

3.3 Choix de conception des énumérations

3.3.1 Constantes nulles

En programmation, la valeur **`null`** est une valeur ne référençant aucun objet (Fähndrich et Leino, 2003). Il existe différentes approches quant à l'intégration, ou non, de **`null`** dans un langage de programmation. Par exemple, en Java, toute valeur contenant une référence peut valoir **`null`** (Arnold *et al.*, 2005), en C#, à partir de la version 2.0 du langage, seule une valeur nullable peut valoir **`null`** (Hejlsberg *et al.*, 2003), et en Scala, aucune valeur ne peut être **`null`** (Odersky *et al.*, 2004).

La présence de la valeur **`null`** ajoute une nouvelle valeur possible à toutes les variables qu'elle peut affecter, augmentant donc leur cardinalité. Par exemple, en présence de **`null`**, une variable booléenne peut valoir **`true`**, **`false`** ou **`null`**, soit 3 valeurs au lieu de 2 comme le veut la cardinalité de `Boolean`.

Les notions d'ensemble et de cardinalité étant centraux au concept d'énumération, nous pensons qu'Enumlang ne devrait pas intégrer de valeur **`null`**. En l'absence de **`null`**, nous préférons utiliser l'énumération optionnelle `Maybe[T]`, qui correspond à un type optionnel tel qu'il en existe en Haskell (`Maybe`) (Marlow *et al.*, 2010) ou Scala (`Option`) (Odersky *et al.*, 2004). Nous utilisons également cette approche de classe optionnelle pour les instances.

3.3.2 Énumérations récursives

Un type récursif est un type défini comme se composant lui-même (Pierce, 2002). Ainsi, un objet d'un type récursif peut être composé d'un autre objet du même type, voire de lui-même dans le cas d'un objet cyclique. Les énumérations à at-

tributs sont des types produits, pour lesquels il nous convient donc d'accepter ou non la récursion.

Enumlang est un langage à évaluation stricte[3] ne possédant pas de mécanisme spécifique permettant de modéliser, obtenir ou manipuler des constantes infinies.

En présence d'énumérations sommes telles que `Maybe`, il est cependant possible de définir des énumérations récursives dont la récursion n'est pas nécessairement infinie. Le niveau de récursion d'un type n'étant pas borné, la cardinalité d'une énumération récursive est infinie.

```
# Il est impossible d'obtenir une constante de cette énumé
   ration en Enumlang
enum Infinite {
    var attribute: Infinite;
}
```

FIGURE 3.10 – Exemple d'énumération à attributs récursive infinie

```
enum Finite {
    # « Maybe » agit comme un type somme
    var attribute: Maybe[Finite];
}

# Obtention d'une constante de l'énumération récursive « Finite
    »
var constant = Finite(some[Finite](Finite(none[Finite]()));
```

FIGURE 3.11 – Exemple d'énumération à attributs récursive

L'exemple 3.10 nous présente une énumération `Infinite` récursive infinie. À l'inverse, l'exemple 3.11 nous présente une énumération `Finite` récursive utilisant `Maybe` afin de permettre l'obtention de constantes finies.

---

3. Par « évaluation stricte », nous entendons « évaluation non paresseuse ».

Les constantes d'énumérations récursives ne brisant pas les propriétés inhérentes de notre système d'énumérations, nous avons choisi d'accepter la définition d'énumérations récursives en Enumlang. Nous pensons que l'expressivité apportée par les énumérations récursives est bénéfique pour Enumlang, en permettant par exemple de définir des arbres ou des listes chaînées telle que celle présentée dans l'annexe C.3.

3.3.3 Constantes cycliques

Un objet cyclique est un objet se référençant lui-même directement ou indirectement. Autrement dit, un objet cyclique est un objet composite étant l'un de ses composants. Il existe différentes manières de créer ou modéliser des objets cycliques qui varient selon les langages et paradigmes de programmation.

Dans un langage impératif à évaluation stricte tel que Java (Arnold *et al.*, 2005), il est possible de créer des cycles par mutation comme dans l'exemple 3.12 : puisque les objets sont créés séquentiellement et qu'il n'est pas possible de référencer un objet pas encore construit, tous les objets du cycle doivent être construits avant que celui-ci ne puisse être fermé par assignation de références.

```
var parent = new Parent();
var child = new Child();
parent.child = child;
child.parent = parent;
```

FIGURE 3.12 – Exemple d'objets cycliques en Java

La présence ou l'absence de cycles dans un programme change la nature du graphe d'objets d'un programme. Dans un graphe acyclique, chaque objet est la racine d'un sous-graphe et est indépendant des objets qu'il compose. À l'inverse dans

un graphe cyclique, les objets inclus dans un cycle ne sont pas la racine d'un sous-graphe, et dépendent de tous les autres objets du cycle. [4]

Une constante étant un objet immutable, et Enumlang étant un langage à évaluation stricte, l'introduction de constantes cycliques demanderait l'implémentation de fonctionnalités permettant la création et l'analyse de ces cycles. Aussi, la présence de cycles dans le graphe d'objets pose des questions quant à la nature universelle des constantes et à leur comparaison. Nous avons donc choisi de ne pas permettre la définition de constantes cycliques en Enumlang.

3.3.4 Pureté évaluative

Les énumérations évaluatives présentées dans la section 3.2.4 déterminent l'appartenance de leurs constantes avec des prédicats. Une énumération représentant un ensemble fixe, et un prédicat pouvant ne pas être appelé en cas de mémoïsation éventuelle, il nous paraît préférable que le code d'un prédicat soit pur, c'est-à-dire déterministe et sans effets de bord :

— Un code déterministe est un code produisant toujours le même résultat pour des arguments donnés. Un prédicat n'ayant que sa constante receveuse comme argument, un prédicat déterministe retourne toujours le même statut d'acceptation pour une constante donnée.
— Un code sans effets de bord est un code ne modifiant pas son environnement, à l'intérieur ou à l'extérieur de son programme. L'appel d'un prédicat pouvant être imprévisible, la logique d'un programme ne devrait pas reposer sur d'éventuels effets de bord de ses prédicats.

---

4. Le typage propriétif, en anglais *ownership typing*, permet de définir une arborescence principale dans une structure cyclique, mais ne résout pas le problème fondamental de dépendance des objets (Clarke *et al.*, 1998).

La pureté d'un code pourrait être vérifiée statiquement en s'assurant qu'il ne contient pas d'opérations potentiellement impures. Cette tâche est récursive puisqu'elle demande de vérifier également la pureté de chaque fonction appelée directement ou indirectement par le code vérifié, en utilisant par exemple de l'inférence ou des annotations de pureté (Sălcianu et Rinard, 2005). Alternativement, la pureté d'un code pourrait être vérifiée dynamiquement lors de son exécution en utilisant par exemple un indicateur global d'évaluation pure.

Enumlang se voulant être un langage simple, et la vérification de la pureté d'un code étant selon nous un problème complexe, nous avons choisi de laisser la responsabilité de la pureté des prédicats au programmeur. Un prédicat se voulant généralement simple, nous estimons que l'écriture d'un prédicat involontairement non déterministe devrait être rare. De plus, permettre l'utilisation d'effets de bord à des fins de journalisation, d'optimisation ou de débogage nous paraît potentiellement bénéfique pour le programmeur.

3.4 Conclusion

Ce chapitre a présenté les énumérations introductrices d'Enumlang.

Une énumération introductrice est une énumération introduisant de nouvelles constantes dans un programme, et qui s'oppose donc aux énumérations non introductrices qui réutilisent des constantes provenant d'autres énumérations.

Il existe plusieurs structures d'énumérations introductrices en Enumlang :

— Une énumération primitive a une structure définie directement par le langage.
— Une énumération listée est composée d'un ensemble de constantes nommées.

— Une énumération à attributs est composée du produit d'autres énumérations, structuré par des attributs.
— Une énumération à prédicat est une énumération dont les constantes sont conditionnées à l'évaluation d'un code prédicat.

En plus d'être introductrice ou non, une énumération peut aussi être atomique ou composite, selon qu'elle soit composée ou non par d'autres énumérations, et statique ou évaluative, selon qu'elle dépende ou non de l'évaluation d'un prédicat. Finalement, une énumération peut être récursive, c'est-à-dire se composer elle-même, mais ne peut pas contenir de constante cyclique, c'est-à-dire qui s'autoréférence, ou de constante égale à **`null`**.

CHAPITRE IV

HÉRITAGE ÉNUMÉRATIF ET ÉVALUATIF

Ce chapitre présente notre application de l'héritage de classes aux énumérations. Cet héritage, que nous appelons héritage énumératif, permet de modéliser aisément les relations ensemblistes des énumérations dans un contexte d'objets.

Dans la première section, nous effectuons un rappel sur la définition de l'héritage de classes.

Dans les deuxième et troisième sections, nous abordons les propriétés ensemblistes de l'héritage classique et étudions comment celui-ci peut s'appliquer aux énumérations. Nous présentons ensuite les énumérations parentes d'Enumlang.

Dans les quatrième et cinquième sections, nous présentons l'héritage évaluatif, une approche de l'héritage pour les énumérations basée sur l'évaluation de prédicats. Nous présentons ensuite les énumérations enfantes d'Enumlang faisant usage de cet héritage.

Finalement dans la dernière section, nous décrivons le fonctionnement de notre interpréteur quant aux différentes fonctionnalités liées à l'héritage énumératif et évaluatif.

4.1 Héritage classique

En programmation orientée objet, et plus précisément dans un langage à classes, l'héritage est un mécanisme permettant de définir une classe comme étant une version plus spécifique d'une autre. La classe générale est alors qualifiée de classe *parente* et la classe plus spécifique de classe *enfante*. Une classe ne pouvant pas hériter d'elle-même, l'ensemble des classes du programme appartiennent à un graphe dirigé acyclique nommé la *hiérarchie de classes*. Cette hiérarchie peut éventuellement avoir pour racine une classe parente de toutes les autres telle que `Object` (Snyder, 1986; Taivalsaari, 1996).

L'héritage permet à une classe enfante, en plus de définir ses propres propriétés, de réutiliser celles de ses parents voire d'en redéfinir des versions plus spécifiques. Notamment, la redéfinition de méthodes permet d'appeler dynamiquement la version la plus spécifique d'une méthode en fonction de la classe d'un objet receveur, et l'extension de structure permet à une classe enfante de définir des attributs additionnels à ceux de ses parents.

Dans l'exemple 4.1, la classe `Student` hérite de la classe `Person`, définit un attribut additionnel `university`, et redéfinit la méthode héritée `get_greeting`.

Il existe différentes manières d'intégrer de l'héritage dans un langage de programmation. L'héritage de classes peut être simple, si une classe ne peut hériter que d'une seule autre classe, ou multiple, si une classe peut hériter d'un nombre arbitraire d'autres classes. Un objet appartient généralement à une seule classe la plus spécifique (Bertino et Guerrini, 1995), que nous appelons sa *classe propre*.

```
class Person {
    String name;

    String get_greeting() {
        return "Hello, I am " + this.name + " !";
    }
}

class Student extends Person {
    String university;

    @Override
    String get_greeting() {
        return "Hello, I am " + this.name + " and I study at "
           + this.university + " !";
    }
}
```

FIGURE 4.1 – Exemple d'héritage en Java

4.2 Héritage énumératif

4.2.1 Présentation

Dans une relation d'héritage, tout objet d'une classe enfante appartient également à ses classes parentes. Ainsi, l'héritage permet de décrire des relations ensemblistes, puisqu'une classe parente représente un sur-ensemble d'objets contenant tous les objets de ses classes enfantes.

L'héritage étant un concept central de la programmation orientée objet permettant de représenter des relations ensemblistes, et nos énumérations ayant pour objectif de représenter des ensembles dans un contexte de langage à objets, nous souhaitons concevoir un système d'héritage pour les énumérations. Nous appelons cet héritage l'*héritage énumératif*.

Étant donné que nous définissons des constantes dans les énumérations introductrices, nous souhaitons permettre la définition d'énumérations parentes et enfantes aux énumérations introductrices :

— Nous appelons *sur-énumérations* les énumérations parentes des énumérations introductrices définissant des sur-ensembles de ces dernières.
— Nous appelons *sous-énumérations* les énumérations enfantes des énumérations introductrices définissant des sous-ensembles de ces dernières.

Finalement, de la même manière qu'une instance peut appartenir à plusieurs classes instanciatives, dont une strictement plus spécifique, une constante peut appartenir à plusieurs énumérations. Nous appelons l'*appartenance* d'une constante sa propriété décrivant l'ensemble des énumérations auxquelles elle appartient.

4.2.2 Problèmes de l'héritage énumératif naïf

L'application naïve de l'héritage instanciatif, c'est-à-dire l'héritage des classes instanciatives, aux énumérations posent plusieurs problèmes qu'il nous convient d'identifier afin de concevoir un système d'héritage énumératif robuste.

Problème d'extension de structure

En héritage instanciatif, toute classe peut posséder des attributs additionnels à ceux hérités de ses parents. Dans le chapitre 3, nous considérons la structure des énumérations comme terminale afin de leur déterminer une cardinalité précise, ce qui rend les énumérations incompatibles avec l'extension de structure.

