Structures discrètes

Licence informatique 2e année

2024-2025

Table des matières

1 Introduction **1**
- 1.1 La notion d'algorithme . . . 1
- 1.2 Plan de l'enseignement . . . 8

2 Terminaison d'algorithmes **10**
- 2.1 Un problème difficile . . . 10
- 2.2 Rappels sur les relations et les ordres . . . 11
- 2.3 Relations d'ordre bien fondées . . . 11
- 2.4 Preuve de terminaison des boucles . . . 14
- 2.5 Preuves de terminaisons des algorithmes récursifs . . . 17

3 Définitions inductives et structures inductives **20**
- 3.1 Définir des valeurs . . . 20
- 3.2 Exemple introductif : les entiers naturels . . . 22
- 3.3 Deuxième exemple : les expressions arithmétiques . . . 24
- 3.4 Formes des définitions inductives . . . 26
- 3.5 Représentation des définitions inductives . . . 28
- 3.6 Arbre de dérivation . . . 30

4 Quelques définitions inductives classiques **35**
- 4.1 Les séquences (ou listes) . . . 35
- 4.2 Les algèbres de termes . . . 36
- 4.3 Les arbres d'arité non-fixe . . . 38

5 Fonctions et algorithmes sur les structures inductives **40**
- 5.1 Définitions inductives et ordres bien-fondés . . . 40
- 5.2 Fonctions définies inductivement . . . 42
- 5.3 Fonctions inductives à plusieurs arguments . . . 48
- 5.4 Preuves par induction . . . 50

6 Application : une introduction à la sémantique **55**

7 Preuves de correction d'algorithmes **58**
- 7.1 Correction d'un algorithme : définition . . . 58
- 7.2 Une logique minimale (lecture facultative). . . . 59
- 7.3 La logique de Hoare . . . 61
- 7.4 Extensions . . . 68
- 7.5 Présentation de la logique de Hoare . . . 71

8 Complexité algorithmique **75**
8.1 Définir la complexité 75
8.2 Comparer les complexités d'algorithmes 78
8.3 Comment évaluer une complexité dans le pire des cas ? 82
8.4 Calculer la complexité d'un algorithme récursif 83

9 Correction des exercices **88**

1 Introduction

1.1 La notion d'algorithme

Un *algorithme* est une méthode automatique permettant d'obtenir un résultat à partir de données. Le terme *automatique* signifie qu'un algorithme est automatisable : réalisable par une machine comme une séquence d'étapes simples, il ne nécessite pas de prendre des décisions. Le plus souvent, un algorithme est décrit comme un suite finie d'instructions et d'opérateurs de contrôle (les boucles notamment). Dès le plus jeune âge, nous utilisons des algorithmes, comme celui de l'addition qui fournit une méthode générale pour le calcul de la somme de deux entiers, et ce, peu importe le nombre de chiffres [1]. Nous en utilisons également souvent inconsciemment dans la vie quotidienne en suivant une recette de cuisine, en tricotant, ou en appliquant des stratégies bien déterminées dans certains jeux ou sports.

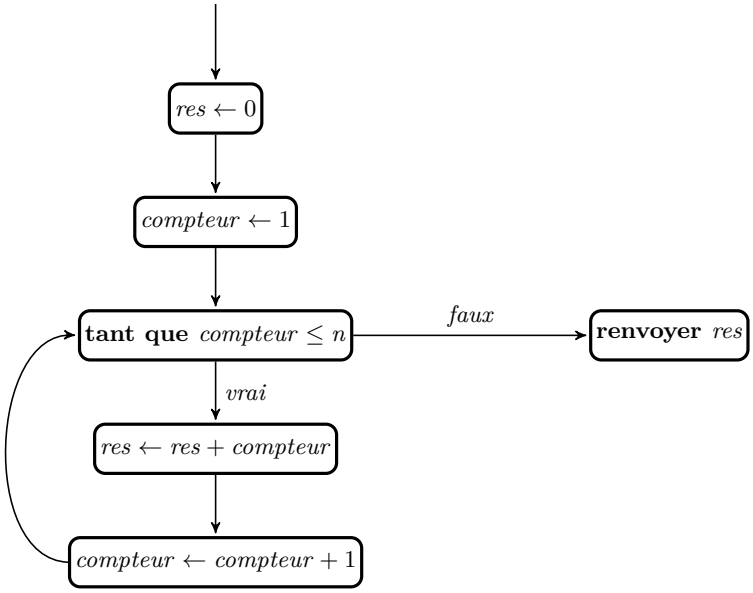
L'Algorithme 1.1 va nous servir d'exemple pour introduire quelques éléments clés.

premierAlgorithme(n)
Entrée: un entier $n \geq 0$;
Sortie: un entier r.
1: **soit** $res = 0$
2: **soit** $compteur = 1$
3: **tant que** $compteur \leq n$ **faire**
4: $\quad res \leftarrow res + compteur$
5: $\quad compteur \leftarrow compteur + 1$
6: **renvoyer** res

Algorithme 1.1 – Un exemple d'algorithme tel que nous les étudions.

Un algorithme peut de façon équivalente être présenté par son *graphe de flot de contrôle* comme ci-dessous :

1. Notons que nous apprenons aux enfants les algorithmes de l'arithmétique sans donner leurs représentations sous la forme de suite d'instructions. Les algorithmes existent indépendamment de leur représentation.



Pour définir formellement ce graphe, nous dirons qu'une instruction ou une structure de contrôle est *évaluée* lors de l'exécution de l'algorithme, lorsque l'instruction est réalisée ou respectivement lorsque la condition de la structure de contrôle est calculée. Notons qu'en présence de boucle, chaque instruction ou structure de contrôle peut être évaluée plusieurs fois.

Définition 1.1. Le *graphe de flot de contrôle* d'un algorithme est un graphe orienté tel que

- ▷ chaque instruction et chaque structure de contrôle (**si**, **pour tout**, **tant que**,...) est un sommet,
- ▷ il existe un arc du sommet u vers le sommet v s'il existe une exécution de l'algorithme pour laquelle une évaluation de v est immédiatement consécutive d'une évaluation de u.

Nous distinguerons le sommet évalué en premier, à l'aide d'une flèche pointant sur ce sommet. Nous annoterons les arcs sortants des structures de contrôle par *vrai* ou *faux*, pour préciser le comportement de la structure de contrôle dans le graphe.

Exercice 1.1. Dessiner le graphe de flot de contrôle de l'algorithme suivant :

somme(n, p)
Entrée: deux entiers $n, p \geq 0$;
Sortie: un entier $s \geq 0$.
1: **tant que** $p > 0$ **faire**
2: $\quad n \leftarrow n + 1$
3: $\quad p \leftarrow p - 1$
4: **renvoyer** n

Syntaxe générale. Un algorithme est toujours la donnée d'une pré-condition, d'une post-condition et du corps de l'algorithme comme illustré ci-dessous

Entrée: pré-condition sur les entrées.
Sortie: post-condition sur les sorties.
1:
2: Corps de l'algorithme
3:

amU Faculté des sciences Aix Marseille Université

La pré-condition indique la nature des entrées avec éventuellement des restrictions ; dans notre exemple, l'algorithme n'a qu'une seule entrée notée n qui doit être un entier positif. Cela signifie que lorsque l'algorithme sera exécuté sur une entrée particulière (on l'appelle une *instance*) ce sera obligatoirement un entier positif, par exemple 12.

La post-condition exprime les propriétés que doivent satisfaire certaines données de l'algorithme (les sorties) une fois l'algorithme exécuté. C'est donc elle qui précise le rôle de l'algorithme. Dans notre exemple, la sortie — c'est-à-dire le résultat calculé par l'algorithme — n'est pour le moment pas spécifiée au-delà d'être un entier.

Le corps de l'algorithme est composé d'une séquence d'instructions et de structures de contrôle. Les *instructions* provoque un changement d'état élémentaire, comme une écriture en mémoire ou la création d'une nouvelle variable. Les *structures de contrôle* permettent des contrôler quelles instructions doivent être exécutées, dans quel ordre et combien de fois. Par exemple les boucles permettent de répéter des instructions, les conditionnelles (**si** ... **alors** ...) autorisent ou empêchent l'exécution d'instructions. L'algorithme peut être écrit en plusieurs fonctions, chacune étant définie avec ses préconditions et ses post-conditions. Les appels de fonction sont à ce titre une forme de contrôle du flot d'instructions. L'algorithme dispose alors d'une fonction particulière, sa *fonction principale*, qui est le point d'entrée de l'algorithme. Les appels de fonction sont aussi nécessaire pour écrire des algorithmes récursifs.

Variables et affectation. Les noms n, *res* et *compteur* sont des identifiants de *variables*. Une variable est une unité de mémorisation, assimilable à une boîte pouvant contenir une valeur. Dans l'exemple nous avons spécifié que les variables *res* et *compteur* ne peuvent contenir que des entiers [2].

Le contenu d'une variable peut être modifiée à tout moment. L'instruction de modification d'une variable s'appelle l'*affectation*, elle est représentée par l'instruction suivante :

$var \leftarrow$ expr

où *var* est l'identifiant d'une variable, et expr est une expression pouvant être évaluée en une valeur. L'affectation consiste à remplacer le contenu de cette variable par la valeur de expr. Si la variable contenant déjà une valeur, cette dernière disparaît.

Notons que l'opérateur d'affectation n'est pas commutatif, si a et b sont deux variables, les instructions $a \leftarrow b$ et $b \leftarrow a$ sont différentes comme illustré par cet exemple :

	valeur de a	valeur de b
initialement	2	3
après $a \leftarrow b$	3	3

	valeur de a	valeur de b
initialement	2	3
après $b \leftarrow a$	2	2

De façon générale, l'opérande gauche doit être un identifiant de variable (ou la case d'un tableau), spécifiquement elle doit désigner une unité de mémorisation. L'opérande droite désigne une expression représentant une valeur. L'identifiant d'une variable représente à la fois l'unité de mémorisation et son contenu qui est une valeur [3], et peut donc être utilisé à gauche comme à droite, mais pas avec le même rôle. Cela permet notamment de modifier la valeur d'une variable en fonction de sa valeur courante. Par exemple, pour augmenter de 1 la valeur de la variable a, nous écrivons $a \leftarrow a + 1$.

L'instruction renvoyer. L'instruction **renvoyer**, lorsqu'elle est exécutée, provoque toujours l'arrêt de la fonction en cours. Elle peut être utilisée seule, ou avec un argument : **renvoyer** expression, ce qui a d'évaluer expression en une valeur de retour, de terminer la fonction et de renvoyer à l'instruction ayant appelée la fonction. Enfin, cela donne comme valeur à l'expression de l'appel de fonction, la valeur de retour ainsi calculée. Dans le cas de la fonction principale, cela termine l'algorithme avec pour solution la valeur de retour.

2. Les types d'un langage de programmation permettent de représenter certaines de ces contraintes, et parfois même de les garantir.

3. Quelques langages de programmation proposent deux syntaxes différentes, l'une pour l'unité de mémorisation et l'autre pour la valeur contenue.

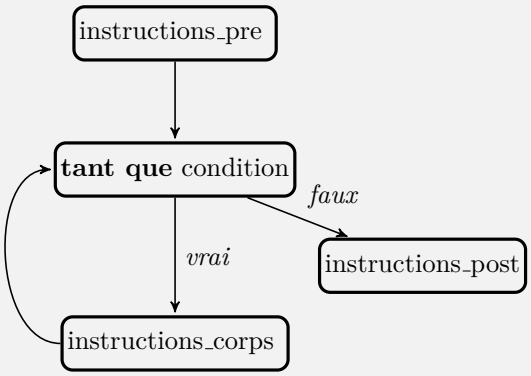


FIGURE 1 – Une boucle **tant que**

L'instruction tant que. Ce type d'instruction est souvent appelé *boucle* car elle a pour effet de répéter une suite d'instructions, autant de fois que nécessaire. Sa syntaxe générale est la suivante :

tant que condition **faire**
    instructions

L'expression condition est une expression booléenne (qui doit pouvoir s'évaluer à vrai ou faux). Dans notre exemple, c'est $compteur \leq n$. Supposons qu'au moment de l'évaluation, *compteur* contient la valeur c et n contient la valeur n, la valeur de $compteur \leq n$ est vrai ssi $c \leq n$. De manière générale, les opérateurs booléens ou arithmétiques seront interprétés selon leur sens usuel.

Le terme instructions réfère au *corps* de la boucle. C'est la séquence d'instructions simples ou de structures de contrôle qui sera répétée. Dans notre exemple il s'agit de la séquence

$$res \leftarrow res + compteur \qquad compteur \leftarrow compteur + 1$$

À chaque itération, la condition est testée :

- ▷ si elle est évaluée à *vrai*, le corps de la boucle est exécuté, avant de passer à l'itération suivante (et donc au test de la condition) ; nouveau tester la condition).
- ▷ si elle est évaluée à *faux*, l'algorithme ignore le corps de la boucle, et continue avec l'instruction suivante, en-dessous du corps. L'exécution *quitte* la boucle.

Pour mieux comprendre, représentons la portion d'algorithme suivante par un graphe de flot de contrôle, donné en Figure 1 :

instructions_pre
tant que condition **faire**
    instructions_corps
instructions_post

La condition est donc évaluée avant d'exécuter le corps. La condition n'est pas évaluée pendant l'exécution du corps : toute itération qui commence l'exécution des instructions du corps évalue tout le corps, sans interruption sauf en présence d'une instruction spécifique comme **renvoyer**.

Simulons maintenant l'exécution de notre algorithme lorsque l'entrée n a pour valeur 5. Les deux premières lignes du corps de l'algorithme fixent les valeurs des variables *res* et *compteur* respectivement à 0 et 1. Nous représentons l'exécution de la boucle par un tableau, dont chaque colonne représente l'état de la mémoire et la valeur de la condition après un nombre donné d'itérations du bloc d'instructions, précisément au moment d'évaluer la condition de la boucle.

itérations	0	1	2	3	4	5
n	5	5	5	5	5	5
res	0	1	3	6	10	15
compteur	1	2	3	4	5	6
$compteur \leq n$	vrai	vrai	vrai	vrai	vrai	faux

Dans la dernière colonne, après 5 itérations, l'expression *compteur* $\leq n$ est évaluée à faux, la boucle s'arrête donc et on passe à l'instruction suivante : **renvoyer** *res*. Cela a pour effet de terminer l'algorithme et de renvoyer la valeur 15.

Les autres constructions du langage. Nous n'allons pas présenter toutes les constructions pouvant être utilisés dans les langages de programmation modernes. D'abord, cela représenterait un travail volumineux, qui n'est pas l'objet de ce cours. Mais nous avons aussi une très bonne raison de ne présenter que la boucle **tant que** et l'affectation : cela définit un langage de programmation qui a déjà la même expressivité que nos langages de programmation favoris : tout algorithme pouvant être écrit dans un langage de programmation classique, peut être écrit sous une forme équivalente [4] comme un programme utilisant uniquement des instructions de déclaration de variables, d'affectations et de boucles **tant que**.

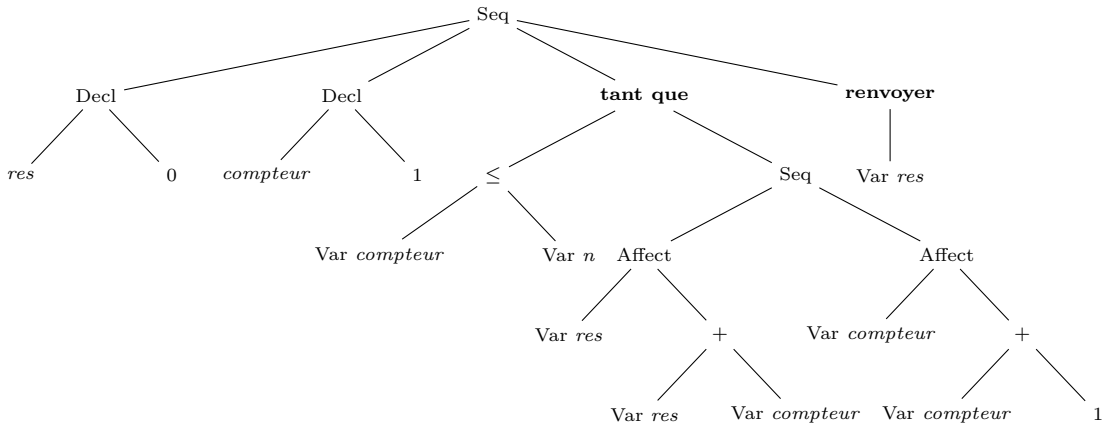
Ainsi, ces constructions suffisent en théorie pour analyser les algorithmes. Cependant, pour des raisons pratiques, nous nous permettrons l'usage des conditionnelles **si** ... **alors**, des appels de fonctions, et d'autres constructions si nécessaire.

Exercice 1.2. Montrer comment simuler une conditionnelle **si** ... **alors**, à l'aide d'une boucle **tant que**, d'une variable et de quelques affectations. C'est-à-dire, écrire des instructions équivalentes aux instructions suivantes, mais en utilisant comme seule structure de contrôle une boucle **tant que**.

instructions_pre
si condition **alors**
    instructions_corps
instructions_post

Ensuite, faire le même exercice pour la conditionnelle **si** ... **alors** ... **sinon**.

Une autre représentation de l'algorithme. Les instructions de l'Algorithme 1.1 sont présentées sous un format textuel auquel nous avons l'habitude, puisqu'il ressemble à celui que nous utilisons pour programmer. Nous avons aussi vu le graphe de flot de contrôle, qui représente le même algorithme. Il existe une autre représentation d'importance, puisqu'elle est utilisée par les compilateurs : l'*arbre de syntaxe abstraite*.



Chaque type de nœud correspond à une construction du langage de notre programmation.

- ▷ Seq représente la séquence d'instructions, et a pour enfants les arbres représentant chacune des instructions en séquence, dans l'ordre d'exécution.
- ▷ Decl représente la déclaration d'une variable, et a pour enfant gauche l'identifiant de la variable, et pour enfant droit une expression représentant la valeur initiale de la variable.

4. Il faudrait être rigoureux sur ce que nous entendons par équivalent dans ce contexte, mais ce serait l'objet d'autres cours, sur la théorie de la calculabilité et celle des langages de programmation.

▷ **tant que** représente la structure de contrôle de répétition conditionnelle éponyme, son enfant gauche représente l'expression conditionnelle, et son enfant droit le bloc d'instruction.

▷ Affect représente l'instruction d'affectation, ayant pour enfant gauche une expression représentant l'emplacement mémoire ciblé par l'affectation, et pour enfant droit une expression représentant la valeur à stocker en mémoire.

De la même façon nous pouvons définir des nœuds pour toutes les briques qui constituent ce langage. L'arbre de syntaxe abstraite révèle la structure hiéarchique des instructions du programme.

Un des enjeux de ce cours est de nous doter d'outils mathématiques pour représenter et manipuler les arbres. Ils sont en effet essentiels pour plusieurs étapes du processus de compilation : l'optimisation qui consiste pour partie en des transformations d'arbres, la vérification statique (par exemple pour les types) qui s'effectue par des algorithmes récursifs, la génération du programme en langage d'assemblage qui est une fonction d'une représentation arborescente du programme vers le langage cible. Quand à la construction de l'arbre de syntaxe abstraite depuis le texte du programme, c'est l'une des principales motivations du cours d'Automates et langages formels.

Nous retrouverons des arbres dans d'autres enseignements. Par exemple en logique, ils permettent de représenter non seulement les formules, mais aussi les preuves. En algorithmique, les arbres servent à structurer les données, afin d'optimiser des requêtes ou des opérations. Dans les systèmes d'exploitation, les arbres sont utilisés pour organiser les fichiers au sein des systèmes de fichiers, et les processus sont aussi organisés en arbre selon la relation de parent.