Dans l'exemple 4.2, l'énumération `A` possède un unique attribut booléen et a donc une cardinalité de 2. Sa sous-énumération `B` possède additionnellement un

```
enum A {
    var x: Boolean;
}

enum B {
    super A;

    var y: Boolean;
}
```

FIGURE 4.2 – Exemple d'erreur d'héritage énumératif par extension

deuxième attribut booléen et a donc une cardinalité de 4, ce qui est incompatible avec la relation de sous-ensemble que nous voulons définir par héritage.

Ainsi, l'héritage énumératif ne doit pas permettre l'ajout d'attributs dans des énumérations enfantes.

Problème de détermination d'appartenance

En héritage instanciatif, la classe d'instanciation d'une instance est également sa classe la plus spécifique. Afin de pouvoir définir des sous-énumérations, nous souhaitons qu'une constante puisse appartenir à des énumérations enfantes de son énumération d'obtention, ce qui est impossible si cette énumération est l'énumération la plus spécifique à laquelle appartient la constante.

Dans l'exemple 4.3, les constantes `a` et `b`, appartenant toutes les deux à l'énumération `A` et contiennent les mêmes données. Ces constantes étant donc égales, il serait illogique que seul `b` appartienne à l'énumération `B`.

Ainsi, l'héritage énumératif doit permettre à toute constante d'appartenir à des énumérations plus spécifiques que son énumération d'obtention.

```
enum A {
    var x: Integer;
    var y: Integer;
}

enum B {
    super A;
}

var a = A(0, 0);
var b = B(0, 0);
```

FIGURE 4.3 – Exemple d'erreur d'héritage énumératif par obtention

Problème d'énumérations disjointes

En héritage instanciatif, deux classes décrivent des ensembles joints s'il existe au moins une classe héritant simultanément de ces deux classes. Avec les énumérations, chaque énumération introductrice définit son propre ensemble indépendamment de la hiérarchie d'héritage. Les énumérations introductrices étant donc toujours disjointes, toute énumération héritant de plusieurs énumérations introductrices décrit un ensemble vide, ce qui nous paraît inutile.

Dans l'exemple 4.4, l'énumération `C` hérite des deux énumérations introductrices `A` et `B` et ne contient donc aucune constante.

Ainsi, l'héritage énumératif devrait interdire à une sous-énumération d'hériter de plusieurs énumérations introductrices pour éviter la définition d'énumérations vides.

```
enum A {
    case a;
}

enum B {
    case b;
}

enum C {
    super A;
    super B;
}
```

FIGURE 4.4 – Exemple d'erreur d'héritage énumératif par distinction

4.3 Sur-énumérations

Une sur-énumération est une énumération définie comme étant un sur-ensemble d'autres énumérations dont elle est parente. En représentant un sur-ensemble d'énumérations, une sur-énumération correspond à une union de ces énumérations. La cardinalité d'une sur-énumération est alors égale à la somme des cardinalités de ses composants, ce qui en fait un type somme (Pierce, 2002).

Une sur-énumération peut être héritée, directement ou à travers d'autres sur-énumérations, par des énumérations introductrices. Ainsi, les sur-énumérations se situent au sommet de la hiérarchie d'héritage des énumérations et ont des constantes à la structure potentiellement hétérogène.

En Enumlang, une sur-énumération est déclarée avec le mot-clé **supenum** (*super-enumeration*). Elle ne définit pas de structure puisqu'elle réutilise celles de ses composants, c'est-à-dire les énumérations qui en héritent avec le mot-clé **super**.

```
supenum Point {
    # ...
}

enum Point2d {
    super Point;

    var x: Integer;
    var y: Integer;
}

enum Point3d {
    super Point;

    var x: Integer;
    var y: Integer;
    var z: Integer;
}
```

FIGURE 4.5 – Exemple de sur-énumération

L'exemple 4.5 présente une sur-énumération `Point` représentant l'union de deux énumérations `Point2d` et `Point3d`.

Une énumération décrivant un ensemble fixe, l'ensemble des énumérations composant une sur-énumération doit être statiquement connu, la démarquant notamment d'une interface.

4.4 Héritage évaluatif

4.4.1 Présentation

En complément des sur-énumérations, nous souhaitons permettre la définition de sous-énumérations, c'est-à-dire d'énumérations héritant d'énumérations introductrices. Dans ce but, nous introduisons le concept d'*héritage évaluatif*, c'est-à-dire

d'héritage dans lequel une énumération peut hériter d'autres énumérations en fonction de prédicats.

L'héritage évaluatif se base sur le concept d'énumérations évaluatives introduit dans la section 3.2.4. Comme avec ces dernières, il permet de déterminer l'appartenance d'une constante à une sous-énumération en fonction de l'évaluation d'un prédicat, qui, en considérant qu'il est déterministe et qu'une constante est immutable, est invariable dans le temps.

4.4.2 Problèmes de l'héritage énumératif évaluatif

En plus des problèmes énoncés dans la section 4.2.2, la définition de sous-énumérations par le biais de l'évaluation de prédicats introduit de nouveaux problèmes qu'il nous convient de résoudre.

Problème de stratégie d'évaluation

Premièrement, nous devons déterminer une stratégie d'évaluation des prédicats permettant de déterminer l'appartenance d'une constante à certaines ou l'ensemble de ses énumérations potentielles.

Dans notre interpréteur, nous avons fait le choix de déterminer l'appartenance d'une constante de manière incrémentale et paresseuse, c'est-à-dire en évaluant des prédicats uniquement lorsque cela est nécessaire pour déterminer l'appartenance d'une constante à une énumération donnée.

Problème d'intersection propre

En permettant la définition de sous-énumérations sans restrictions, il est possible que certaines d'entre elles soient jointes, signifiant qu'il peut exister des constantes appartenant à plusieurs sous-énumérations d'une énumération donnée. Une constante n'a donc pas toujours une unique énumération la plus spécifique, mais appartient plutôt à un ensemble d'énumérations les plus spécifiques que nous appelons son *intersection propre*.

Par exemple, en considérant une sous-énumération `A` héritée par deux sous-énumérations jointes `B` et `C`, une constante appartenant à `A` peut avoir pour intersection propre {`A`}, {`B`}, {`C`} ou {`B`, `C`}. Nous remarquons qu'en considérant une énumération ayant n sous-énumérations jointes, le nombre d'intersections propres possibles est l'ensemble des combinaisons de ses sous-énumérations, qui est donc égal à 2^n intersections propres.

Les différentes énumérations d'une intersection propre ayant une spécificité égale, les versions des méthodes redéfinies dans ces énumérations ont également une spécificité égale. Ainsi, si une méthode est redéfinie dans plusieurs énumérations d'une intersection propre, cette intersection ne possède pas une unique version la plus spécifique de cette méthode. Pour résoudre ce problème, nous avons choisi de prioriser les redéfinitions de méthodes selon l'ordre de linéarisation de leurs énumérations.

4.5 Sous-énumérations

4.5.1 Déclaration

Une sous-énumération est une énumération définie comme étant un sous-ensemble d'autres énumérations dont elle hérite.

Une sous-énumération, ne peut hériter, directement ou indirectement, que d'une unique énumération introdutrice. À l'inverse, une énumération introdutrice ne peut pas hériter d'une sous-énumération. Les sous-énumérations se situent donc en bas de la hiérarchie d'héritage des énumérations et ont des constantes à la structure homogène.

Une sous-énumération est déclarée avec le mot-clé **`subenum`** (*sub-enumeration*) et hérite de ses énumérations parentes avec le mot-clé **`super`**.

```
enum Color {
    case red;
    case green;
    case blue;
}

subenum ColorBis {
    super Color;
}
```

FIGURE 4.6 – Exemple d'héritage évaluatif

Dans l'exemple 4.6, l'énumération `ColorBis` est une sous-énumération héritant de l'énumération `Color`. En l'absence de restrictions supplémentaires, `ColorBis` hérite de toutes les constantes de `Color` et représente donc en pratique le même ensemble.

4.5.2 Héritage par prédicat

Afin de n'hériter que de certaines constantes d'une énumération parente, une sous-énumération peut déclarer un prédicat similaire à ceux de la section 3.2.4. En présence d'un prédicat, une sous-énumération n'hérite que des constantes de son énumération parente acceptée par ce prédicat.

```
subenum MultipleOf3 {
    super Integer;

    predicate {
        return self % 3 == 0;
    }
}
```

FIGURE 4.7 – Exemple d'héritage par prédicat

Dans l'exemple 4.7, la sous-énumération `MultipleOf3` hérite de l'énumération `Integer` et possède un prédicat s'assurant que chaque entier hérité est divisible par 3. Cela n'étant pas le cas de chaque entier, la sous-énumération `MultipleOf3` représente un strict sous-ensemble de l'énumération `Integer`.

Dans un prédicat d'héritage, **`self`** n'est pas restreint comme dans une énumération à prédicat. Une sous-énumération par prédicat ne peut avoir qu'une seule énumération parente. Celle-ci sert de type pour **`self`** puisque l'appartenance étant évaluée des énumérations parentes aux énumérations enfantes, le prédicat ne peut être évalué que si cette constante appartient à l'énumération parente. À noter qu'un prédicat n'empêche pas l'héritage multiple dans les énumérations parentes ou enfantes de son énumération.

4.5.3 Héritage par partition

Additionnellement à l'usage d'un prédicat classique, une sous-énumération peut aussi hériter d'une énumération parente par le biais d'une partition. Une partition est un genre de prédicat permettant à une énumération d'attribuer chacune de ses constantes à une, ou aucune, énumération donnée parmi un ensemble de sous-énumérations.

Une partition est déclarée avec le mot-clé **`partition`** suivi du nom de la partition et de son corps. Dans ce corps, l'instruction **`select`** permet de sélectionner une sous-énumération et de la retourner. Similairement à une fonction classique, **`select`** peut ne sélectionner aucune énumération, ou la partition peut finir son évaluation sans exécuter de **`select`**, indiquant que la constante receveuse n'appartient à aucune des sous-énumérations de la partition.

Un héritage par partition est déclaré en spécifiant le nom de la partition après le parent dans le lien d'héritage déclaré avec **`super`**.

L'exemple 4.8 présente une partition `EvenOrOdd` déclarée dans l'énumération `Integer`. Cette partition détermine si chaque entier est pair ou impair et les assigne en conséquence aux sous-énumérations `Even` et `Odd`.

4.5.4 Héritage par intersection

Une sous-énumération peut hériter de plusieurs énumérations parentes. Lorsque cela est le cas, une constante doit appartenir à toutes les énumérations parentes, et être acceptées par les partitions et le prédicat éventuels de la sous-énumération afin d'appartenir à celle-ci.

```
enum Integer {
    # ...

    partition EvenOrOdd {
        if self % 2 == 0 {
            select Even;
        } else {
            select Odd;
        }
    }
}

subenum Even {
    super Integer.EvenOrOdd;
}

subenum Odd {
    super Integer.EvenOrOdd;
}
```

FIGURE 4.8 – Exemple d'héritage par partition

Dans l'exemple 4.9, la sous-énumération `MultipleOf6` hérite des sous-énumérations `MultipleOf2` et `MultipleOf3`, elles-mêmes sous-énumérations de `Integer`.

Une sous-énumération héritant toujours d'une unique énumération introductrice, une sous-énumération héritant de plusieurs énumérations parentes est un cas d'héritage en losange.

4.5.5 Obtention

L'obtention d'une constante d'une sous-énumération est similaire à l'obtention d'une constante d'une énumération à prédicat présentée dans la section 3.2.4. Ainsi, une sous-énumération possédant des attributs possède également une fonc-

```
subenum MultipleOf2 {
    super Integer;

    predicate {
        return self % 2 == 0;
    }
}

subenum MultipleOf3 {
    super Integer;

    predicate {
        return self % 3 == 0;
    }
}

subenum MultipleOf6 {
    super MultipleOf2;
    super MultipleOf3;
}
```

FIGURE 4.9 – Exemple d'héritage par intersection

tion d'obtention qui évalue l'appartenance des constantes obtenues à la sous-énumération et soulève une erreur si les arguments qui lui sont fournis sont incorrects.

L'exemple 4.10 présente deux obtentions de constantes de la sous-énumération `Origin`. La première constante est acceptée et la seconde est refusée et soulève une erreur lors de l'exécution.

```
enum Point {
    var x: Integer;
    var y: Integer;
}

subenum Origin {
    super Point;

    predicate {
        return self.x == 0 && self.y == 0;
    }
}

var origin = Origin(0, 0); # Constante acceptée
var incorrect = Origin(1, 1); # Constante refusée
```

FIGURE 4.10 – Exemple d'obtentions de constantes avec des sous-énumérations

4.6 Interprétation

4.6.1 Structure

En programmation orientée objet, un objet est généralement lié à une classe par le biais d'un méta-objet appelé sa *table virtuelle*. Dans l'interpréteur Enumlang, nous représentons une constante comme étant un méta-objet contenant une référence vers ses données et une référence vers son énumération, qui lui fait office de table virtuelle.

Les données d'une constante peuvent correspondre à une constante nommée, à un ensemble d'attributs ou à une valeur primitive. Dans certains cas, la valeur pourrait également être écrite directement à la place de la référence pour éviter des allocations et indirections inutiles.

Nous appelons l'énumération du méta-objet d'une constante son *énumération présente*, qui ne correspond pas à son intersection propre mais plutôt à son énumération la plus spécifique pour un usage donné. Cette énumération présente peut être changée au cours de l'exécution du programme, par mutation ou par copie du méta-objet de la constante, en fonction des besoins lors de coercitions de types ou d'appels de méthodes.

4.6.2 Coercitions d'énumérations

Les opérations de coercitions de types permettent de vérifier l'appartenance d'une constante à une énumération cible, voire de la convertir à cette énumération.

En l'absence d'héritage évaluatif, une coercition d'énumération est similaire à une coercition de type classique. Cependant, si l'énumération cible est une sous-énumération, il est nécessaire de vérifier l'appartenance de la constante à toutes les énumérations situées entre l'énumération présente de la constante et l'énumération cible dans la hiérarchie d'énumérations. Si la constante est acceptée par toutes les partitions et prédicats, alors la constante appartient à l'énumération cible.

Si la vérification d'énumération succède, les expressions **`isa`** et **`as`**, décrites dans la section 1.8.2, peuvent modifier la constante par mutation de son méta-objet afin de remplacer son énumération présente par l'énumération cible. L'expression **`as`** peut également effectuer cette modification par copie en retournant la constante copiée.

L'exemple 4.11 comporte deux coercitions de types :

1. L'expression **`isa`** vérifie l'appartenance de la constante `8` à l'énumération `MultipleOf2`, ce qui demande d'évaluer son prédicat.

```
subenum MultipleOf2 {
    super Integer;

    predicate {
        return self % 2 == 0;
    }

    fun get_rank(): Integer {
        return self / 2;
    }
}

var a = 8;
# Affiche "true"
print(a isa MultipleOf2);
# Affiche "4"
print((a as MultipleOf2).get_rank());
```

FIGURE 4.11 – Exemple de conversion dynamique de constante

2. L'expression **as** vérifie également cette appartenance, mais n'a pas à ré-exécuter le prédicat de `MultipleOf2` puisque l'opération précédente a pu changer l'énumération présente de la constante.

Nous pouvons par ailleurs noter que l'appel de la méthode `MultipleOf2.get_rank` requiert que l'énumération présente de la constante receveuse soit `MultipleOf2`, cette méthode n'étant pas présente dans l'énumération `Integer`.

4.6.3 Appels de méthodes

En programmation orientée objet, la table virtuelle de chaque classe contient généralement un pointeur vers chaque méthode la plus spécifique de cette classe (Stroustrup, 1988). L'énumération présente d'une constante n'étant pas toujours

son énumération propre, il peut être nécessaire de déterminer autrement la méthode la plus spécifique d'une constante lors d'un appel.

Comme énoncé dans la section 4.4.2, la méthode la plus spécifique d'une constante correspond à la méthode la plus spécifique définie dans la hiérarchie d'héritage énumératif et prioritaire selon la linéarisation d'Enumlang en cas de conflit. Afin de déterminer cette méthode, il peut être nécessaire de vérifier l'appartenance d'une constante aux sous-énumérations de son énumération présente dans lesquelles la méthode à appeler est redéfinie.

L'exemple 4.12 comporte trois appels de méthodes :

1. L'appel de méthode `a.get_description` vérifie si cette constante appartient à la sous-énumération `MultipleOf2`. Cela n'étant pas le cas, la méthode `Integer.get_description` est appelée.
2. L'appel de méthode `b.get_description` vérifie si cette constante appartient à la sous-énumération `MultipleOf2`. Cela étant le cas, l'appel vérifie ensuite si la constante appartient à la sous-énumération `MultipleOf4`. Cela n'étant pas le cas, la méthode `MultipleOf2.get_description` est appelée.
3. L'appel de méthode `c.get_description` vérifie si cette constante appartient à la sous-énumération `MultipleOf2`. Cela étant le cas, l'appel vérifie ensuite si la constante appartient à la sous-énumération `MultipleOf4`. Cela étant le cas, la méthode `MultipleOf4.get_description` est appelée.

Similairement aux coercitions de types, les appels de méthodes évaluatifs peuvent changer l'énumération présente d'une constante, permettant d'éviter la réévaluation de ses partitions et prédicats. Ainsi, bien que l'héritage évaluatif puisse pa-

raître inefficace au premier abord, nous estimons que les vérifications qu'il effectue auraient également été présentes en son absence, et qu'il apporte plutôt un gain d'expressivité au programme.

```
enum Integer {
    fun get_description(): String {
        return "1 * " + self;
    }
}

subenum MultipleOf2 {
    super Integer;

    predicate {
        return self % 2 == 0;
    }

    redef fun get_description(): String {
        return "2 * " + self / 2;
    }
}

subenum MultipleOf4 {
    super MultipleOf2;

    predicate {
        return self % 4 == 0;
    }

    redef fun get_description(): String {
        return "4 * " + self / 4;
    }
}

var a = 3;
var b = 6;
var c = 16;

# Affiche "1 * 3"
print(a.get_description());
# Affiche "2 * 3"
print(b.get_description());
# Affiche "4 * 4"
print(c.get_description());
```

FIGURE 4.12 – Exemple d'appel dynamique de méthode

4.7 Conclusion

Ce chapitre a présenté le système d'héritage énumératif d'Enumlang, une application de l'héritage de classes aux énumérations permettant de modéliser aisément leurs relations ensemblistes dans un contexte d'objets.

Les énumérations sont fondamentalement incompatibles avec l'héritage instanciatif. À sa place, l'héritage énumératif permet de définir des sur-énumérations par le biais d'unions et des sous-énumérations par le biais de prédicats et de partitions déterminant l'appartenance ou non de constantes à ces dernières.

En pratique, l'héritage évaluatif permet à une constante d'appartenir à plusieurs énumérations strictement plus spécifiques et pouvant être inconnues lors de son obtention. Enumlang détermine donc l'appartenance d'une constante paresseusement, en associant d'abord une constante à son énumération d'obtention, puis en affinant son appartenance au cours de l'exécution du programme notamment lors d'appels de méthodes et de conversions de types.

CHAPITRE V

FORMES ALTERNATIVES

La structure d'une énumération désigne la manière dont celle-ci arrange ses données pour permettre la représentation de constantes. En pratique, il peut exister plusieurs manières de représenter une même constante. Par exemple, un nombre premier peut être représenté avec son entier, tel que 7, ou avec son rang, qui pour 7 vaut 4.

Ce chapitre présente le système de formes alternatives d'Enumlang, un système permettant à une énumération d'avoir plusieurs structures équivalentes. Celui-ci permet de représenter de manière plus adaptée des constantes dans certains contextes, notamment en présence d'héritage énumératif, et introduit de la paresse en permettant le calcul de formes complexes uniquement lorsque cela est nécessaire.

Dans la première section, nous présentons les règles de déclaration des formes alternatives et des membres qui leurs sont associés en Enumlang.

Dans la seconde section, nous présentons les usages possibles des formes alternatives ainsi que leur implémentation dans l'interpréteur Enumlang.

5.1 Déclaration

5.1.1 Formes alternatives

En Enumlang, la structure décrite dans la déclaration d'une énumération correspond à sa forme principale. En plus de celle-ci, il est possible de déclarer des formes alternatives à une énumération avec d'autres déclarations de classes où chacune a un nom composé du nom de l'énumération puis du nom de la forme alternative.

Une forme alternative possède sa propre structure, composée d'attributs ou de constantes nommées, et ses propres méthodes. Elle n'est cependant pas une nouvelle énumération, et partage donc les liens d'héritage et les paramètres génériques de sa forme principale.

```
enum Boolean {
    # ...
}

enum Boolean.Integer {
    var integer: Integer;
}
```

FIGURE 5.1 – Exemple de déclaration de formes alternatives

L'exemple 5.1 présente une forme alternative `Boolean.Integer` représentant un booléen avec un nombre entier. Comme en C, nous pouvons supposer que 0 représente la constante **`false`** et que tous les autres entiers représentent la constante **`true`** (Kernighan et Ritchie, 1989). Nous utiliserons cet exemple tout au long de ce chapitre.

5.1.2 Convertisseurs de formes

Un convertisseur de forme est un membre d'une énumération permettant de convertir une constante d'une forme à une autre. Un convertisseur de forme est déclaré dans la classe de la forme source, principale ou alternative, avec le mot-clé **`as`** suivi du nom de la forme de destination et d'un corps. Au sein de ce corps, l'expression **`self`** fait référence à la constante receveuse sous sa forme lors de l'appel et l'instruction **`return`** permet de retourner cette constante sous sa nouvelle forme.

Chaque forme alternative doit déclarer un convertisseur vers sa forme principale, qui doit elle-même déclarer un convertisseur vers chacune de ses formes alternatives. La conversion directe entre formes alternatives n'est actuellement pas supportée.

L'exemple 5.2 présente des convertisseurs de formes entre les formes `Boolean` et `Boolean.Integer`.

5.1.3 Méthodes alternatives

Toutes les méthodes d'une même énumération sont partagées indépendamment des formes dans lesquelles celles-ci sont déclarées. Aussi, Enumlang permet de déclarer plusieurs versions d'une même méthode dans des formes différentes, à condition que leurs signatures soient identiques.

Dans l'exemple 5.3, la méthode `Boolean.get_inverse` est définit dans les deux formes `Boolean` et `Boolean.Integer`.

```
enum Boolean {
    as Integer {
        match self {
            case true {
                return Boolean.Integer(1);
            }
            case false {
                return Boolean.Integer(0);
            }
        }
    }
}

enum Boolean.Integer {
    as Boolean {
        if self.integer != 0 {
            return true;
        } else {
            return false;
        }
    }
}
```

FIGURE 5.2 – Exemple de déclaration de convertisseurs de formes

```
enum Boolean {
    fun get_inverse(): Boolean {
        return !self;
    }
}

enum Boolean.Integer {
    fun get_inverse(): Boolean {
        if self.integer != 0 {
            return Boolean.Integer(0);
        } else {
            return Boolean.Integer(1);
        }
    }
}
```

FIGURE 5.3 – Exemple de déclaration de méthodes alternatives

5.2 Utilisation et interprétation

5.2.1 Obtention

L'obtention d'une constante sous une forme alternative se fait de manière similaire à l'obtention d'une constante de toute énumération, notamment listée ou à attributs.

```
enum Boolean.YesNo {
    case yes;
    case no;

    # Convertisseurs
    # ...
}

var true1 = true;
var true2 = Boolean.Integer(1);
var true3 = Boolean.YesNo.yes;
```

FIGURE 5.4 – Exemple d'obtention de formes alternatives

Dans l'exemple 5.4, les trois variables `true1`, `true2` et `true3` contiennent la même constante. Celle-ci est cependant obtenue sous trois formes différentes `Boolean`, `Boolean.Integer` et `Boolean.YesNo`.

5.2.2 Typage

Typage statique

Bien qu'une énumération peut avoir plusieurs formes, celle-ci ne possède qu'un seul type. Le système de typage statique d'Enumlang fait donc abstraction de la forme des constantes, notamment lors d'assignations ou d'appels de fonctions ou de méthodes.

```
fun integer_to_boolean(integer: Integer): Boolean {
    return Boolean.Integer(integer);
}

var true1: Boolean = Boolean.Integer(1);
var true2: Boolean = integer_to_boolean(2);
```

FIGURE 5.5 – Exemple de typage statique avec des formes alternatives

Dans l'exemple 5.5, la variable `true1` de type `Boolean` se fait assigner une constante de la forme `Boolean.Integer`. Similairement, la fonction `integer_to_boolean` retourne statiquement des constantes de type `Boolean`, bien que leur forme lors de l'exécution soit `Boolean.Integer`

Typage dynamique

Les différentes formes d'une même énumération partagent également son type lors d'opérations de vérification dynamique du type d'une constante.

```
# Affiche « true »
print(true isa Boolean);
# Affiche « true »
print(Boolean.Integer(1) isa Boolean);
```

FIGURE 5.6 – Exemple de typage dynamique avec des formes alternatives

Dans l'exemple 5.6, plusieurs constantes représentant des booléens sont considérées comme appartenant à l'énumération `Boolean` indépendamment de leurs formes.

5.2.3 Comparaison

La comparaison de constantes d'Enumlang décrite dans la section 6.1 nécessite que les constantes comparées aient la même forme. Aussi, la forme principale d'une énumération étant considérée comme sa forme standard, c'est-à-dire où chaque représentation décrit une constante différente, Enumlang utilise les formes principales des constantes lors de leur comparaison.

```
# Affiche "true"
print(true == true);
# Affiche "true"
print(Boolean.Integer(1) == true);
# Affiche "true"
print(Boolean.Integer(1) == Boolean.Integer(2));
```

FIGURE 5.7 – Exemple de comparaisons avec formes énumératives

L'exemple 5.7 présente trois comparaisons de constantes :

— Dans la première comparaison, les deux constantes sont sous forme principale, et ne nécéssite donc pas de conversion de forme.
— Dans la deuxième comparaison, l'une des constantes est sous forme alternative, et doit donc être convertie sous forme principale pour être comparée.
— Dans la troisième comparaison, les deux constantes sont sous forme alternative, et sont donc converties sous forme principale pour être comparées.