Exercice 1.3. Reprendre l'algorithme de l'Exercice 1.1, sous la forme de son arbre de syntaxe abstraite.

Analyse de l'algorithme. Nous avons vu que pour une entrée `n` égale à 5, le résultat de l'algorithme était 15. Nous pouvons également remarquer facilement en consultant le tableau d'exécution que ce résultat correspond au calcul $1+2+3+4+5$. Par généralisation, nous devinons que cet algorithme calcule la somme des n premier entiers, où n est la valeur passée en entrée à l'algorithme. Cependant deviner ne permet pas d'établir une certitude. Dans notre souci de produire des algorithmes fiables, nous souhaitons acquérir des outils permettant de décrire et de certifier par une preuve formelle le comportement de nos algorithmes.

En l'absence de ces outils, nous pouvons déjà nous convaincre que notre intuition ne nous a pas trompés, en remarquant qu'une exécution est toujours de la forme suivante :

itération	0	1	2	3	$\cdots$	n-1	n
n	n	n	n	n	$\cdots$	n	n
res	0	1	$1+2$	$1+2+3$	$\cdots$	$1+2+\cdots+(n-1)$	$1+2+\cdots+n$
compteur	1	2	3	4	$\cdots$	n	$n+1$
compteur $\leq n$	vrai	vrai	vrai	vrai	$\cdots$	vrai	faux

Ce tableau nous donne aussi une indication sur le temps d'exécution de l'algorithme. En effet, la variable *compteur* augmente de 1 à chaque itération de la boucle, la condition d'arrêt de la boucle implique donc qu'il y aura exactement n itérations de la boucle. Nous en tirons 2 informations cruciales :

▷ l'algorithme termine. Cette propriété (appelée *terminaison*) assure qu'il fournira toujours une réponse, quelle que soit la valeur de l'entrée du moment qu'elle est valide (ici qu'elle est un entier positif). C'est bien entendu une propriété nécessaire de tout algorithme [5] Il sera donc indispensable de toujours la vérifier, là aussi avec une preuve formelle.

▷ il exécute en tout $3+2n$ instructions élémentaires. Ceci nous donne une mesure de la performance de l'algorithme qui nous permettra par exemple de le comparer avec un autre algorithme calculant le même résultat. Cette mesure est appelée *complexité*, et est établie comme une fonction de l'entrée (ici n), ou de la taille de la représentation de l'entrée en mémoire.

5. Il faut mettre un bémol sur ce point, certains systèmes ont vocation à ne jamais être arrêtés, comme une horloge, et nécessite des algorithmes qui ne terminent pas. Cependant ce n'est pas l'objet de ce cours.

Exercice 1.4. En procédant de la même façon, établir un tableau représentant le comportement de l'algorithme de l'Exercice 1.1, et en déduire le nombre d'instructions élémentaires exécutées par l'algorithme pour deux entrées n et p.

Sur l'importance de vérifier la correction des algorithmes *Zune* est le nom donné par Microsoft à son logiciel de gestion et de lecture de fichiers musicaux dans les baladeurs du même nom. Le 31 décembre 2008 (année bissextile), une grande partie des Zune de première génération a subi une panne critique bloquant le démarrage du lecteur. Le problème venait du module de mise à jour, contenant une conversion d'un nombre de jours absolu compté depuis le 1er janvier 1980 en un couple (année, jour relatif dans l'année). Spécifiquement, cette conversion était réalisée avec l'Algorithme 1.2.

algorithmeDuZune($jour$)
Entrée: $jour$, un entier positif ;
Sortie: an, $jour$, deux entiers positifs.

```
1:  soit an = 1980
2:  tant que jour > 365 faire
3:      si estBissextile(an) alors
4:          si jour > 366 alors
5:              jour ← jour − 366
6:              an ← an + 1
7:      sinon
8:          jour ← jour − 365
9:          an ← an + 1
10: renvoyer (an, jour)
```

Algorithme 1.2 – L'algorithme (erroné) de conversion du balladeur Zune.

Voici des exemples de conversions telle que souhaitées :

jour	date représentée	résultat attendu
1	1 janvier 1980	(1980,1)
380	14 janvier 1981	(1981,14)
4021	3 janvier 1991	(1991,3)
10592	30 décembre 2008	(2008,365)
10593	31 décembre 2008	(2008,366)

Cependant, pour l'entrée 10593, l'algorithme ne termine pas car après un certain nombre d'itération de la boucle, les valeurs des variables vérifient :

- ▷ $an = 2008$, $jour = 366$;
- ▷ estBissextile(an) est vrai ;
- ▷ $jour > 366$ est faux.

Ce cas n'est pas prévu dans l'algorithme, les valeurs ne sont alors plus jamais modifiées et la boucle est répétée indéfiniment.

Un des objectifs de ce cours sera donc d'introduire des outils permettant de raisonner rigoureusement sur les algorithmes, afin d'être capable de vérifier la correction d'un algorithme, ou le cas échéant d'en détecter les erreurs.

Sur l'importance de mesurer la performance des algorithmes L'Algorithme 1.3 calcule la somme des n premiers entiers impairs. Cet algorithme effectue $2n$ additions et n multiplications.

sommeEntiersImpairsLent(n)
Entrée: n, un entier positif ;
Sortie: *somme*, la somme des n premiers entiers impairs.
1: **soit** $somme = 0$
2: **pour** i de 1 à n **faire**
3: $\quad somme \leftarrow somme + (2 \times i - 1)$
4: **renvoyer** *somme*

Algorithme 1.3 – Un algorithme calculant la somme d'entiers impairs.

Il est en fait possible d'écrire un algorithme plus performant en remarquant que la somme des n premiers entiers impairs est égale à n^2. Vérifions que pour tout $n \in \mathbb{N}$, $\sum_{i=1}^{n}(2i-1) = n^2$, par récurrence sur n.

▷ Pour $n = 0$, la somme est vide, donc a pour valeur l'élément neutre pour l'addition, c'est-à-dire 0, qui vaut bien 0^2.

▷ Soit $n \in \mathbb{N}$, et supposons que $n^2 = \sum_{i=1}^{n}(2i-1)$. Alors

$$\begin{aligned}
\sum_{i=1}^{n+1}(2i-1) &= 2(n+1) - 1 + \sum_{i=1}^{n}(2i-1) && \text{(définition de } \textstyle\sum\text{)} \\
&= 2n + 1 + \sum_{i=1}^{n}(2i-1) && \text{(simplification)} \\
&= 2n + 1 + n^2 && \text{(hypothèse de récurrence)} \\
&= (n+1)^2 && \text{(factorisation)}
\end{aligned}$$

Nous en déduisons l'Algorithme 1.4 n'effectuant qu'une seule multiplication, et qui est donc dramatiquement plus efficace.

sommeEntiersImpairsRapide(n)
Entrée: n, un entier positif ;
Sortie: la somme des n premiers entiers impairs.
1: **renvoyer** $n \times n$

Algorithme 1.4 – Un deuxième algorithme calculant la somme d'entiers impairs, plus efficace.

1.2 Plan de l'enseignement

Au cours de cet enseignement, nous allons introduire des outils mathématiques particulièrement utiles en informatique, et que nous réutiliserons dans les autres enseignements, typiquement en algorithmique, langages formels, logique, compilation.

1. Dans un premier temps, nous étudierons la notion d'ordre bien-fondé, utile pour démontrer la terminaison des algorithmes récursifs.
2. Ensuite nous introduirons les définitions inductives, qui permettent d'écrire des définitions mathématiques similaires aux définitions de certaines structures de données dans les langages de programmation (particulièrement les langages dits fonctionnels). En association avec ces définitions inductives, nous verrons comment définir des fonctions et écrire des preuves inductives. Ce sont de puissants outils permettant d'écrire des algorithmes récursifs et démontrer leur correction. Ils ont aussi un rôle primordial dans l'analyse de systèmes formels comme ceux étudiés en théorie des langages ou en logique.
3. Nous aborderons ensuite les preuves de correction d'algorithmes basés sur la logique de Hoare, consistant à trouver et prouver des formules logiques décrivant l'état d'exécution du programme. Les raisonnements fondant cette technique sont applicables dans le cadre de la recherche de bugs dans les programmes. Ils seront aussi essentiels pour comprendre les algorithmes parfois subtils qui seront étudiés en algorithmique.

4. Nous terminerons en introduisant des outils d'analyse de la complexité des algorithmes.

2 Terminaison d'algorithmes

Nous allons dans cette section, introduire des outils mathématiques utiles pour l'étude de la *terminaison* des algorithmes, c'est-à-dire décider si un algorithme donné termine pour toute entrée valide.

2.1 Un problème difficile

L'Algorithme 2.1 énumère les termes de la suite de Syracuse à partir d'un entier n et s'arrête lorsqu'il rencontre un terme égal à 1.

syracuse(n)
Entrée: un entier $n > 0$.
1: **soit** $syr = n$
2: **tant que** $syr \neq 1$ **faire**
3: | **si** syr est pair **alors**
4: | | $syr = syr/2$
5: | **sinon**
6: | | $syr = 3 \times syr + 1$

Algorithme 2.1 – L'algorithme de la suite de Syracuse. Ici nous terminons le calcul des termes de la suite dès que nous trouvons 1, les termes suivants seront alors $4, 2, 1, 4, 2, 1, \ldots$

En faisant tourner cet algorithme sur quelques valeurs, nous constatons que la variable *syr* finit toujours par prendre la valeur 1, et donc que l'algorithme termine. La conjecture de Syracuse, aussi appelée conjecture de Collatz, prétend que la suite de Syracuse, partant de n'importe quel entier strictement positif n, finit nécessairement par atteindre la valeur 1. Quelques exemples le suggère mais cela ne constitue pas un preuve : puisque l'ensemble des entiers est infini, il existera toujours un entier que nous n'avons pas essayé, et qui est susceptible d'être tel que la suite n'atteint jamais 1 !