5.2.4 Appel de méthodes

Les formes d'une énumération partageant son type, toute fonction ou méthode prenant une constante en paramètre doit l'accepter sous toutes ses formes. Le

receveur **self** d'une méthode est lui de la forme dans laquelle la méthode est déclarée, demandant possiblement une conversion de forme à l'exécution.

Afin de convertir efficacement les constantes lors d'appels de méthodes, chaque forme énumérative possède son propre méta-objet représentant sa table virtuelle dans l'interpréteur Enumlang. Au sein de celui-ci, les conversions du receveur nécessaires à l'appel de chaque méthode sont statiquement connues.

```
enum Boolean.Integer {
      fun get_integer(): Integer {
          return self.integer;
    }
}

var true1 = true;
var true2 = Boolean.Integer(2);
var one = true1.get_integer();
var two = true2.get_integer();
```

FIGURE 5.8 – Exemple d'appels de méthodes avec formes énumératives

L'exemple 5.8 présente deux appels de la méthode `Boolean.Integer.get_integer` :

— Lors du premier appel, la constante `true1` est sous la forme `Boolean` et doit donc être convertie sous la forme `Boolean.Integer`.

— Lors du second appel, la constante `true2` est sous la forme `Boolean.Integer` et n'a donc pas besoin d'être convertie.

5.2.5 Conversion

Comme énoncé dans les sections 5.2.3 et 5.2.4, les comparaisons et appels de méthodes sur des constantes peuvent nécessiter des conversions de formes, qui se font automatiquement en appelant les convertisseurs de formes adaptés.

Un convertisseur de forme peut convertir une constante de sa forme principale à une forme alternative ou inversement. Sans possibilité de conversion directe, la conversion d'une constante entre deux formes alternatives peut se faire en deux étapes en passant par sa forme principale.

Le système de typage d'Enumlang ne permettant pas de connaître statiquement la forme d'une constante, il n'est pas possible de vérifier statiquement que la forme retournée par un convertisseur est correcte. Cependant, une vérification dynamique est effectuée à chaque appel d'un convertisseur afin de s'assurer de la validité de la forme retournée.

Les constantes étant immutables, les formes alternatives d'une constante restent valides dans le temps, rendant possible leur mémoïsation. Ainsi, dans l'interpréteur Enumlang, chaque forme d'une constante possède des références vers ses autres formes, qui sont initialisées lors d'une première conversion et réutilisées pour chaque conversion successive.

```
var a = true;
a.get_integer();
a.get_integer();
```

FIGURE 5.9 – Exemple de conversions avec formes énumératives

Dans l'exemple 5.9, la constante de la variable `a` est obtenue sous sa forme `Boolean` avant d'être utilisée deux fois en temps que receveuse de la méthode

`Boolean.Integer.get_integer`. Lors du premier appel, la constante est convertie sous sa forme `Boolean.Integer`, appelant le convertisseur de cette forme et mémoïsant cette dernière. Lors du second appel, la forme mémoïsée est utilisée sans qu'un convertisseur ne soit appelé.

5.3 Conclusion

Ce chapitre a présenté le système de formes alternatives d'Enumlang, une fontionnalité de ce langage permettant de mieux représenter une constante en fonction du contexte ou d'introduire de la paresse dans un programme.

Chaque énumération possède une forme principale, et éventuellement une ou plusieurs formes alternatives déclarées dans leurs propres classes énumératives. Toutes les formes d'une énumération partagent le même type, qui fait donc abstraction de celles-ci, et peuvent définir voire redéfinir leurs propres structures et méthodes.

La conversion d'une constante d'une forme à une autre se fait paresseusement et automatiquement lorsque cela est nécessaire par le biais de l'évaluation de convertisseurs de formes. Chaque constante mémoïse l'ensemble de ses formes connues afin d'éviter l'exécution répétée de convertisseurs.

CHAPITRE VI

UTILISATION DES CONSTANTES

En tant qu'objets universels, les constantes possèdent différentes propriétés telles que l'immutabilité ou l'absence d'identité. Enumlang utilise ces propriétés dans diverses déclarations, instructions et expressions destinées à rendre l'utilisation des constantes plus ergonomique.

Ce chapitre présente quelques fonctionnalités d'Enumlang liées à l'utilisation des constantes, dont la comparaison de constantes, la déclaration de constantes globales, et une instruction d'aiguillage énumératif.

6.1 Comparaison de constantes

En Enumlang, les opérateurs `==` et `!=` permettent de comparer deux objets entre eux suivant un mode de comparaison adapté au genre des objets comparés :

- Les instances étant définies par leur identité, elles sont comparées *par référence*.
- Les constantes étant définies par leurs données, elles sont comparées *par valeur*.

Enumlang ne permet actuellement pas de redéfinir la comparaison de constantes, et utilise plutôt un algorithme de comparaison dépendant de la structure des constantes comparées :

— Constantes primitives : Les constantes primitives sont comparées par une fonction interne à l'interpréteur.
— Constantes nommées : Deux constantes nommées sont égales si elles partagent la même énumération et le même nom.
— Constantes à attributs : Deux constantes à attributs sont égales si tous leurs attributs sont égaux, ce qui est donc une opération récursive.

```
enum Power {
    case on;
    case off;
}

enum Point {
    var x: Integer;
    var y: Integer;
}

print(Power.on == Power.on); # Affiche "true"
print(Power.on == Power.off); # Affiche "false"
print(0 == 0); # Affiche "true"
print(0 == 1); # Affiche "false"
print(Point(0, 0) == Point(0, 0)); # Affiche "true"
print(Point(0, 0) == Point(1, 1)); # Affiche "false"
```

FIGURE 6.1 – Exemple de comparaisons de constantes

L'exemple 6.1 présente diverses comparaisons de constantes.

Similairement aux instances, si deux constantes sont égales alors elles désignent une unique constante. Par ailleurs, l'appartenance d'une constante étant déter-

minée incrémentalement pendant l'exécution du programme, la comparaison de constantes fait abstraction de leur énumération présente lors de la comparaison.

```
# Affiche "true"
print((3 as Integer) == (3 as Prime));
```

FIGURE 6.2 – Exemple de comparaisons de constantes d'énumérations différentes

L'exemple 6.2 présente une comparaison montrant que la constante `3` est égale à elle-même indépendamment de son énumération présente.

Selon l'implémentation des paramètres génériques d'Enumlang décrite dans la section 3.2.5, les arguments génériques sont considérés comme faisant partie d'un objet. Ainsi, bien que la comparaison fasse abstraction de l'énumération propre des constantes, elle reste sensible à leurs arguments génériques.

```
# Affiche "true"
print(Sequence[Odd](1, 3, 5) == Sequence[Odd](1, 3, 5));
# Affiche "false"
print(Sequence[Odd](1, 3, 5) == Sequence[Integer](1, 3, 5));
```

FIGURE 6.3 – Exemple de comparaisons de constantes d'énumérations paramétriques

Dans l'exemple 6.3, deux constantes de type `Sequence[Odd]` et `Sequence[Integer]` sont considérées comme différentes malgré la similarité de leurs composants car leurs arguments génériques sont différents.

6.2 Constantes globales

En Enumlang, une constante globale est une constante déclarée par le biais d'une déclaration statique et visible dans l'ensemble du programme. Cette constante est obtenue par l'évaluation statique d'une expression et ne peut pas être réassignée.

Une constante globale est déclarée avec le mot-clé **`const`** suivi de son nom, de son type, de l'opérateur d'assignement = et d'une expression.

```
fun square(i: Integer): Integer {
    return i * i;
}

const area: Integer = square(side);
const side: Integer = 5;
```

FIGURE 6.4 – Exemple de constantes globales

L'exemple 6.4 présente deux constantes globales `area` et `side`. La première dépend de la deuxième, qui est pourtant déclarée après, et de la fonction `square`.

L'évaluation des constantes globales est la dernière étape de la modélisation d'un programme. Cette évaluation est similaire à l'évaluation dynamique d'un programme mais soulève une erreur en cas d'accès aux entrées et sorties du programme ou si une constante globale dépend directement ou indirectement d'elle-même.

6.3 Aiguillage énumératif

Enumlang possède une instruction d'aiguillage énumératif permettant d'exécuter un bloc d'instructions donné en fonction de la comparaison d'une constante donnée à d'autres constantes.

Un aiguillage énumératif est déclaré avec le mot-clé **match** suivi d'une expression, d'un ensemble de cas, et éventuellement d'un cas par défaut. Un cas est déclaré avec le mot-clé **case** suivi d'une expression et d'un bloc, et un cas par défaut est déclaré avec le mot-clé **default** suivi d'un bloc.

```
match get_random_integer() {
    case 0 {
        print("The number is zero.");
    }
    case 1 {
        print("The number is one.");
    }
    default {
        print("The number is neither zero nor one.");
    }
}
```

FIGURE 6.5 – Exemple d'aiguillage énumératif

L'exemple 6.5 présente un aiguillage énumératif possédant deux cas avec constantes et un cas par défaut.

Dans un aiguillage énumératif, la constante à comparer est évaluée dynamiquement, et les constantes des cas sont évaluées statiquement. Lors de son exécution, la constante dynamique est comparée aux constantes de chacun des cas, si elle est égale à l'une d'entre elles, alors le bloc du cas correspondant est exécuté, sinon, le bloc par défaut est exécuté.

Bien qu'un aiguillage énumératif puisse être reproduit avec une chaîne de **if**, il s'en différencie grâce une syntaxe plus concise et de plus nombreuses garanties statiques et dynamiques :

— Un aiguillage garantit statiquement l'absence de collisions de cas en comparant statiquement les constantes de ces derniers entre elles. Cela rend

notamment le résultat de l'exécution d'un aiguillage indépendant de l'ordre de déclaration de ses cas.

— Si la cardinalité de l'énumération de la constante comparée est connue, un aiguillage garantit statiquement son exhaustivité en vérifiant que sa quantité de cas est égale à la cardinalité de l'énumération, ou qu'il possède sinon un cas par défaut.

— Si la cardinalité de l'énumération de la constante comparée est inconnue, un aiguillage garantit dynamiquement l'exécution d'un cas en soulevant une erreur si une constante comparée ne correspond à aucun de ses cas.

6.4 Conclusion

Ce chapitre a présenté quelques fonctionnalités d'Enumlang permettant une utilisation ergonomique des constantes.

La comparaison de constantes est une comparaison par valeur, elle ne peut pas être redéfinie et utilise un algorithme de comparaison variant selon l'énumération introductrice des constantes comparées.

Une constante globale est une constante déclarée dans un module et correspondant à une expression évaluée statiquement. Elle ne peut pas être réassignée ou dépendre d'elle-même.

Un aiguillage énumératif est une instruction permettant de sélectionner dynamiquement un bloc d'instructions à exécuter en fonction d'une constante. Il apporte des garanties statiques et dynamiques supplémentaires par rapport à une chaîne de **`if`** notamment en terme d'exhaustivité et d'absence de collisions des cas possibles.

Ces fonctionnalités ergonomiques complémentent et s'appuient sur les mécanismes centraux d'Enumlang que sont l'héritage énumératif et les énumérations évaluatives.

CHAPITRE VII

EXEMPLE : SYSTÈME NUMÉRIQUE

Ce chapitre présente un exemple d'utilisation des énumérations en Enumlang sous forme d'un programme de modélisation de nombres rationnels. Ce programme utilise diverses fonctionnalités des énumérations telles que les prédicats, l'héritage évaluatif et les formes alternatives afin de mieux modéliser les nombres rationnels.

Ce chapitre est structuré de manière incrémentale. Chaque section présente une partie du code du programme d'exemple et explique comment celui-ci utilise les fonctionnalités des énumérations d'Enumlang. Le code complet du programme d'exemple est disponible dans l'annexe D.

7.1 Énumération des nombres rationnels

Notre premier objectif est de définir une énumération permettant de décrire l'ensemble des nombres rationnels. Un rationnel peut s'exprimer comme étant le quotient de deux nombres entiers.

Dans le code 7.1, nous définissons donc une énumération `Rational` ayant deux attributs `num` et `denom` correspondant au numérateur et au dénominateur du rationnel. Le type des attributs, `Integer`, est le type primitif des nombres entiers d'Enumlang.

```
enum Rational {
    var num: Integer;
    var denom: Integer;
}
```

FIGURE 7.1 – Structure de l'énumération des rationnels

7.2 Prédicat de validation des constantes

Un nombre rationnel possède de nombreuses propriétés. Notamment, le dénominateur d'un rationnel ne peut pas être nul, et tout rationnel possède une forme canonique dans laquelle son dénominateur est strictement positif, et ses numérateur et dénominateur sont premiers entre eux.