Malheureusement, il n'existe pour le moment aucune théorie mathématique permettant de prouver cette conjecture : il est impossible à ce jour de dire si cet algorithme termine ou non sur toutes ses entrées valides. Pire, il est possible de démontrer qu'il n'existe pas de méthode générale (et donc pas d'algorithme !) pour prouver la terminaison d'un algorithme. Donnons une idée de la raison de cette impossibilité, en raisonnant par l'absurde : supposons qu'il existe un algorithme, appelons-le terminaison qui prenne en paramètre un algorithme p et qui retourne vrai si p termine sur toute entrée. Considérons alors l'Algorithme 2.2, qui n'a pas de paramètres d'entrée, c'est-à-dire qu'il a une seule entrée : le 0-tuple ().

bizarre()
1: **tant que** terminaison(bizarre) **faire**
2: | ▷ *rien*

Algorithme 2.2 – Un algorithme hypothétique, qui ne peut ni terminer, ni ne pas terminer.

- ▷ Si la condition terminaison(bizarre) s'évalue en vrai, par définition de terminaison c'est que bizarre termine. Mais dans ce cas, la condition du **tant que** est toujours vraie, et donc bizarre ne termine pas, ce qui est une contradiction.
- ▷ Sinon la condition terminaison(bizarre) s'évalue en faux, par définition de terminaison c'est que bizarre ne termine pas. Mais dans ce cas, la condition du **tant que** est toujours fausse, et donc bizarre termine. C'est encore une contradiction.

Ainsi, nous obtenons dans chaque cas une contradiction, notre hypothèse était donc fausse : il n'existe pas un algorithme tel que terminaison.

Nous devons donc rester modeste, et développer des outils permettant de démontrer la terminaison de la plupart des algorithmes qui nous intéressent, c'est-à-dire ceux que nous étudions dans

les autres enseignements ou que nous sommes susceptibles d'utiliser dans un contexte professionnel.

À ce titre, la technique la plus répandue est celle du *variant bien fondé*. Pour prouver la terminaison, nous déterminerons une expression bien choisie, qui diminue systématiquement à chaque itération de boucle. Cette expression est nommé un *variant*. Il nous faut préciser ce que nous entendons par diminution : particulièrement nous voulons interdire la possibilité de diminuer indéfiniment. Par exemple, la suite réelle définie par $u_0 = 1$, $u_{n+1} = u_n/2$ diminue strictement à chaque terme, donc diminue indéfiniment. Nous devons donc imposer que la valeur du variant soit prise dans un espace muni d'un ordre interdisant les suites infinies décroissantes, ce qu'on appelle un ordre *bien fondé*. Avant de formaliser ces notions, quelques rappels sont nécessaires.

2.2 Rappels sur les relations et les ordres

Soient E un ensemble E et $n \geq 0$ un entier. E^n est l'ensemble des n-tuples d'éléments de E. Une *relation n-aire* (ou d'*arité n*) sur E est un sous-ensemble de E^n.

Exemple de l'ordre sur les entiers naturels. Dans ce cas, $E = \mathbb{N}$ et $n = 2$. La relation $<$ est l'ensemble des couples (a, b) avec $a, b \in \mathbb{N}$ tels qu'il existe $c \in \mathbb{N} \setminus \{0\}$ pour lequel $b = a + c$.

Exemple de la congruence. Soit $k \in \mathbb{N} \setminus \{0\}$. Nous prenons $E = \mathbb{N}$ et $n = 2$, et définissons la relation R, d'arité 2, par $(a, b) \in R$ si $a \equiv b[k]$. C'est la relation de congruence modulo k.

Exemple de l'égalité. Quelque soit l'ensemble E, la relation d'égalité est une relation d'arité 2, qui contient précisément les couples (x, x) pour tout $x \in E$.

Exemple des prédicats sur un ensemble. Si $n = 1$, E^1 est isomorphe à E, donc un sous-ensemble de E^1 correspond à un sous-ensemble de E. Une relation d'arité 1 est donc essentiellement un sous-ensemble. Le terme *relation*, bien que correct, semble inapproprié lorsque l'arité est 1, et nous privilégions alors le synonyme *prédicat*. Des exemples sur les entiers naturels seraient les prédicats *être pair* ou *être premier*.

Exemple d'une relation d'intermédiarité. Si $E = \mathbb{R}^2$ est l'ensemble des points du plan Cartésien, nous pouvons définir une relation R d'arité 3, en posant $(p, q, r) \in R$ si q appartient au segment $[p, r]$ (autrement dit, s'il existe $\lambda \in [0, 1]$ tel que $q = \lambda p + (1 - \lambda r)$). Par exemple nous avons $((3, 2), (1.5, 0), (0, -2)) \in R$.

Notation Si R est une relation d'arité n, nous notons $R(a_1, \ldots, a_n)$ si $(a_1, \ldots, a_n) \in R$. Pour les relations d'arité 2, il est courant d'utiliser un symbole, comme par exemple $<$ (pour des relations d'ordre) ou $\equiv$ (pour des relations d'équivalence), en position infixe. Ainsi si R est une relation d'ordre, à laquelle est associée le symbole $\prec$, la notation $a \prec b$ signifie $(a, b) \in R$.

Propriétés des relations binaires Les relations binaires que nous considérons ont souvent des propriétés qui les rendent intéressantes. Une relation binaire R sur un ensemble E est ainsi dite :

▷ *réflexive* si

$$\text{pour tout } x \in E,\ R(x, x)\,;$$

▷ *symétrique* si

$$\text{pour tout } x, y \in E, \text{ si } R(x, y) \text{ alors } R(y, x)\,;$$

▷ *antisymétrique* si

$$\text{pour tout } x, y \in E, \text{ si } R(x, y) \text{ et } R(y, x) \text{ alors } x = y\,;$$

▷ *transitive* si

$$\text{pour tout } x, y, z \in E, \text{ si } R(x, y) \text{ et } R(y, z) \text{ alors } R(x, z).$$

2.3 Relations d'ordre bien fondées

Définition 2.1 (Relation de pré-ordre)**.** Soit E un ensemble. Une relation binaire R sur E est une relation de *pré-ordre* si elle est réflexive et transitive. (E, R) est alors appelé un *ensemble pré-ordonné*.

Définition 2.2 (Relation d'ordre)**.** Soit un ensemble E. Une relation binaire R sur E est une relation d'ordre sur E (également appelée *ordre*) si elle est réflexive, antisymétrique et transitive. (E, R) est alors appelé un *ensemble ordonné*.

Par définition, un *ordre* est donc un *préordre* antisymétrique. Notons qu'il s'agit d'ordres *partiels* : tous les éléments ne sont pas forcément comparables deux-à-deux. Si tous les éléments sont comparables, nous appelons cela un ordre *total*.

Exercice 2.1. Vérifier que la relation $\leq$ sur les entiers est une relation d'ordre : $n \leq p$ s'il existe $q \in \mathbb{N}$ tel que $p = n + q$. Montrer que cet ordre est total : pour tout $p, q \in \mathbb{N}$, $(p \leq q) \vee (q \leq p)$.

Exercice 2.2. Montrer que la relation de divisibilité $\mid$ sur $\mathbb{N} \setminus \{0\}$ est une relation d'ordre partiel, avec $p \mid n$ s'il existe $q \in \mathbb{N}$ tel que $p \times q = n$.

Définition 2.3 (Élément minimal)**.** Soit $(E, \preceq)$ un ensemble ordonné et F une partie non vide de E. Un élément $x \in F$ est un *élément minimal* de F quand aucun élément de F n'est strictement plus petit que x : pour tout $y \in F$, $y \preceq x$ implique $x = y$.

Il ne faut pas confondre élément minimal et plus petit élément. Un *plus petit élément* de F est un élément plus petit que tous les autres : x est plus petit élément de F si pour tout $y \in F$, $x \leq y$. Tous les autres éléments sont plus grands que lui. Par constraste, un élément minimal n'a pas d'élément plus petit que lui. Les deux notions coïncident uniquement pour les ordres totaux. Par exemple, pour la relation de divisibilité, les nombres premiers sont les éléments minimaux de $\{n \in \mathbb{N} \mid n \geq 2\}$; il n'y a pas de plus petit élément.

Définition 2.4 (Ordre bien fondé)**.** Soit $(E, \preceq)$ un ensemble ordonné, on dit que $(E, \preceq)$ est *bien fondé* (ou que $\preceq$ est un *bon ordre* sur E) si toute partie non vide de E admet un élément minimal.

Exemple 2.5. L'ensemble ordonné $(\mathbb{N}, \leq)$ est bien fondé, mais $(\mathbb{Z}, \leq)$ ne l'est pas puisque $\mathbb{Z}$ n'admet pas d'élément minimal.

Comme informaticiens, notre intérêt dans les ordres bien-fondés s'explique principalement par le théorème suivant, qui caractérise les ordres bien-fondés comme précisément ceux dans lesquels il est impossible de réaliser une *descente infinie*, pour employer un vocabulaire peu rigoureux.

Théorème 2.6. L'ordre $(E, \preceq)$ est bien fondé si et seulement si il n'existe aucune suite infinie décroissante d'éléments distincts deux-à-deux dans E.

Démonstration. Supposons que $(E, \preceq)$ est bien fondé, et prouvons par contradiction qu'il n'existe aucune suite infinie décroissante dans E. Supposons qu'il existe une telle suite $\ldots \prec u_n \prec \ldots \prec u_2 \prec u_1$. Alors, l'ensemble $X = \{u_i, i \geq 1\}$ serait un sous-ensemble de E ne contenant évidemment aucun élément minimal, ce qui contredit le fait que $(E, \preceq)$ est bien fondé.

Nous prouvons la réciproque par contraposition : nous supposons que $(E, \preceq)$ n'est pas bien fondé, et nous construisons une suite infinie décroissante dans E. Si $\preceq$ n'est pas bien fondé, alors il existe un ensemble $X \subseteq E$ non vide, sans élément minimal. Soit $u_1 \in X$, alors, puisque x_1 n'est pas minimal, il existe $u_2 \in X$ tel que $u_2 \prec u_1$. Puis, toujours parce que X n'admet pas d'élément minimal, u_2 n'est pas minimal, il existe donc $u_3 \in X$ tel que $u_3 \prec u_2$. Nous pouvons donc, par récurrence, construire une suite $(u_n)_{n\in\mathbb{N}^*}$ infinie décroissante d'éléments distinctes deux-à-deux. Les éléments sont en effet distincts, puisque par transitivité, pour tout $i < j$, $u_j \prec u_i$. □

Un exemple intéressant. L'ordre lexicographique est un ordre sur les k-tuples d'un ensemble lui-même ordonné. Le terme lexicographie fait référence au dictionnaire, puisque cet ordre se construit comme l'ordre des mots d'un dictionnaire, si du moins le dictionnaire n'avait que des mots de même longueur. La comparaison entre deux tuples consiste donc à comparer la 1re lettre, puis en cas d'égalité, la 2e, et ainsi de suite. Nous formalisons cette idée.

Nous commencerons par le cas des couples, c'est-à-dire les 2-tuples.

Définition 2.7. Soient $(A, \prec_A)$ et $(B, \prec_B)$ deux ensembles ordonnés. Alors $(A \times B, \preceq_{A\times B})$ est un ensemble ordonné, avec : pour tout $a_1, a_2 \in A$, $b_1, b_2 \in B$,

$$(a_1, b_1) \prec_{A\times B} (a_2, b_2) \text{ ssi } a_1 \prec_A a_2 \text{ ou } (a_1 = a_2 \text{ et } b_1 \prec_b b_2).$$

Exercice 2.3. Écrire la preuve qu'il s'agit bien d'une ensemble ordonné (montrer l'anti-symétrie et la transitivité).

Ainsi pour comparer deux couples, nous pouvons comparer les premières projections, puis en cas d'égalité les deuxièmes projections.

Proposition 2.8. Si $(A, \preceq_A)$ et $(B, \preceq_B)$ sont bien fondés, alors $(A \times B, \preceq_{A\times B})$ est bien fondé.

Démonstration. Soit X une partie de $A \times B$, nous devons prouver que X possède un élément minimal (a_0, b_0). Nous pouvons pour a_0 prendre l'élément minimal apparaissant dans une des paires de X. Formellement, définissons $\pi_1(X) := \{a \in A \mid \exists b \in B, (a, b) \in X\}$. $\pi_1(X) \subseteq A$ et $(A, \prec_A)$ est bien-fondé, donc $\pi_1(X)$ possède un élément minimal a_0.

Ensuite, pour b_0, nous voulons le choisir minimal parmi tous les éléments b tels que $(a_0, b) \in X$. Formellement, soit $\pi_1^{-1}(a_0) := \{(a, b) \in X \mid a = a_0\}$. $\pi_1^{-1}(a_0)$ est non-vide par définition. Sa deuxième projection $\pi_2(\pi_1^{-1}(a_0)) := \{b \in B \ : \ (a_0, b) \in \pi_1^{-1}(a_0)\}$ est donc non-vide, et possède une élément minimal b_0, par le fait que $(B, \prec_B)$ est bien-fondé.

Il nous reste à démontrer que (a_0, b_0) est un élément minimal de F pour $\prec_{A\times B}$. Soit $(a, b) \in X$ une paire arbitraire de X. Notamment nous savons que $a \in \pi_1(X)$, et comme a_0 est minimal dans cet ensemble $a \not\prec_A a_0$. Du coup, trois cas sont possibles.

- ▷ Si a_0 et a ne sont pas comparables par $\prec_A$, en particulier cela implique que $a \neq a_0$. Alors, selon la définition de $\prec_{A\times B}$, (a, b) et (a_0, b_0) sont incomparables.
- ▷ Si $a_0 \prec_A a$, alors par définition $(a_0, b_0) \prec_{A\times B} (a, b)$.
- ▷ Si $a_o = a$, alors b et b_0 par construction sont tous les deux éléments de $\pi_2(\pi_1^{-1}(a_0)))$. Puisque b_0 est minimal dans cet ensemble, $b \not\prec_B b_0$. Si $b = b_0$, alors $(a, b) = (a_0, b_0)$. Si $b_0 \prec_B b$ alors $(a_0, b_0) \prec_{A\times B} (a, b)$, et enfin si b et b_0 sont incomparables, par définition de $\prec_{A\times B}$, (a, b) et (a_0, b_0) sont aussi incomparables.

Ainsi, quelque soit le cas, nous n'avons jamais $(a, b) \prec_{A\times B} (a_0, b_0)$, et donc (a_0, b_0) est bien un élément minimal de F. □

Pour généraliser cela aux tuples de longueur arbitraire, il suffit de répéter cette construction par récurrence.

Définition 2.9 (Ordre lexicographe)**.** Soient $(A_1, \prec_{A_1}), \ldots, (A_k, \prec_{A_k})$, k ensembles ordonnés. L'ordre lexicographique sur $A_1 \times \ldots \times A_k$ est $\prec_{A_1 \times (A_2 \times (\ldots \times A_k))}$.

En notant simplement $<_{\text{prec}}$ l'ordre ainsi obtenu, nous avons alors que $(a_1, \ldots, a_k) <_{\text{prec}} (b_1, \ldots, b_k)$ si $a_1 \prec_{A_1} b_1$ ou $(a_1 = b_1$ et $(a_2 \prec_{A_2} b_2$ ou $(a_2 = b_2$ et $(\ldots))))$. Autrement dit, nous cherchons le premier indice tel que a_i et b_i sont distincts, et nous utilisons leur comparaison pour comparer les deux tuples (plus petit, plus grand ou incomparable).

Nous utiliserons en particulier l'ordre lexicographique sur $\mathbb{N}^k$.

Exemple 2.10. Nous pouvons vérifier simplement que les inégalités suivantes sont vraies :

$$(1, 2) \leq_{\text{lex}} (1, 3) \leq_{\text{lex}} (2, 1) \leq_{\text{lex}} (2, 10) \leq_{\text{lex}} (10, 2).$$

Proposition 2.11. Soient k ensembles ordonnés $(A_1, \prec_{A_1}), \ldots, (A_k, \prec_{A_k})$, pour $k \geq 2$. L'ordre lexicographique $<_{\text{lex}}$ sur $A_1 \times A_2 \times \ldots \times A_k$ est bien fondé.

Démonstration. Par récurrence sur $k \geq 2$, sachant que pour $k = 2$ il s'agit de la Proposition 2.8. Soit donc $k \geq 3$, et supposons que l'ordre lexicographique $<_{k-1}$ sur $A_2 \times A_3 \times \ldots \times A_k$ est bien-fondé. Alors en appliquant Proposition 2.8 sur $(A_1, \prec_{A_1})$ et $(A_2 \times A_3 \times \ldots \times A_k, <_{k-1}$, nous obtenons que $(A_1, \ldots, A_k, <_{\text{lex}})$ est bien fondé. □

2.4 Preuve de terminaison des boucles

Un algorithme sans boucle (et sans appel de fonctions, notamment sans récursion) termine toujours, puisque chaque instruction n'est effectuée qu'une seule fois. Ainsi dans notre langage restreint aux affectations et aux boucles, le problème de la terminaison ne se pose que pour les boucles. Pour prouver qu'une boucle termine, la technique usuelle est celle du variant. Pour présenter cette technique, il nous faut d'abord définir la notion d'*état* de l'exécution d'un algorithme.

Un exemple. Considérons l'Algorithme 2.3. La post-condition indique que l'algorithme calcule la division de n par m, donc pour être correctement définie, l'entier m doit-être non nul, comme spécifié par la pré-condition.

divisionNat(n, m)
Entrée: n, un entier naturel, m, un entier naturel non nul.
Sortie: le résultat de la division entière de n par m

```
1: soit quotient = 0
2: tant que n ≥ m faire
3:     quotient ← quotient + 1
4:     n ← n − m
5: renvoyer quotient
```

Algorithme 2.3 – Un algorithme pour la division entière.

L'état d'exécution d'un algorithme. L'état d'exécution d'un algorithme est une description instantanée des valeurs des variables de l'algorithme, au cours de son exécution. Il s'agit d'une fonction (éventuellement partielle) associant à chaque variable une valeur, qui est son contenu à cet instant de l'exécution. Dans notre exemple de division entière, un état au commencement de l'exécution de l'algorithme sera décrit par exemple par le triplet

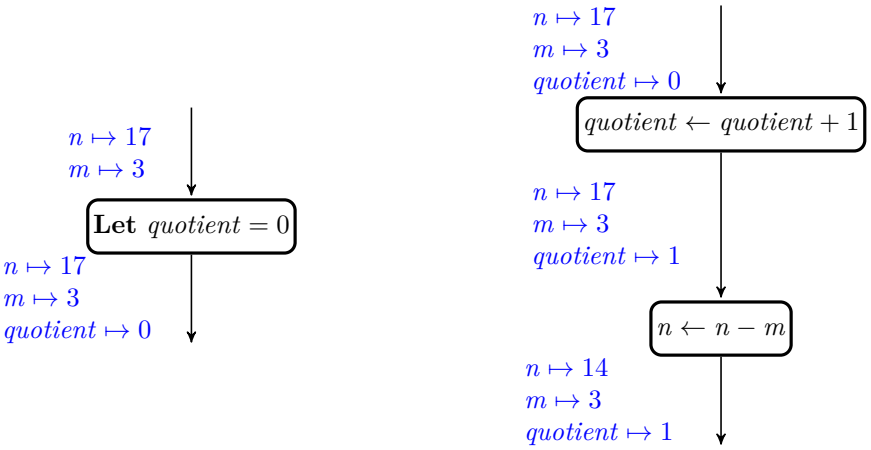
$$e = (n \mapsto 17, m \mapsto 3)$$

Il s'agit donc ici d'une exécution avec les arguments divisionNat$(17, 3)$. Nous sommes avant l'exécution de la première instruction, qui est une déclaration de la variable *quotient*, qui n'est donc pas encore dans le domaine de l'environnement.

amU Faculté des sciences Aix Marseille Université

Puisqu'un état est une fonction, on peut accéder directement à la valeur d'une variable i dans l'état e, en le notant $e(i)$. Ainsi, pour l'état décrit ci-dessus, on a $e(n) = 17$.