Nous choisissons de représenter les rationnels sous forme canonique. Dans le code 7.2, nous utilisons un prédicat afin de nous assurer de la validité des constantes de notre énumération. Dans ce prédicat, nous utilisons notamment la fonction `gcd` pour déterminer le plus grand diviseur commun de deux entiers.

```
enum Rational {
    predicate {
        return self.denom > 0 && gcd(self.num, self.denom) ==
           1;
    }
}

fun gcd(a: Integer, b: Integer): Integer {
    if b == 0 {
        return a;
    } else {
        return gcd(b, a % b);
    }
}
```

FIGURE 7.2 – Prédicat de l'énumération des rationnels

7.3 Méthodes des nombres rationnels

Il existe une multitude d'opérations possibles avec les nombres rationnels. Nous implémentons certaines d'entre-elles grâce à des méthodes dans le code 7.3 :

- La méthode `Rational.to_string` retourne une chaîne de caractères décrivant le rationnel.
- La méthode `Rational.get_num` retourne le numérateur du rationnel.
- La méthode `Rational.get_denom` retourne le dénominateur du rationnel.
- La méthode `Rational.get_inverse` retourne l'inverse du rationnel. [1]

1. La fonction `rational` utilisée dans la méthode `get_inverse` est définie dans la section 7.4.

```
enum Rational {
    redef fun to_string(): String {
        return "" + self.num + "/" + self.denom;
    }

    fun get_num(): Integer {
        return self.num;
    }

    fun get_denom(): Integer {
        return self.denom;
    }

    fun get_inverse(): Rational {
        return rational(self.denom, self.num);
    }
}
```

FIGURE 7.3 – Méthodes des rationnels

7.4 Forme alternative en fraction

La simplification d'un rationnel nécessaire à la détermination de sa forme canonique peut avoir un coup significatif. Afin d'éviter ce calcul lorsque cela est possible, nous définissons une forme alternative non canonique des rationnels.

Dans les codes 7.4 et 7.5, nous définissons la forme alternative `Rational.Fraction` permettant de représenter les rationnels sans les simplifier. Similairement à sa forme principale, `Rational.Fraction` possède un prédicat s'assurant que le déminateur de ses constantes ne soit pas nul, ainsi que quelques méthodes utilitaires. La fonction `rational` nous sert de raccourci pour obtenir des rationnels sous forme de fraction.

```
enum Rational {
    as Fraction {
        return Rational.Fraction(self.num, self.denom);
    }
}

fun rational(num: Integer, denom: Integer): Rational {
    return Rational.Fraction(num, denom);
}
```

FIGURE 7.4 – Forme alternative des rationnels en fraction 1

```
enum Rational.Fraction {
    var num: Integer;
    var denom: Integer;

    predicate {
        return self.denom != 0;
    }

    as Rational {
        var num = self.num;
        var denom = self.denom;
        if denom < 0 {
            num = -num;
            denom = -denom;
        }

        var divisor = gcd(num, denom);
        return Rational(num / divisor, denom / divisor);
    }

    fun get_inverse(): Rational {
        return rational(self.num, self.denom);
    }

    fun get_opposite(): Rational {
        return rational(-self.num, self.denom);
    }
}
```

FIGURE 7.5 – Forme alternative des rationnels en fraction 2

7.5 Opérations des nombres rationnels

Afin de permettre la réalisation de calculs arithmétiques avec nos nombres rationnels, nous définissons dans le code 7.6 des méhodes correspondant aux opérations d'addition, de soustraction, de multiplication et de division sur l'énumération `Rationnal`. Les calculs effectués pouvant retourner un rationnel non simplifié, nous utilisons la forme `Rational.Fraction` pour retourner nos résultats.

Nous pouvons noter qu'il est également possible de définir ces calculs dans la forme `Rational.Fraction` des rationnels. Sans simplification, ceux-ci pourraient cependant faire grandir indéfiniment les entiers primitifs utilisés pour représenter les rationnels.

```
enum Rational {
    fun add(other: Rational): Rational {
        return rational(self.num * other.denom + self.denom *
           other.num, self.denom * other.denom);
    }

    fun sub(other: Rational): Rational {
        return self.add(other.get_opposite());
    }

    fun mul(other: Rational): Rational {
        return rational(self.num * other.num, self.denom *
           other.denom);
    }

    fun div(other: Rational): Rational {
        return self.mul(other.get_inverse());
    }
}
```

FIGURE 7.6 – Opérations arithmétiques des rationnels

7.6 Sous-énumération des nombres entiers

En plus de notre énumération des nombres rationnels, nous souhaitons définir une énumération des nombres entiers, définie comme étant un sous-ensemble des rationnels.

Dans le code 7.7, nous définissons donc une sous-énumération `RationalInteger` héritant de `Rational` avec un prédicat. Nos rationnels étant sous forme canonique, tout rationnel dont le dénominateur est égal à un est un entier.

```
subenum RationalInteger {
    super Rational;

    predicate {
        return self.denom == 1;
    }
}
```

FIGURE 7.7 – Prédicat de la sous-énumération des entiers

7.7 Forme simplifiée des nombres entiers

Notre énumération `RationalInteger` utilise la même structure que `Rational` pour représenter les entiers, c'est-à-dire deux entiers primitifs en tant que numérateur et dénominateur. Cette représentation paraît inadaptée à notre nouvelle énumération puisqu'elle utilise deux entiers pour n'en représenter qu'un seul.

Dans le code 7.8, nous définissons donc une forme alternative `RationalInteger.Simple` utilisant un seul entier primitif afin de représenter un nombre entier. La fonction `integer` nous sert de raccourci pour obtenir des entiers sous forme simplifiée.

```
subenum RationalInteger {
    as Simple {
        return RationalInteger.Simple(self.num);
    }
}

subenum RationalInteger.Simple {
    var value: Integer;

    as RationalInteger {
        return RationalInteger(self.value, 1);
    }
}

fun integer(value: Integer): Integer {
    return RationalInteger.Simple(value);
}
```

FIGURE 7.8 – Forme alternative simplifiée des entiers

7.8 Méthodes des nombres entiers

Nous redéfinissons les méthodes utilitaires de `Rational` et `RationalInteger` dans la forme `RationalInteger.Simple` afin d'éviter de devoir convertir nos entiers lorsque cela est possible.

```
subenum RationalInteger {
    fun get_value(): Integer {
        return self.num;
    }
}

subenum RationalInteger.Simple {
    fun get_value(): Integer {
        return self.value;
    }

    redef fun to_string(): String {
        return self.value.to_string();
    }

    redef fun get_num(): Integer {
        return self.value;
    }

    redef fun get_denom(): Integer {
        return 1;
    }

    redef fun get_inverse(): Rational {
        return rational(1, self.value);
    }

    redef fun get_opposite(): Rational {
        return integer(-self.value);
    }
}
```

FIGURE 7.9 – Méthodes des entiers

7.9 Opérations des nombres entiers

Les calculs arithmétiques sont généralement plus simples à effectuer sur des nombres entiers que sur des rationnels, ces derniers pouvant notamment demander de mettre les nombres au même dénominateur voire de simplifier le résultat pour retrouver sa forme canonique. Nous souhaitons donc mettre en place un système privilégiant les opérations arithmétiques simplifiées lorsque celles-ci ont des entiers comme opérandes.

Dans les codes 7.10 et 7.11, nous définissons des méthodes `Rational.add_rational`, `Rational.mul_rational`, `Rational.add_integer` et `Rational.mul_integer`, que nous redéfinissons aussi dans `RationalInteger`. Afin que la méthode la plus spécifique soit appelée en fonction des opérandes de chaque calcul, nous mettons en place un système de *double dispatch*[2] en réécrivant les méthodes `Rational.add` et `Rational.mul`.

		a	
		Rational	Integer
b	Rational	Rational.add_rational	Rational.add_integer
	Integer	Integer.add_rational	Integer.add_integer

TABLEAU 7.1 – Tableau de sélection de méthode lors du calcul `a.add(b)`

Le tableau 7.1 décrit la méthode spécifique appelée par l'appel `a.add(b)` en fonction des types des constantes `a` et `b`. Dans la méthode spécifique, le receveur et le paramètre sont inversés par rapport au premier appel.

2. Le système de *double dispatch* mis en place n'est pas géré par le langage Enumlang. Il s'agit d'un enchaînement manuel d'appels utilisant des receveurs différents afin d'appeler les versions les plus spécifiques de certaines méthodes.

```
subenum Rational {
    fun add(other: Rational): Rational {
        return other.add_rational(self);
    }

    fun mul(other: Rational): Rational {
        return other.mul_rational(self);
    }

    fun add_rational(other: Rational): Rational {
        return rational(other.num * self.denom + self.num *
           other.denom, other.denom * self.denom);
    }

    fun mul_rational(other: Rational): Rational {
        return rational(other.num * self.num, other.denom *
           self.denom);
    }

    fun add_integer(other: RationalInteger): Rational {
        return rational(other.get_value() * self.denom + self.
           num, self.denom);
    }

    fun mul_integer(other: RationalInteger): Rational {
        return rational(other.get_value() * self.num, self.
           denom);
    }
}
```

FIGURE 7.10 – Opérations des entiers 1

```
subenum RationalInteger.Simple {
    redef fun add(other: Rational): Rational {
        return other.add_integer(self);
    }

    redef fun mul(other: Rational): Rational {
        return other.mul_integer(self);
    }

    redef fun add_rational(other: Rational): Rational {
        return rational(other.num + self.value * other.denom,
           other.denom);
    }

    redef fun mul_rational(other: Rational): Rational {
        return rational(other.num * self.value, other.denom);
    }

    redef fun add_integer(other: RationalInteger):
       RationalInteger {
        return integer(other.get_value() + self.value);
    }

    redef fun mul_integer(other: RationalInteger):
       RationalInteger {
        return integer(other.get_value() * self.value);
    }
}
```

FIGURE 7.11 – Opérations des entiers 2

7.10 Utilisation des nombres

Le code 7.12 présente un exemple d'utilisation de notre système numérique basé sur les énumérations.

Premièrement, nous déclarons les variables `a`, `b` et `c` :

- `a` est un rationnel déclaré comme étant un rationnel.
- `b` est un entier déclaré comme étant un rationnel.
- `c` est un entier déclaré comme étant un entier.

Les rationnels étant déclarés avec la fonction `rational`, qui retourne des rationnels sous la forme `Rational.Fraction`, ceux-ci ne sont pas simplifiés tant que cela n'est pas nécéssaire.

Nous affichons ensuite ces trois nombres. La méthode `to_string` étant redéfinie dans une forme de la sous-énumération `RationalInteger`, l'appartenance de `a` et `b` à cette énumération est évaluée. Bien que déclaré comme étant un rationnel, `b` se comporte alors comme un entier.

Nous effectuons finalement des calculs avec ces trois nombres. Les deux premiers calculs impliquant `a` utilisent les méthodes arithmétiques des rationnels. Toutefois, `b` et `c` étant des entiers, le dernier calcul utilise les méthodes simplifiées des entiers, et retourne un résultat également sous forme d'entier.

```
var a = rational(3, 2);
var b = rational(6, 3);
var c = integer(5);

# Affiche "3/2"
print(a);
# Affiche "2"
print(b);
# Affiche "5"
print(c);

# Affiche "7/2"
print(a.add(b));
# Affiche "-7/2"
print(a.sub(c));
# Affiche "10"
print(b.mul(c));
```

FIGURE 7.12 – Exemple d'utilisation des nombres

7.11 Conclusion

Dans ce chapitre, nous avons présenté un programme de nombres rationnels en Enumlang. Afin de mieux modéliser et manipuler les nombres rationnels, ce programme utilise les fonctionnalités des énumérations suivantes :

- L'énumération à attributs `Rational` est utilisée pour représenter les rationnels avec un numérateur et un dénominateur.
- L'énumération `Rational` possède un prédicat afin de s'assurer que ses rationnels sont valides et sous forme canonique.
- La forme alternative `Rational.Fraction` permet d'éviter la simplification des rationnels lorsque cela est possible.

— La sous-énumération `RationalInteger`, qui hérite de `Rational` grâce à l'héritage évaluatif, permet de représenter les entiers comme étant un sous-ensemble des rationnels.

— La forme alternative `RationalInteger.Simple` permet d'optimiser les entiers grâce à une représentation et des méthodes simplifiées.

Le code complet du programme d'exemple est disponible dans l'annexe D.

CHAPITRE VIII

TRAVAUX CONNEXES

Ce chapitre présente divers articles de recherche et concepts de langages de programmation présentant des similarités ou nous ayant inspirés lors de nos recherches sur les énumérations. Nos recherches se concentrant sur la combinaison de plusieurs fonctionnalités, nous préférons comparer chaque travail non pas à l'ensemble de nos travaux mais seulement aux fonctionnalités qui s'en rapprochent et qui nous paraissent les plus pertinentes.

Dans les premières sections, nous abordons les énumérations « classiques », les valeurs et les objets immutables. Ensuite, nous présentons quelques systèmes permettant de garantir statiquement certaines propriétés des énumérations. Finalement, nous présentons divers travaux liés à l'héritage énumératif, aux formes alternatives et à la description des énumérations.

8.1 Énumérations classiques

8.1.1 C

En C, une énumération est déclarée avec le mot-clé **`enum`** et permet de définir un ensemble de constantes nommées. Chaque constante est ensuite substituée par un entier, implicite ou explicite, et peut être utilisée en tant que tel dans le code

(Kernighan et Ritchie, 1989), similairement à *#define* qui peut être utilisé dans ce même but.

```
enum Color {
    RED,
    GREEN,
    BLUE
};
```

FIGURE 8.1 – Exemple d'énumération en C

L'exemple 8.1 présente une énumération `Color` composée des constantes `RED`, `GREEN` et `BLUE` qui correspondent respectivement aux entiers `0`, `1` et `2`.