L'exécution d'une instruction fait passer d'un état à un autre comme représenté dans les deux exemples ci-dessous :



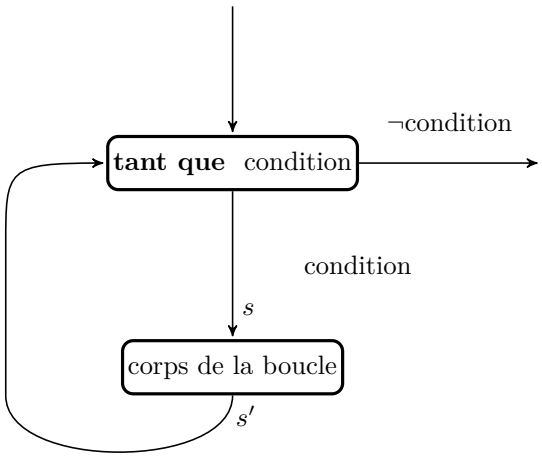
Définition 2.12. Étant donnés deux états e et e' et bloc un bloc d'instructions, on note $\langle e \rangle$`bloc`$\langle e' \rangle$ si l'exécution de bloc depuis l'état e produit l'état e'.

Terminaison des boucles. La notion d'état est essentielle pour analyser le comportement des boucles, puisque la différence existant entre deux itérations successives d'une boucle est l'état de l'exécution. Si l'état restait le même à chaque itération, ou revenait à un état antérieur, la boucle ne terminerait pas. Pour montrer la terminaison d'une boucle, il est donc nécessaire de considérer comment l'état change à chaque itération, ce qui nous amène à cette définition :

Définition 2.13 (Variant)**.** Considérons un algorithme contenant une boucle **tant que**. Un *variant* de la boucle est une fonction (possiblement partielle) V des états de l'algorithme vers un ordre bien-fondé $(E, \preceq)$, telle que pour tout état s atteignable en début de boucle, l'état s' obtenu après une itération de la boucle depuis l'état s satisfait :

$$V(s) \text{ et } V(s') \text{ sont définis et } V(s') \prec V(s).$$

La figure ci-dessous illustre les positions des états s et s' de la définition précédente :



Ainsi, la valeur d'un variant décroit dans l'ordre bien fondé par itération de la boucle. Une boucle se répétant indéfiniment engendrerait donc une suite infinie décroissante dans l'ordre bien-fondé, ce qui est contradictoire et nous assure que la boucle termine, comme nous allons le démontrer.

Théorème 2.14. Si une boucle admet un variant, alors elle termine pour tout exécution de l'algorithme.

Démonstration. Supposons qu'une boucle admette un tel variant V dans $(E, \preceq)$ bien-fondé, mais qu'elle ne termine pas toujours. Considérons une exécution qui ne termine pas. Nommons $s_0, s_1, \ldots, s_n, \ldots$ l'infinité d'états de l'algorithme, apparaissant au début de chaque itération de la boucle lors de cette exécution. Ainsi chaque état s_i est par construction atteignable par l'algorithme, et s_{i+1} est l'état produit par l'exécution de la boucle depuis l'état s_i. Par définition de variant, $V(s_{i+1}) \preceq V(S_i)$ pour tout $i \in \{0, 1, \ldots\}$. Ainsi la suite $V(S_0), V(S_1), \ldots$ est décroissante dans $(E, \preceq)$, ce qui est une contradiction puisque $(E, \preceq)$ est bien-fondé. □

En conséquence, pour prouver la terminaison d'une boucle, nous disposons maintenant d'une technique consistant en l'exhibition d'un variant pour cette boucle. Il nous faudra

- ▷ choisir un ordre bien-fondé adéquat $(E, \preceq)$,
- ▷ trouver le variant, c'est-à-dire la fonction V des états dans l'ordre bien fondé,
- ▷ démontrer que pour tout état accessible s en début d'itération $V(s)$ est bien défini, et que l'état s' produit par le corps de la boucle depuis s vérifie que $V(s')$ est bien défini, avec $V(s') \preceq V(s)$.

En pratique, selon les cas, n'importe laquelle de ces trois étapes peut entraîner des difficultés, mais c'est souvent le choix de V qui demande le plus de réflexion. Nous devons considérer qu'il nous faut capturer dans le variant l'idée que l'algorithme *progresse*. Le variant V est alors une mesure de ce progrès. Ce progrès doit avoir une fin, peut-être une valeur minimale ou maximale, qu'il faudra prendre en compte pour le choix de l'ordre bien-fondé.

Dans les cas les plus simples, nous pourrons utiliser l'ordre usuel des entiers naturels $(\mathbb{N}, \leq)$, qui est bien-fondé. Il convient évidemment lorsque chaque itération de la boucle diminue une valeur entière bien identifiable (un indice, la longueur d'une liste, le nombre d'élément d'un ensemble, la hauteur d'un arbre), et que cette valeur ne peut pas être négative. Dans certains cas, nous aurons une valeur qui augmente, mais dont la valeur ne peut dépasser une borne fixe. Dans ce cas, il suffira de prendre la différence entre la borne et cette valeur comme variant, mesurant ainsi le rapprochement de la valeur à la borne.

Les ordres lexicographiques sont utiles lorsque plusieurs valeurs varient, mais pas nécessairement de façon stricte ou même monotone. Enfin, nous introduirons par la suite les définitions inductives, qui fournissent des ordres bien-fondés appropriés pour définir des variants sur les boucles manipulant des listes ou des arbres.

Un exemple d'application. Prouvons que l'Algorithme 2.3, calculant la division entière, termine. Il suffit de prouver que son unique boucle termine. Remarquons d'abord que la variable m ne subit jamais d'affectation, sa valeur restera donc la même dans tous les états atteignable. Au contraire, n décroit à chaque itération, ce qui suggère d'utiliser sa valeur comme variant, d'autant plus que la condition de sortie indique que n ne peut décroitre au-delà de m qui est fixe. Considérons donc la fonction $V : s \mapsto s(n)$ pour tout état s où $s(n)$ est définie et positive ou nulle. Ainsi, cette fonction est à valeur dans $(\mathbb{N}, \leq)$.

Soit s valide en début de boucle et s' tel que

$$\langle s \rangle \, quotient \leftarrow quotient + 1; n \leftarrow n - m \langle s' \rangle.$$

Alors $s'(n) = s(n) - m < s(n)$, puisque m est strictement positif. De plus, par la condition de continuation de la boucle, nous savons que $s(n) \geq s(m)$, et donc $s(n) > s(n') \geq 0$. Nous en déduisons que $V(s)$ et $V(s')$ sont bien définis, et que $V(s') \leq V(s)$, V est un variant de la boucle. Par le Théorème 2.14, cette boucle termine, ce qui implique que l'algorithme de division entière termine aussi.

Exercice 2.4. Montrer que la boucle de l'Algorithme 2.4 termine.

insertion$(t[1..n], i)$
Entrée: t un tableau d'entiers de longueur n, $i \in \{1, 2, \ldots, n\}$ un indice du tableau.
1: **tant que** $i > 1$ et $t[i] < t[i-1]$ **faire**
2: $\quad t[i] \leftrightarrow t[i-1]$ ▷ *échange du contenu des 2 cases*
3: $\quad i \leftarrow i - 1$

Algorithme 2.4 – Un algorithme utilisé par le tri par insertion.

2.5 Preuves de terminaisons des algorithmes récursifs

Les algorithmes récursifs. Nous avons dès le début admis que nos algorithmes peuvent faire des appels à d'autres algorithmes, ce qui est une capacité essentielle en pratique pour pouvoir écrire de larges programmes. Par exemple, l'Algorithme 2.5 fait appel à un algorithme de division d'entiers naturels pour calculer une *moyenne entière* : la valeur renvoyée par l'algorithme auxiliaire, sur des entrées fournies par l'algorithme maître, est utilisée par l'algorithme maître pour ses propres besoins.

moyenne(x, y)
Entrée: x,y deux entiers positifs.
Sortie: la moyenne entière de x et de y.
1: **renvoyer** divisionNat$(x + y, 2)$ ▷ *un appel à un autre algorithme*

Algorithme 2.5 – Un algorithme faisant appel à un autre algorithme (l'Algorithme 2.3).

L'algorithme appelé est arbitraire, en particulier un algorithme est autorisé à s'appeler lui-même. Un algorithme est *récursif* si ses instructions contiennent au moins un appel à lui-même. Ce mécanisme permet l'implémentation des fonctions récursives que nous connaissez en mathématiques. Par exemple, la fonction associant un nombre n à sa factorielle est définie par pour tout $n \in \mathbb{N}$ par :

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \times fact(n-1) & \text{si } n > 0 \end{cases}$$

et correspond à l'Algorithme 2.6.

factorielle(n)
Entrée: n un entier positif ;
Sortie: la factorielle de n.
1: **si** $n = 0$ **alors**
2: $\quad$ **renvoyer** 1
3: **renvoyer** $n \times$ factorielle$(n-1)$

Algorithme 2.6 – L'algorithme récursif pour le calcul de la factorielle.

Les appels récursifs ne sont valides que si les arguments donnés lors de l'appel récursif vérifient ses pré-conditions. Une étape importante de la peuve de correction d'un algorithme récursif est donc de montrer que pour toute entrée s satisfaisant les pré-conditions, si s' est une entrée telle que l'appel récursif d'entrée s' est provoqué lors de l'exécution de l'algorithme sur l'entrée s, alors s' vérifie aussi les précondition.

Exemple 2.15. Pour l'Algorithme 2.6 de la factorielle, vérifier que les appels récursifs satisfont les pré-conditions demande de prouver que $n-1$ est un entier positif lorsque l'appel de la dernière ligne est exécuté. C'est effectivement le cas, puisque la condition du **si** ... **alors** filtre le cas $n = 0$, nous laissant dans le cas $n > 0$. L'appel récursif factorielle$(n-1)$ vérifie donc bien la pré-condition.

Définition 2.16. Soit un algorithme récursif $\mathcal{A}$, et soit S l'ensemble des entrées de $\mathcal{A}$ vérifiant les pré-conditions. L'algorithme $\mathcal{A}$ est *validement défini* si

pour toute entrée $s \in S$, si l'exécution de $\mathcal{A}(s)$ provoque l'appel $\mathcal{A}(s')$, alors $s' \in S$.

Les variants bien-fondés pour les algorithmes récursifs. Les algorithmes récursifs posent le même problème de terminaison que les algorithmes utilisant des boucles. Si chaque résolution d'un appel à l'algorithme provoque un autre appel à l'algorithme, quelle garantie avons-nous que l'exécution va terminer ? Afin d'apporter une telle garantie, nous adaptons la technique du variant bien-fondé au cas des algorithmes récursifs.

Nous ne traitons ici que le cas de la récursivité simple : l'algorithme s'appelle lui-même. La technique peut être étendue pour des récursions plus complexes entre plusieurs fonctions. Comme pour le cas des algorithmes itératifs, nous prouverons la terminaison des algorithmes récursifs en exhibant un variant. Dans le cas d'un algorithme récursif $\mathcal{A}$, le variant V sera une fonction des entrées de l'algorithme, à valeur dans un ensemble bien fondé $(E, \preceq)$. La condition de décroissance s'exprime sur les appels récursifs : la valeur des entrées décroit dans l'ordre bien-fondé à chaque appel récursif.

Définition 2.17. Un *variant* d'un l'algorithme récursif $\mathcal{A}$ est une application V de l'ensemble S des états d'entrée (c'est-à-dire qu'elle respecte les pré-conditions) de l'algorithme vers un ensemble bien fondé $(E, \preceq)$ telle que :

pour tous s et $s' \in S$, si l'exécution de $\mathcal{A}(s)$ provoque l'appel $\mathcal{A}(s')$, alors $V(s') \prec V(s)$.

Théorème 2.18. Si un algorithme récursif sans boucle et validement défini admet un variant, alors il termine.

Démonstration. Supposons qu'un algorithme récursif admette un variant V mais ne termine pas. Puisqu'il ne possède pas de boucle, il existe une entrée s_0 telle que l'exécution de l'algorithme sur s_0 engendre une suite infinie d'appels récursifs sur les entrées successives $s_0, s_1, s_2, \ldots, s_n, \ldots$. Par récurrence sur n, parce que l'algorithme est validement défini, s_n vérifie les pré-conditions pour tout $n \in \mathbb{N}$. Puisque V est un variant, on a alors $V(s_0) > V(s_1) > V(s_2) \ldots > V(s_n) > \ldots$ ce qui est impossible puisque $(E, \prec)$ est bien fondé, donc n'admet pas de suite infinie strictement décroissante, d'après le Théorème 2.6. □

Si l'algorithme récursif contient une boucle, il faut montrer que la boucle termine d'une part, et que la récursion termine d'autre part, à chaque fois en exhibant un variant.

Exemple 2.19. Prouvons la terminaison de l'Algorithme 2.6 calculant la factorielle. La fonction $V : n \mapsto n$ est un variant (vers l'ordre bien-fondé des entiers naturels $(\mathbb{N}, \leq)$), puisque nous avons déjà remarqué que lors du calcul de factorielle(n) le seul appel récursif est factorielle(n') avec $n' = n - 1$ et donc $V(s') < V(s)$. Ainsi l'Algorithme 2.6 termine.

Exemple 2.20. Certains algorithmes récursifs requièrent un peu plus d'attention. C'est le cas de l'Algorithme 2.7 d'Ackermann. Prouvons qu'il termine.

Ackermann(m, n)
Entrée: m, n deux entiers positifs ;
Sortie: un entier positif.
1: **si** $m = 0$ **alors**
2: **renvoyer** $n + 1$
3: **si** $n = 0$ **alors**
4: **renvoyer** Ackermann$(m - 1, 1)$
5: **renvoyer** Ackermann$(m - 1,$ Ackermann$(m, n - 1))$

Algorithme 2.7 – L'algorithme d'Ackermann, un exemple de récursion compliquée.

1. Nous vérifions d'abord que l'algorithme est validement défini. Le calcul de Ackermann(m, n), pour $(m, n) \in \mathbb{N}^2$ peut provoquer trois appels récursifs différents,
   - ▷ le premier sur $(m - 1, 1)$, si $m \neq 0$ et $n = 0$. Dans ce cas, à cause de la présence des conditionnelles, nous avons bien $(m - 1, 1) \in \mathbb{N}^2$, l'appel récursif est valide ;
   - ▷ le deuxième sur $(m, n-1))$, avec $m > 0$ et $n > 0$, donc en particulier $m \in \mathbb{N}$ et $n-1 \in \mathbb{N}$, cet appel récursif est valide ;
   - ▷ le troisième sur $(m - 1,$ Ackermann$(m, n - 1))$. avec $n, m > 0$. Nous avons $m > 0$ donc $m-1 \in \mathbb{N}$. Puisque par la post-condition, Ackermann$(m, n-1) \in \mathbb{N}$, cet appel est aussi valide.
2. Ensuite nous exhibons un variant : la fonction $V : (m, n) \mapsto (m, n) \in (\mathbb{N}^2, \leq_{lex})$ est un variant de l'Algorithme 2.7. En effet, lors de l'exécution de Ackermann(m, n), les appels récursifs possibles sont :
   - ▷ Ackermann$(m - 1, 1)$ avec $(m - 1, 1) <_{lex} (m, n)$;
   - ▷ Ackermann$(m, n - 1)$ avec $(m, n - 1) <_{lex} (m, n)$;
   - ▷ Ackermann$(m - 1, k)$, où $k =$ Ackermann$(m, n - 1)$, avec $(m - 1, k) <_{lex} (m, n)$.

À retenir.
- ▷ Il n'existe pas de méthode systématique pour décider si un algorithme termine. Cependant, nous disposons de techniques qui sont en générale suffisante, et en premier lieu celle des variants.
- ▷ Un élément est minimal s'il n'admet pas d'élement strictement plus petit.
- ▷ Un ordre bien-fondé est tel que chaque sous-ensemble admet au moins un élément minimal. De manière équivalente, un ordre bien-fondé n'admet pas de suite infinie strictement décroissante.
- ▷ L'état d'exécution d'un algorithme est une fonction partielle des variables vers les valeurs qu'elles contiennent.
- ▷ Un variant d'une boucle **tant que** est une fonction des états d'exécution d'un algorithme vers un ordre bien-fondé, tel que le variant décroit à chaque itération de la boucle.
- ▷ Un variant d'une fonction récursive est une fonction des entrées de la fonction vers un ordre bien fondé, dont l'image décroit à chaque appel récursif.
- ▷ Si une boucle ou un algorithme récursif admet un variant, alors elle termine nécessairement.

```java
public enum RainbowColor {
  RED, ORANGE, YELLOW, GREEN,
  BLUE, INDIGO, VIOLET;
}
```

```c
typedef enum {
  RED, ORANGE, YELLOW, GREEN,
  BLUE, INDIGO, VIOLET
} RainbowColor;
```

FIGURE 2 – Définition de type par énumération en Java (à gauche), en C (à droite).

3 Définitions inductives et structures inductives

3.1 Définir des valeurs

Les programmes informatiques permettent la manipulation de données, que ce soit l'analyse de données existantes ou la transformation et la production de nouvelles données. D'un point de vue technique, toutes ces données sont des séquences d'octets traitées par le processeur et stockées en mémoire. Mais cette vision des données est très réductrice : les données ont un sens dans leur contexte qui n'est pas explicite dans une représentation en binaire. Afin d'exploiter la spécificité de chaque donnée, nos programmes doivent pouvoir exploiter les données au travers de représentations structurées adaptées aux données et aux traitements que nous souhaitons leur appliquer. Il nous faut donc des techniques pour définir de nouvelles données, selon nos usages, qui rendent explicites leurs structures et leurs propriétés.

Dans ce chapitre, nous étudions comment définir des données (point de vue informatique) ou des valeurs (point de vue mathématique). Nous approfondirons dans les chapitres suivants la notion de définition inductive, et leur utilisation pour définir des fonctions, écrire des algorithmes, et prouver des propriétés.

Commençons par faire le point sur les techniques auxquelles nous sommes déjà habitués pour définir un nouvel ensemble de valeurs, avec nos deux points de vue mathématique et informatique.

Définition d'un ensemble par énumération. L'ensemble est défini en listant explicitement chacun des éléments de l'ensemble un par un. Ceci nécessite néanmoins que l'ensemble soit fini.

Exemples :

- ▷ l'ensemble des booléens $\{\text{vrai}, \text{false}\}$;
- ▷ l'ensemble des nombres premiers à un seul chiffre $\{2, 3, 5, 7\}$;
- ▷ l'ensemble des couleurs de l'arc-en-ciel $\{\text{rouge}, \text{orange}, \text{jaune}, \text{vert}, \text{bleu}, \text{indigo}, \text{violet}\}$.

La plupart des langages de programmation supporte la création de nouveaux types permettant de représenter les valeurs d'une énumération, comme dans l'exemple de la Figure 2.