Les énumérations de C permettent de définir un type (ou plutôt alias de type) et des constantes nommées comme nos énumérations listées. Cependant, C n'étant ni un langage à objets, ni un langage développant particulièrement le concept d'énumérations, nous estimons qu'il n'y a ici que peu de parallèles à faire avec Enumlang.

8.1.2 Java

En Java, depuis Java 5, une énumération est déclarée avec le mot-clé **`enum`** et permet dé définir une classe accompagnée d'un ensemble fixe d'instances. Chaque énumération hérite de la classe abstraite `java.lang.Enum`, peut implémenter des interfaces et définir ses propres méthodes voire attributs (Arnold *et al.*, 2005).

L'exemple 8.2 présente une énumération `Color` contenant les constantes `BLUE`, `BROWN` et `PURPLE`.

L'exemple 8.3 présente une énumération `Color` ayant un attribut `abbr` de type `String` et contenant les constantes `BLUE`, `BROWN` et `PURPLE`. Les constantes Java

```
enum Color {
    RED, GREEN, BLUE;
}
```

FIGURE 8.2 – Exemple d'énumération en Java

```
enum Color {
    BLUE   ("B"),
    BROWN  ("B"),
    PURPLE ("P");

    final String abbr;

    private Color(String abbr) {
        this.abbr = abbr;
    }
}
```

FIGURE 8.3 – Exemple d'énumération avec attributs en Java

étant des objets standards, les constantes `BLUE` et `BROWN` sont considérées comme différentes par l'opérateur == et par l'implémentation par défaut de la méthode `Object.equals` indépendamment de leur contenu.

Ainsi, comme en Enumlang, chaque énumération Java est une classe définissant un ensemble fixe de constante étant des objets. Cependant, Java ne distingue pas les instances des constantes, faisant de ces dernières des objets mutables par défaut généralement comparés par référence et ayant donc une sémantique assez différente d'Enumlang.

8.1.3 Scala

En Scala, une énumération est une classe déclarée avec le mot-clé **`enum`** et dans laquelle le mot-clé **`case`** permet de définir des constantes. Similairement au Java, les énumérations de Scala implémentent le trait `scala.reflect.Enum`, et peuvent définir des attributs (Odersky *et al.*, 2004).

```
enum Color:
  case Red, Green, Blue
```

FIGURE 8.4 – Exemple d'énumération en Scala

L'exemple 8.4 présente une énumération `Color` contenant les constantes `Red`, `Green` et `Blue`.

Les énumérations de Scala sont très similaires à celles de Java et peuvent déclarer des attributs avec la même sémantique. Ainsi, les commentaires que nous avons formulées sur les énumérations de Java s'appliquent aussi aux énumérations de Scala.

8.2 Valeurs fonctionnelles

8.2.1 Haskell

Haskell est un langage de programmation fonctionnel permettant de manipuler des valeurs immutables et comparées par valeur (Marlow *et al.*, 2010). Celles-ci sont structurées par des types algébriques, c'est-à-dire des types sommes et produits (Pierce, 2002).

L'exemple 8.5, qui est transcrit en Enumlang dans la figure 8.6, définit deux types algébriques :

```
data Color = Red | Green | Blue

data Tree a = Empty | Node a (Tree a) (Tree a)
```

FIGURE 8.5 – Exemples de types algébriques en Haskell

— Le type `Color` est défini comme étant la somme des constructeurs de données `Red`, `Green` et `Blue`, qui sont unitaires. Il agit donc comme une somme de constantes, correspondant aux énumérations classiques (section 8.1) ou à nos énumérations listées (section 3.2.1).

— Le type `Tree a` est définit comme étant la somme des constructeurs de données `Empty` et `Node`, ce dernier étant lui-même le produit du type `a` et de deux arbres `Tree a`. En Enumlang, la somme correspond à une sur-énumération (section 4.3) et le produit à une énumération à attributs (section 3.2.3).

```
enum Color {
    case red; case green; case blue;
}

supenum Tree[T] {}

enum Empty[T] {
    super Tree[T];
    case empty;
}

enum Node[T] {
    super Tree[T];
    var value: T;
    var left: Tree[T];
    var right: Tree[T];
}
```

FIGURE 8.6 – Transcription de l'exemple 8.5 en Enumlang

Il est aussi intéressant de noter que la stratégie d'évaluation paresseuse d'Haskell permet la création de valeurs cycliques (Bird, 2012), qui pourraient correspondre à des constantes cycliques absentes en Enumlang.

```
cycle = 0:cycle
```

FIGURE 8.7 – Exemple de valeur cyclique en Haskell

L'exemple 8.7 décrit une liste chaînée dont l'unique nœud contient la valeur `0` et est suivi de lui-même.

Bien que les types algébriques d'Haskell soient définis différemment des énumérations d'Enumlang, ceux-ci permettent d'exprimer les mêmes opérations de sommes et de produits que les sur-énumérations et les énumérations à attributs. De plus, les valeurs d'Haskell possèdent la propriété d'immutabilité profonde de nos constantes, voire de comparaison par valeur selon la manière dont le programmeur l'implémente. Haskell permet aussi la création de valeurs cycliques.

Cependant, les contextes d'Haskell et d'Enumlang sont très différents, puisque le premier est un langage fonctionnel là où nous nous intéressons aux langages à objets, limitant les comparaisons possibles entre ces langages.

8.3 Objets immutables

8.3.1 Java

À partir de Java 16, la JEP 395 introduit des classes d'objets immutables nommées « enregistrements » à Java (Bierman, 2020). Celles-ci héritent de `java.lang.Record` et définissent automatiquement leurs constructeurs et certaines méthodes

telles que `Object.equals` et `Object.hashCode` pour que leurs objets soient comparés par valeur.

```
record Transaction(Person sender, Person receiver, int amount)
    {
     // ...
}
```

FIGURE 8.8 – Exemple d'enregistrement en Java

L'exemple 8.8 présente la déclaration d'une classe d'enregistrements `Transaction` contenant trois attributs `sender`, `receiver` et `amount`, de types `Person` et `int`.

Comme les constantes à attributs d'Enumlang, les enregistrements de Java sont composés d'attributs, immutables et comparés par valeur avec la méthode `Object.equals`. Ils peuvent cependant référencer des objets mutables tels que des instances et ne sont donc pas universels comme les constantes d'Enumlang.

8.3.2 C#

C#, à partir de sa version 9, permet de définir des enregistrements. Ceux-ci sont assez similaires à ceux de Java puisqu'ils agissent comme des classes immutables définissant automatiquement leurs constructeurs et étant comparés par valeurs. Contrairement au Java, les enregistrements de C# peuvent hériter d'autres enregistrements (Hejlsberg *et al.*, 2003).

L'exemple 8.9 présente la déclaration d'une classe d'enregistrements `Transaction` contenant trois attributs `sender`, `receiver` et `amount`, de types `Person` et `int`.

Les enregistrements de C# étant très proches de ceux de Java, les commentaires que nous avons formulées sur ces derniers s'appliquent aussi aux enregistrements de C#.

```
record Transaction(Person sender, Person receiver, int amount)
    {
     // ...
}
```

FIGURE 8.9 – Exemple d'enregistrement en C#

8.3.3 Scala

Le Scala différencie les variables mutables et immutables avec les mots-clés `val` (*value*) et **`var`** (*variable*), y compris au niveau des attributs (Odersky *et al.*, 2004). Ainsi, une classe terminale dont tous les attributs sont déclarés avec `val` est effectivement immutable.

```
class Point(val x: Int, val y: Int):
  // ...
```

FIGURE 8.10 – Exemple de classe en Scala

L'exemple 8.10 décrit une classe `Point` dont les attributs sont immutables.

Scala possède également des *case classes* permettant de déclarer directement des classes d'objets immutables et comparés par valeur.

```
case class Point(x: Int, y: Int):
  // ...

val point = Point(0, 0)
val copy = point.copy()
// Affiche "true"
print(point == copy)
```

FIGURE 8.11 – Exemple de *case classe* en Scala

L'exemple 8.11 décrit une *case class* `Point`, dont un objet `point` est créé, copié, puis comparé à sa copie.

Scala permet donc de déclarer des classes d'objets immutables avec des classes simples et avec des *case classes*. Ces dernières s'approchent plus des énumérations d'Enumlang puisque leurs objets sont comparés par valeur. Elles sont également très similaires aux enregistrements de Java et C#, et sont sujettes aux mêmes critiques de notre part.

8.4 Typage propriétif

8.4.1 Rust

Rust est un langage de programmation dont le système de typage permet de traquer statiquement les références au sein d'un programme en combinant du typage propriétif et des annotations de régions (Matsakis et Klock, 2014). Ce système permet de garantir statiquement l'immutabilité et l'autonomie de divers objets.

```
struct Car<'a> {
    engine: Engine,
    driver: &'a Driver,
}
```

FIGURE 8.12 – Exemple de type avec régions en Rust

L'exemple 8.12 décrit un type `Car` ayant une région `'a` en paramètre générique. Celui-ci nous indique que tout objet du type `Car` dépend d'un ou plusieurs objets extérieurs et n'est donc pas autonome.

Les constantes étant immutables et sans références extérieures, la généralisation du contrôle statique de ces propriétés par un système de typage est pertinente pour nos recherches. Cependant, le typage propriétif de Rust introduit une complexité non négligeable au langage avec des annotations de régions et une analyse statique poussée qui ne correspondent pas aux contraintes d'un langage à objets simple que nous avons définies pour Enumlang.

8.4.2 Jimuva

Jimuva est un langage à objets créé dans le cadre de l'article *Objets immutables pour un langage à la Java* (Haack *et al.*, 2007). Celui-ci implémente un système de classes immutables et différentes fonctionnalités qui leur sont liées.

```
immutable class ImmutableInt ext Object {
    int value;

    anon wrlocal ImmutableInt.k(int i) {
        this.value = i;
    }

    rdonly int get() {
        this.value
    }
}
```

FIGURE 8.13 – Exemple de classe immutable en Jimuva

Afin de restreindre la mutabilité de ses objets, Jimuva utilise un système comprenant deux droits d'accès.

— Le droit `rdwr` (*read-write*) permet l'accès à un objet en lecture et en écriture.
— Le droit `rd` (*read*) permet l'accès à un objet en lecture seule.

Jimuva associe chaque objet à un propriétaire afin de contrôler ses droits d'accès, chaque objet n'étant accessible que par l'intermédiaire de son propriétaire :

— Le propriétaire d'un objet peut être un autre objet tel que `this`.
— Le propriétaire d'un objet peut être générique avec le terme `myowner`.
— Le propériatire d'un objet peut être inconnu avec le terme `world`.

Finalement, afin de s'assurer que les droits d'accès sont respectés dans un programme, Jimuva possède différentes annotations permettant de limiter la portée de certaines fonctions :

— L'annotation `rdonly` (*read-only*) utilisée sur une méthode indique qu'elle ne mute pas son receveur.
— L'annotation `immutable` (*immutable*) utilisée sur une classe indique que ses instances sont immutables.
— L'annotation `anon` (*anonymous*) utilisée sur un constructeur lui interdit de faire fuiter l'objet en construction.
— L'annotation `wrlocal` (*write-local*) utilisée sur un constructeur lui interdit d'écrire dans une autre instance de la même classe.

Jimuva permet donc de définir des classes et des objets immutables voire autonomes dans un langage à objets, et donc de garantir certaines propriétés des constantes sur des classes plus générales. Cependant, ce langage utilise un système de typage propriétif, des droits d'accès, et des annotations de fonctions qui ne correspondent encore une fois pas aux contraintes de simplicité que nous avons fixées pour Enumlang.

8.5 Héritage énumératif

8.5.1 Types énumératifs avancés

Comme décrit dans la section 8.1, une énumération désigne dans la plupart des langages de programmation un type composé d'un ensemble de constantes nommées (Giles, 2001; Gonzalez-Perez, 2018b), correspondant en Enumlang à une énumération listée. Bien que ces énumérations puissent parfois avoir des méthodes voire hériter d'interfaces, l'héritage d'énumérations est généralement impossible.

L'article *Advanced Enumerated Types* (Gonzalez-Perez, 2018a) présente une approche pour définir une relation de spécialisation entre des énumérations listées. Plus précisément, il affirme qu'une énumération listée peut hériter d'une autre énumération listée en divisant les constantes héritées en sous-constantes, mais ne peut dans ce cas pas introduire de nouvelle constante indépendante.

```
SectionThemes: Fiction
               Biography
               Reference

BookGenres: Fiction
                Crime
                Fantasy
                ScienceFiction
                Historical
            Biography
                Autobiography
            Reference
                TextBooks
                Dictionaries
```

FIGURE 8.14 – Exemple d'énumérations avancées

L'exemple 8.14, tiré de l'article, définit une énumération `BookGenres` qui spécialise l'énumération `SectionThemes`. Nous pouvons constater que chacune des

constantes de `BookGenres` est associée à une constante de `SectionThemes`, divisant effectivement ces dernières en sous-constantes spécialisées. Une transcription de cet exemple en Enumlang est présentée dans la figure 8.15.

```
supenum BookGenre {
    # ...
}

enum Fiction {
    super BookGenre;

    case crime;
    case fantasy;
    case science_fiction;
    case historical;
}

enum Biography {
    super BookGenre;

    case autobiography;
}

enum Reference {
    super BookGenre;

    case textbook;
    case dictionary;
}
```

FIGURE 8.15 – Transcription de l'exemple 8.14 en Enumlang

Nous trouvons cet article pertinent puisqu'il possède comme nos travaux une vision ensembliste des énumérations en permettant de diviser celles-ci en sous-énumérations. Cependant, cet article ne s'intéresse qu'aux énumérations classiques et n'aborde pas les questions d'héritage et de langages à objets tel que nous le faisons. Cet article ne présente pas non plus d'implémentation de son système de spécialisation.

8.5.2 Classes à prédicats

L'article *Predicate Classes* (Chambers, 1993) présente un système d'héritage permettant la définition de classes héritant d'autres classes par prédicat dans le langage Cecil.

```
object list;
field elements(l@list);
method length(l@list) { l.elements.length }

pred empty_list isa list when list.length = 0;
method first(l@empty_list) { ... } -- Soulève une erreur.

pred non_empty_list isa list when not(list.length = 0);
method first(l@non_empty_list) { ... } -- Retourne le premier é
   lément de la liste.
```

FIGURE 8.16 – Exemple de classes à prédicats en Cecil

L'exemple 8.16 présente une classe `list` héritée par les deux classes à prédicats `empty_list` et `non_empty_list`. Ces deux sous-classes possèdent des définitions différentes pour la méthode `first`. Une transcription de cet exemple en Enumlang est présentée dans la figure 8.17.

Comparé à notre système d'héritage évaluatif, Cecil permet additionnellement d'hériter par prédicat de classes mutables. Cette approche permet une application des prédicats plus générale qu'en Enumlang mais limite en contrepartie les possibilités de mémoïsation puisque les prédicats doivent être réévalués après tout changement d'état potentiel.

```
subenum EmptySequence[T] {
    super Sequence[T];

    predicate {
        return self.length() == 0;
    }

    fun first(): T {
        error("Cannot get an element of an empty list");
    }
}

subenum NonEmptySequence[T] {
    super Sequence[T];

    predicate {
        return self.length() > 0;
    }

    fun first(): T {
        return self.get(0);
    }
}
```

FIGURE 8.17 – Transcription de l'exemple 8.16 en Enumlang

8.6 Formes alternatives

8.6.1 JitDS-Java

L'article *Just-in-Time Data Structures* (De Wael *et al.*, 2015) présente JitDS-Java, une extension de Java implémentant un système de combinaison de classes permettant à un objet de changer de représentation au cours de sa vie. Ceci permet alors à l'objet d'être dans une représentation optimale pour chacun des algorithmes auquel il est sujet.

```
class Matrix combines RowMajorMatrix, ColMajorMatrix {
    RowMajorMatrix to ColMajorMatrix {
        target(source.getCols(),
            source.getRows(),
            source.getDataAsArray());
        target.transpose();
    }

    ColMajorMatrix to RowMajorMatrix {
        target(source.getCols(),
            source.getRows(),
            source.getDataAsArray());
        target.transpose();
    }
}
```

FIGURE 8.18 – Exemple de *just-in-time data structure* en JitDS-Java

L'exemple 8.18 présente une classe `Matrix` combinant deux autres classes `RowMajorMatrix` et `ColMajorMatrix` (omises dans l'exemple) et définissant des convertisseurs permettant de transformer un objet d'une classe à l'autre.

JitDS-Java présente une solution au problème de représentations multiples assez semblable aux formes alternatives d'Enumlang, puisque chaque forme est déclarée dans sa propre classe et que les conversions entre celles-ci se font par le biais de convertisseurs exécutant du code arbitraire.

Contrairement à Enumlang, JitDS-Java ne se restreint pas aux constantes et présente une solution certes plus générale mais posant également divers problèmes. Notamment, JitDS-Java ne supporte pas l'échappement de références issues de représentations alternatives, puisque celles-ci peuvent ne plus être maintenues après d'éventuelles mutations de leurs instances. En se restreignant à des objets immutables tels que les constantes, Enumlang évite la persistance de représentations obsolètes et permet ainsi le partage de références sans restrictions central à l'objec-

tif de simplicité de ce langage. Également, cette immutabilité permet à Enumlang de mémoïser les formes alternatives d'une constante et ainsi d'éviter l'exécution répétée de convertisseurs entre différentes formes.

8.7 Interfaces énumératives

8.7.1 Haskell

Afin de permettre la description d'ensembles dénombrables, le langage Haskell possède les classes (selon la terminologie du langage) `Bounded` et `Enum` définies dans le module `GHC.Enum` (Marlow *et al.*, 2010) :

```
class Bounded a where
    minBound, maxBound :: a

class Enum a where
    succ :: a -> a
    pred :: a -> a
    toEnum   :: Int -> a
    fromEnum :: a -> Int
    enumFrom       :: a -> [a]
    enumFromThen   :: a -> a -> [a]
    enumFromTo     :: a -> a -> [a]
    enumFromThenTo :: a -> a -> a -> [a]
```

FIGURE 8.19 – Définitions des classes `Bounded` et `Enum` en Haskell

— La classe `Bounded` permet de définir deux valeurs associées à un type correspondant à ses bornes basses et hautes.

— La classe `Enum` permet de définir diverses fonctions correspondant à des opérations possibles sur les ensembles séquentiels. Les fonctions `toEnum` et `fromEnum` suffisent à implémenter cette classe pour un type, celles-ci

suffisant à composer les autres fonctions de la classe en utilisant l'indexage des valeurs de ce type aux entiers.

Les énumérations représentant des ensembles de constantes, la définition d'interfaces telles que `Bounded` et `Enum` pour décrire ces ensembles nous paraît pertinente, bien qu'actuellement pas implémentée en Enumlang.

8.8 Conclusion

Dans ce chapitre, nous avons présenté différents travaux ou articles liés par nos recherche. Nous avons d'abord abordé les définitions et représentations d'énumérations classiques dans divers langages de programmation, et avons remarqué qu'elles correspondent souvent à nos énumérations listées.

Nous avons ensuite présenté différentes implémentations des objets immutables, dont notamment la définition de classes immutables comparées par valeur dans des langages à objets tels que Java, C# et Scala. Nous avons également souligné certaines similarités entre les constantes et les valeurs de langages fonctionnels tels que Haskell.

Nous avons également abordé le court article *Advanced Enumerated Types* (Gonzalez-Perez, 2018a) présentant une approche de l'héritage pour les énumérations classiques en divisant leurs constantes en sous-énumérations.

Finalement, nous avons présenté les articles *Predicate Classes* (Chambers, 1993) et *Just-in-Time Data Structures* (De Wael *et al.*, 2015) qui présentent des systèmes respectivement similaires à notre héritage évaluatif et à nos formes alternatives, mais qui en adoptant une solution plus générale sans contrainte de mutabilité se heurtent à divers problèmes.

CONCLUSION

Nous avons présenté dans ce mémoire nos recherches sur l'intégration des constantes et énumérations à un langage à objets. Nous définissons les constantes comme étant des objets universels s'opposant aux instances et étant profondément immutables et sans identité. Les énumérations sont les classes représentant des ensembles de constantes.

Nous avons premièrement présenté Enumlang, un langage à objets développé dans le cadre de nos recherches et pour lequel nous avons développé un interpréteur. Nous avons ensuite présenté les notions de constantes et d'énumérations ainsi que les différents genres d'énumérations introductrices d'Enumlang et leurs propriétés. Nos énumérations ont une cardinalité fixe et peuvent être récursives, en revanche, leurs constantes ne peuvent pas contenir de cycles ou de valeurs nulles.

Nos énumérations existant dans un contexte d'objets, l'un de nos objectifs principaux était l'étude de l'héritage énumératif. Nous avons d'abord remarqué que l'héritage instanciatif n'est pas cohérent avec le concept d'énumérations. Nous avons donc développé le concept d'héritage évaluatif, une approche de l'héritage reposant sur l'évaluation de prédicats pour déterminer l'appartenance ou non d'une constante à une énumération.

Nous avons ensuite conçu et développé un système de formes alternatives, une fonctionnalité permettant à une même constante d'avoir plusieurs représentations possibles. En Enumlang, chaque énumération possède une forme principale et éventuellement des formes secondaires, et il est possible d'utiliser un code spécifique pour convertir automatiquement une constante d'une forme à une autre.

Les formes alternatives permettent notamment de définir plusieurs versions d'une même méthode pour les différentes formes d'une énumération.

Finalement, nous avons présenté un exemple pratique utilisant les différentes fonctionnalités énumératives que nous avons développées lors de nos recherches. Nous pensons que le sujet de la représentation des valeurs universelles en programmation, et notamment dans les langages à objets, est un sujet encore peu exploré et présentant une opportunité pour d'autres recherches futures.

ANNEXE A

GRAMMAIRE D'ENUMLANG

Cette annexe présente la grammaire d'Enumlang. Celle-ci est également disponible dans le fichier `grammar.sablecc` de l'interpréteur.

```
digit = '0' .. '9';
letter = ('a' .. 'z') | ('A' .. 'Z');

integer = digit+;
float = digit+ '.' digit+;
string = Shortest '"' (Any - endline)* '"';
name = (letter | '_') (letter | digit | '_')*;

whitespace = #9 | ' ';
endline = #13 | #10;
comment = '#' (Any - endline)*;
```

Listing A.1 – Grammaire du lexeur d'Enumlang

```
file = topdef* statement*;

// Topdefs
topdef = module | class | function | constant;

// Modules
module = 'module' name '{' module_default? topdef* '}';
module_default = 'default' class;

// Classes
```

```
class = 'abstract'? class_kind name generic_parameters?
   class_form? '{' member* '}';
class_kind = 'interface' | 'class' | 'enum' | 'supenum' | '
   subenum';
class_form = '.' name;
member
= 'as' name block
| 'partition' name block
| 'predicate' block
| 'redef'? 'abstract'? function
| member_expression ';';
member_expression = 'super' item | 'var' name typing | 'case'
   name;

// Functions
function = 'primitive'? 'fun' name generic_parameters? '('
   function_parameters_list? ')' typing? function_body;
function_parameters_list = function_parameter
   function_parameters_more?;
function_parameters_more = function_parameters_more? ','
   function_parameter;
function_parameter = name typing;
function_body = block | ';';

// Constants
constant = 'const' name typing '=' expression ';';

// Items
item = item_base? name generic_arguments?;
item_base = item '.';

// Generics
generic_parameters = '[' generic_parameters_list? ']';
generic_parameters_list = generic_parameter
   generic_parameters_more?;
generic_parameters_more = generic_parameters_more? ','
   generic_parameter;
generic_parameter = name typing?;
generic_arguments = '[' generic_arguments_list? ']';
generic_arguments_list = item generic_arguments_more?;
generic_arguments_more = generic_arguments_more? ',' item;

// Types
typing = ':' item;
```

```
// Statements
statement = statement_block | statement_expression ';';
statement_block
= block
| 'match' expression '{' case* default? '}'
| 'if' expression block else?
| 'loop' block
| 'while' expression block
| 'for' name 'in' expression block;
statement_expression
= variable
| reference '=' expression
| 'return' expression?
| 'select' item?
| expression;
block = '{' statement* '}';
case = 'case' expression block;
default = 'default' block;
else = 'else' block;
variable = 'var' name typing?;
reference = variable | expression;

// Expressions
expression = expression infop6 expression8 | expression8;
expression8 = expression8 infop5 expression8 | expression7;
expression7 = expression7 infop4 expression8 | expression6;
expression6 = expression6 infop3 expression8 | expression5;
expression5 = expression5 infop2 expression8 | expression4;
expression4 = expression4 infop1 expression8 | expression3;
expression3 = expression3 'as' item | expression3 'isa' item |
   expression2;
expression2 = preop expression2 | 'new' item | expression1;
expression1
= expression0 generic_arguments?
| 'void'
| 'true'
| 'false'
| integer
| float
| string
| 'self'
| 'super'
| '(' expression ')'
```

```
| expression1 '(' arguments_list? ')';
expression0 = name | expression1 '.' name;
arguments_list = expression arguments_more?;
arguments_more = arguments_more? ',' expression;

// Operators
preop  = '+' | '-' | '!';
infop1 = '*' | '/' | '%';
infop2 = '+' | '-';
infop3 = '<' | '>' | '<=' | '>=';
infop4 = '==' | '!=';
infop5 = '&&';
infop6 = '||';
```

Listing A.2 – Grammaire du parseur d'Enumlang

ANNEXE B

INTERPRÉTEUR

Cette annexe présente la structure et le fonctionnement de l'interpréteur Enumlang développé dans le cadre de nos recherches. Nous y expliquons notamment les différentes étapes de l'analyse et de l'interprétation d'un programme Enumlang.

B.1 Présentation

L'interpréteur Enumlang développé dans le cadre de nos recherches permet d'exécuter des programmes Enumlang. Celui-ci, programmé en Java, peut être découpé en plusieurs parties dont les usages diffèrent lors de l'interprétation d'un programme :

- Premièrement, le parseur génère un arbre de syntaxe abstrait afin de mieux structurer le programme selon la grammaire d'Enumlang.
- Ensuite, le modélisateur crée un modèle du programme à partir de l'arbre de syntaxe abstrait. Ce modèle établit les liens entre les différents méta-objets du programme pour leur donner un sens sémantique.
- Enfin, l'interpréteur proprement dit, parcourt le modèle afin d'exécuter le programme.

Il est intéressant de noter que chacune des structures générées à chaque étape est indépendante des suivantes. Par exemple, un unique programme modélisé peut être exécuté plusieurs fois sans que cela ne cause de problèmes.

B.2 Parseur

L'interpréteur d'Enumlang utilise SableCC 3 afin de générer l'arbre de syntaxe concret d'un programme à partir de sa forme textuelle. SableCC est un générateur de parseur développé par Étienne Gagnon générerant notamment un parseur LALR à partir d'une grammaire non contextuelle tout en vérifiant statiquement son absence de conflits. À partir de l'arbre de syntaxe concret, l'interpréteur génère un arbre de syntaxe abstrait en utilisant un visiteur fourni par SableCC.

B.3 Modélisation

Enumlang étant un langage à typage statique, l'interprétation d'un programme demande préalablement sa vérification statique. Ainsi, l'interpréteur Enumlang procède à un travail de modélisation et d'analyse statique pour valider la conformité de tout programme.

Ce travail de modélisation est découpé en plusieurs étapes exécutées par ordre de dépendance, et résultant en un modèle du programme sémantisé et valide pouvant être exécuté :

1. L'interpréteur découvre l'ensemble des déclarations statiques du programme.
2. L'interpréteur modélise les relations d'héritage et de généricité des classes.
3. L'interpréteur vérifie la validité des relations précédentes (cycles d'héritage, bornes génériques...).

4. L'interpréteur modélise la structure des classes et les signatures des fonctions et méthodes.
5. L'interpréteur modélise le code en créant un arbre sémantique à partir de l'arbre de syntaxe abstrait. Ce nouvel arbre est entièrement typé et lié aux structures créées précédemment par le biais de références plutôt que de simples noms.

B.4 Interprétation

Afin d'exécuter un programme, l'interpréteur Enumlang parcourt le modèle en y exécutant récursivement les différentes instructions et expressions qu'il rencontre. L'interpréteur crée sa propre pile d'appels de fonctions et instancie les objets demandés par le code Enumlang. La modélisation ne faisant pas de monomorphisation des types génériques, les fonctions et classes utilisées sauvegardent ces types lors de l'exécution et effectuent les conversions nécessaires afin de garantir le bon fonctionnement des diverses opérations de types dynamiques.

Il n'existe actuellement pas de compilateur ou de code octet pour Enumlang.

B.5 Vérification

Afin de s'assurer du bon fonctionnement de l'interpréteur au cours de son développement, celui-ci est accompagné d'un système de test. Ce système fonctionne en vérifiant que les sorties standards et d'erreurs correspondent à celles attendues sur un jeu de tests contenant plus d'une centaine de programmes.

ANNEXE C

DÉFINITIONS ALTERNATIVES D'ÉNUMÉRATIONS PRIMITIVES

C.1 `Boolean`

L'énumération primitive `Boolean` pourrait également être définie comme une énumération listée où **`true`** et **`false`** sont des constantes nommées.

```
enum Boolean {
    case true;
    case false;
}
```

Listing C.1 – Définition alternative de l'énumération `Boolean`

C.2 `Maybe`

L'énumération primitive `Maybe` pourrait également être définie avec un ensemble d'énumérations.

```
supenum Maybe[T] {
    # ...
}

enum Some[T] {
    super Maybe[T];

    var constant: T;
```

```
}

enum None[T] {
    super Maybe[T];

    case none;
}
```

Listing C.2 – Définition alternative de l'énumération `Maybe`

En considérant un type vide `Never`, sous-type de tous les types, il serait également possible de définir `None` sans avoir recours à une énumération paramétrique, permettant d'utiliser une même constante `none` avec des `Maybe` de types différents.

```
enum None {
    super Maybe[Never];

    case none;
}
```

Listing C.3 – Définition de `None` avec le type `Never`

C.3 `Sequence`

L'énumération primitive `Sequence` pourrait également être définie comme une liste chaînée avec un ensemble d'énumérations.

```
supenum Sequence[T] {
    # ...
}

enum More[T] {
    super Sequence[T];

    var constant: T;
    var more: Sequence[T];
}

enum Empty[T] {
```

```
    super Sequence[T];

    case empty;
}
```

Listing C.4 – Définition alternative de l'énumération `Sequence`

En considérant un type vide `Never`, sous-type de tous les types, il serait également possible de définir `Empty` sans avoir recours à une énumération paramétrique, permettant d'utiliser une même constante `empty` avec des `Sequence` de types différents.

```
enum Empty {
    super Sequence[Never];

    case none;
}
```

Listing C.5 – Définition de `Empty` avec le type `Never`

ANNEXE D

CODE COMPLET DE L'EXEMPLE DE SYSTÈME NUMÉRIQUE

Cette annexe présente le code complet de l'exemple du chapitre 7.

```
fun gcd(a: Integer, b: Integer): Integer {
    if b == 0 {
        return a;
    } else {
        return gcd(b, a % b);
    }
}

fun rational(num: Integer, denom: Integer): Rational {
    return Rational.Fraction(num, denom);
}

fun integer(value: Integer): RationalInteger {
    return RationalInteger.Simple(value);
}

enum Rational {
    var num: Integer;
    var denom: Integer;

    predicate {
        return self.denom > 0 && gcd(self.num, self.denom) ==
           1;
    }

    as Fraction {
        return Rational.Fraction(self.num, self.denom);
```

```
    }

    redef fun to_string(): String {
        return "" + self.num + "/" + self.denom;
    }

    fun get_num(): Integer {
        return self.num;
    }

    fun get_denom(): Integer {
        return self.denom;
    }

    fun get_inverse(): Rational {
        return rational(self.denom, self.num);
    }

    fun get_opposite(): Rational {
        return rational(-self.num, self.denom)
    }

    fun add(other: Rational): Rational {
        return other.add_rational(self);
    }

    fun sub(other: Rational): Rational {
        return self.add(other.get_opposite());
    }

    fun mul(other: Rational): Rational {
        return other.mul_rational(self);
    }

    fun div(other: Rational): Rational {
        return self.mul(other.get_inverse());
    }

    fun add_rational(other: Rational): Rational {
        return rational(other.num * self.denom + self.num *
           other.denom, other.denom * self.denom);
    }

    fun mul_rational(other: Rational): Rational {
```

```
        return rational(other.num * self.num, other.denom *
            self.denom);
    }

    fun add_integer(other: RationalInteger): Rational {
        return rational(other.get_value() * self.denom + self.
            num, self.denom);
    }

    fun mul_integer(other: RationalInteger): Rational {
        return rational(other.get_value() * self.num, self.
            denom);
    }
}

enum Rational.Fraction {
    var num: Integer;
    var denom: Integer;

    predicate {
        return self.denom != 0;
    }

    as Rational {
        var num = self.num;
        var denom = self.denom;
        if denom < 0 {
            num = -num;
            denom = -denom;
        }

        var divisor = gcd(num, denom);
        return Rational(num / divisor, denom / divisor);
    }
}

subenum RationalInteger {
    super Rational;

    predicate {
        return self.denom == 1;
    }

    as Simple {
```

```
        return RationalInteger.Simple(self.num);
    }

    fun get_value(): Integer {
        return self.num;
    }
}

subenum RationalInteger.Simple {
    var value: Integer;

    as RationalInteger {
        return RationalInteger(self.value, 1);
    }

    fun get_value(): Integer {
        return self.value;
    }

    redef fun to_string(): String {
        return self.value.to_string();
    }

    redef fun get_num(): Integer {
        return self.value;
    }

    redef fun get_denom(): Integer {
        return 1;
    }

    redef fun get_inverse(): Rational {
        return rational(1, self.value);
    }

    redef fun get_opposite(): Rational {
        return integer(-self.value);
    }

    redef fun add(other: Rational): Rational {
        return other.add_integer(self);
    }

    redef fun mul(other: Rational): Rational {
```

```
        return other.mul_integer(self);
    }

    redef fun add_rational(other: Rational): Rational {
        return rational(other.num + self.value * other.denom,
           other.denom);
    }

    redef fun mul_rational(other: Rational): Rational {
        return rational(other.num * self.value, other.denom);
    }

    redef fun add_integer(other: RationalInteger):
       RationalInteger {
        return integer(other.get_value() + self.value);
    }

    redef fun mul_integer(other: RationalInteger):
       RationalInteger {
        return integer(other.get_value() * self.value);
    }
}

var a = rational(3, 2);
var b = rational(6, 3);
var c = integer(5);

print(a);
print(b);
print(c);

print(a.add(b));
print(a.sub(c));
print(b.mul(c));
```

Listing D.1 – Exemple de système numérique complet

BIBLIOGRAPHIE

Amin, N. et Tate, R. (2016). Java and scala's type systems are unsound : The existential crisis of null pointers. Dans *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, p. 838–848., New-York, États-Unis. Association for Computing Machinery

Arnold, K., Gosling, J. et Holmes, D. (2005). *The Java programming language*. Addison Wesley Professional.

Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K. et Withington, P. T. (1996). A monotonic superclass linearization for dylan. Dans *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, p. 69–82., New-York, États-Unis. Association for Computing Machinery

Bertino, E. et Guerrini, G. (1995). Objects with multiple most specific classes. Dans M. Tokoro et R. Pareschi (dir.). *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, 102–126., Berlin et Heidelberg, Allemagne. Springer Berlin Heidelberg

Bierman, G. (2020). Jep 395 : Records. Récupéré de `https://openjdk.java.net/jeps/395`

Bird, R. S. (2012). On building cyclic and shared structures in haskell. *Form. Asp. Comput.*, *24*(4–6), 609–621

Chalin, P. et Rioux, F. (2005). Non-null references by default in the java modeling language. Dans *Proceedings of the 2005 Conference on Specification and Verification of Component-Based Systems*, SAVCBS '05, p. 9–es., New-York, États-Unis. Association for Computing Machinery

Chambers, C. (1993). Predicate classes. Dans O. M. Nierstrasz (dir.). *ECOOP' 93 — Object-Oriented Programming*, 268–296., Berlin et Heidelberg, Allemagne. Springer Berlin Heidelberg

Clarke, D. G., Potter, J. M. et Noble, J. (1998). Ownership types for flexible alias protection. Dans *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, p. 48–64., New-York, États-Unis. Association for Computing Machinery

De Wael, M., Marr, S., De Koster, J., Sartor, J. B. et De Meuter, W. (2015). Just-in-time data structures. Dans *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, p. 61–75., New-York, États-Unis. Association for Computing Machinery

Fähndrich, M. et Leino, K. R. M. (2003). Declaring and checking non-null types in an object-oriented language. Dans *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, p. 302–312., New-York, États-Unis. Association for Computing Machinery

Giles, J. (2001). Enumerated data types. *SIGPLAN Fortran Forum*, *20*(1), 11–15

Gonzalez-Perez, C. (2018a). *Advanced Enumerated Types*, Dans *Information Modelling for Archaeology and Anthropology : Software Engineering Principles for*

Cultural Heritage, (p. 101–104). Cham, Suisse. Springer International Publishing

Gonzalez-Perez, C. (2018b). *Enumerated Types*, Dans *Information Modelling for Archaeology and Anthropology : Software Engineering Principles for Cultural Heritage*, (p. 57–63). Cham, Suisse. Springer International Publishing

Haack, C., Poll, E., Schäfer, J. et Schubert, A. (2007). Immutable objects for a java-like language. Dans R. De Nicola (dir.). *Programming Languages and Systems*, 347–362., Berlin et Heidelberg, Allemagne. Springer Berlin Heidelberg

Hejlsberg, A., Wiltamuth, S. et Golde, P. (2003). *Csharp Language Specification*. États-Unis : Addison-Wesley Longman Publishing Co., Inc.

Kernighan, B. W. et Ritchie, D. M. (1989). *The C Programming Language*. États-Unis : Prentice Hall Press.

Marlow, S. *et al.* (2010). Haskell 2010 language report. Récupéré de `https://www.haskell.org/definition/haskell2010.pdf`

Matsakis, N. D. et Klock, F. S. (2014). The rust language. Dans *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, p. 103–104., New-York, États-Unis. Association for Computing Machinery

Odersky, M., Altherr, P., Cremet, V., Emir, B., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E. et Zenger, M. (2004). The scala language specification.

Pierce, B. C. (2002). *Types and Programming Languages* (1st éd.). The MIT Press.

Privat, J. (2015). A concise reference of the nit language. Récupéré de `https://nitlanguage.org/manual/nitreference.pdf`

Sălcianu, A. et Rinard, M. (2005). Purity and side effect analysis for java programs. Dans R. Cousot (dir.). *Verification, Model Checking, and Abstract Interpretation*, 199–215., Berlin et Heidelberg, Allemagne. Springer Berlin Heidelberg

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. Dans *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, OOPSLA '86, p. 38–45., New-York, États-Unis. Association for Computing Machinery

Stroustrup, B. (1988). What is object-oriented programming ? *IEEE Software*, *5*(3), 10–20

Taivalsaari, A. (1996). On the notion of inheritance. *ACM Comput. Surv.*, *28*(3), 438–479

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, *1*(1), 7–87