Définition d'un ensemble à l'aide d'opérateurs. L'ensemble est défini par l'application d'un opérateur mathématique sur un ou plusieurs autres ensembles déjà définis. Les possibilités sont vastes, néanmoins les opérateurs que nous utiliserons en général sont :

- ▷ l'union de deux ensembles $A \cup B$, défini par $x \in A \cup B$ si et seulement si $x \in A$ ou $x \in B$;
- ▷ le produit cartésien de deux ensembles $A \times B$, qui est un ensemble de paires ordonnées, défini par $(a, b) \in A \times B$ si et seulement si $a \in A$ et $b \in B$;
- ▷ l'étoile de Kleene $A^\star$ d'un ensemble, qui est l'ensemble des séquences $a_1, a_2, \ldots, a_n$ avec $a_i \in A$ pour tout $i \in \{1, 2, \ldots, n\}$ et $n \in \mathbb{N}$.

L'union et le produit cartésien peuvent être généralisés à plus de deux ensembles [6].

Exemples :

- ▷ l'ensemble $\{3 \times n : n \in \mathbb{N}\} \cup \{5 \times n : n \in \mathbb{N}\}$ des entiers divisibles par 3 ou 5 ;

6. Il faudrait aussi définir ce qu'est une paire ordonnée ou une séquence. Dans la théorie des ensembles ZFC, la paire (a, b) peut être définie comme une notation pour l'ensemble $\{a, \{a, b\}\}$. Pour les séquences, nous en donnerons bientôt une définition.

```java
public record People(String name, int age) {}

public sealed interface LampState implements Off, On {}
public record Off() implements LampState {}
public record On(Color color) implements LampState {}
```

```rust
struct People {
  age: i32,
  name: String,
}

enum LampState { Off, On(Color), }
```

FIGURE 3 – Définition de types par produit cartésien (enregistrement) ou par union disjointe en Java (en haut), en Rust (en bas). Une personne est le produit d'un nom et d'un âge. L'état de la lampe est l'union du singleton {éteinte} et de l'ensemble de couleurs (pour la lampe allumée).

▷ l'ensemble $\{1, 2, \ldots, 31\} \times \{\text{janvier}, \ldots, \text{décembre}\}$ des jours d'une année (approximativement, certaines dates n'étant pas valides comme le 31 février) ;

▷ l'ensemble $\{a, b\}^\star$ des mots sur l'alphabet $\{a, b\}$, contenant $\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots$.

La plupart des langages de programmation supporte le produit cartésien, soit avec des enregistrements (`struct` en C, `record` en Pascal, ...), soit avec des classes. Certains langages supportent l'union disjoint de deux types soit nativement, soit grâce au polymorphisme de classe, comme illustré en Figure 3. Les séquences sont réalisées par des tableaux ou par des collections telles que les listes chainées.

Définition d'un ensemble par spécification. L'ensemble est défini comme un sous-ensemble de valeurs ayant une certaine propriété, prises parmi un ensemble plus grand. Pour cela il nous faut :

▷ un ensemble de valeurs S déjà existantes,

▷ une propriété ϕ, sous la forme de formule logique ayant une variable libre s pouvant prendre valeur dans S.

Nous pouvons alors définir $S_\phi := \{s \in S : \phi(s)\}$, c'est-à-dire l'ensemble des éléments s de S tel que $\phi(s)$ est vrai.

Exemples :

▷ l'ensemble des entiers pairs $\{n \in N : n \equiv 0 \bmod 2\}$, avec $\phi(n) := n \equiv 0 \bmod 2$,

▷ l'intervalle des réels entre $-\pi$ et π, $\{x \in \mathbb{R} : -\pi \leq x \leq \pi\}$, avec $\phi(x) := -\pi \leq x \leq \pi$,

▷ l'ensemble des mots sur l'alphabet $\{a, b\}$ comprenant autant de lettres a que de lettres b, $\{w \in \{a, b\}^\star | w|_a = |w|_b\}$, avec $\phi(w) := |w|_a = |w|_b$.

Ce format de définition est difficile à réaliser en tant que type dans les langages de programmation. À cause de la contrainte imposant de partir d'un ensemble S de valeurs déjà existantes, il n'est de toute façon pas adapté pour créer de nouvelles valeurs.

Analyse. Nous disposons de plusieurs façons de définir des valeurs, mais possédant des limitations :

▷ les énumérations sont finies, et ne permettent pas de représenter de valeurs qui ne sont pas connues par avance, puisque leurs valeurs sont fixées une fois pour toute ;

- ▷ les opérateurs supposent la pré-existence d'autres ensembles, dont nous pouvons construire des unions ou des produits cartésiens. Il n'est pas possible de créer ainsi un ensemble de valeur ex-nihilo. Cependant ce sont des constructions largement employées en programmation ;
- ▷ les spécifications nécessitent aussi des valeurs pré-existantes, et sont en pratique difficilement utilisable pour la programmation.

Est-il possible d'avoir un mécanisme permettant de définir simplement des ensembles infinis, sans présupposer l'existence d'un autre ensemble avant ? D'un point de vue concret, est-il toujours possible d'écrire des programmes significatifs, en Java par exemple, sans utiliser les types primitifs (`int`, `double`,...) prédéfinis par la langage ?

Dans cette section, nous allons donner une réponse positive à cette question : les types primitifs ne sont théoriquement pas nécessaires ! (ils sont néanmoins essentiels en pratique pour écrire des programmes efficaces). Plus généralement, il existe une méthode pratique pour définir des ensembles de valeurs infinis, et cette méthode possède de nombreux avantages qui en font un outil de choix dans de nombreux domaines de l'informatique. Nous allons aussi voir comment mettre en œuvre cette technique en Java ou dans d'autres langages de programmation.

À retenir. Les langages de programmation permettent de définir de nouvelles valeurs par énumération, ou en utilisant les opérateurs d'union et de produit cartésien. Par exemple en Java,

- ▷ une classe est le produit cartésien des types de ses attributs ;
- ▷ une interface est l'union des implémentations qu'elle possède ;
- ▷ les énumérations sont réalisés avec les classes `enum`.

Exercice 3.1. Proposer :

(1) une définition par énumération des diviseurs de 60 ;

(2) une définition par spécification des diviseurs de 60 ;

(3) une définition par spécification des nombres premiers (il vous faut écrire une formule logique satisfaite par les nombres premiers, sachant qu'un nombre est premier si tous ses diviseurs positifs sont 1 ou lui-même) ;

(4) une définition, à l'aide d'opérateurs, de l'ensemble des vecteurs réels en 3 dimensions.

3.2 Exemple introductif : les entiers naturels

Comment les mathématiciens définissent-ils l'ensemble des nombres naturels ? En programmation, comment redéfinirions-nous le type des entiers s'il n'était pas proposé par le langage que nous utilisons ? Notre approche (qui n'est pas la seule possible) va être constructive. Nous pouvons faire deux remarques sur ce que nous voulons des entiers naturels :

- ▷ ils commencent par 0, le plus petit des entiers ;
- ▷ il en existe une infinité. Plus précisément, pour tout entier, il existe un autre entier immédiatement supérieur, appelé le *successeur*. 11 est le successeur de 10, et 42 est le successeur de 41.

Pour l'instant, nous voulons uniquement décrire les valeurs possibles. Nous ignorons donc délibérement les opérations arithmétiques comme l'addition ou la multiplication, que nous redéfinirons plus tard.

Définition des entiers naturels. Le principe d'une *définition inductive* est d'observer que nos deux remarques, l'existence de 0 et du successeur, sont suffisantes pour *définir* l'ensemble des entiers. Précisément :

Définition 3.1. L'ensemble des *entiers naturels* $\mathbb{N}$ est le **plus petit ensemble** contenant Z (zéro), et tel que pour tout $n \in \mathbb{N}$, $S(n) \in \mathbb{N}$.

Voilà une définition pour le moins surprenante, car elle semble être circulaire ! En effet la définition de $\mathbb{N}$ fait référence à $\mathbb{N}$. Nous interdisons en général les définitions circulaires, à juste titre puisque les définitions circulaires n'ont pas de sens. Essayons de bien comprendre cette définition, pour comprendre ce qu'elle signifie et pourquoi elle n'est en fait pas circulaire. Le terme crucial est *plus petit ensemble*, parmi tous les ensembles ayant une certaine propriété. Pour un ensemble arbitraire X, la propriété P en question peut s'écrire formellement

$$P(X) := Z \in X \wedge (\forall n, n \in X \implies S(n) \in X).$$

La définition ne considère, parmi tous les ensembles possibles, que ceux qui vérifie la propriété P. Par exemple :

- ▷ $X := \{Z\}$ ne vérifie pas la propriété, car $Z \in X$ mais $S(Z) \notin X$;
- ▷ $X := \{S(Z), S(S(Z)), S(S(S(Z))), \ldots\}$ ne vérifie pas la propriété, car $Z \notin X$;
- ▷ $X := \{Z, S(Z), S(S(Z)), \ldots\}$ vérifie la propriété ;
- ▷ $X := \{Z, S(Z), S(S(Z)), \ldots, Y, S(Y), S(S(Y)), \ldots\}$ vérifie aussi la propriété ;
- ▷ $X := \{Z, S(Z), T(Z), S(S(Z)), T(T(Z)), \ldots\}$ vérifie aussi la propriété.

Ainsi, il y une infinité d'ensembles vérifiant la propriété, et une infinité qui ne la vérifient pas. Il se trouve que parmi tous ces ensembles vérifiant P, il en existe un plus petit, notons le Γ (*gamma*). On peut même le construire assez simplement (si nous disposons d'un temps infini en tout cas) :

- ▷ clairement la propriété P oblige Γ à contenir Z, donc $Z \in \Gamma$;
- ▷ puisque $Z \in \Gamma$, la propriété P oblige Γ à contenir aussi $S(Z)$, donc $S(Z) \in \Gamma$;
- ▷ puisque $S(Z) \in \Gamma$, la propriété P oblige Γ à contenir aussi $S(S(Z))$, donc $S(S(Z)) \in \Gamma$;
- ▷ puisque $S(S(Z))) \in \Gamma$, la propriété P oblige Γ à contenir aussi $S(S(S(Z)))$, donc $S(S(S(Z))) \in \Gamma$;
- ▷ ...

Ainsi, à cause de P, Γ doit contenir au moins $N := \{Z, S(Z), S(S(Z)), S(S(S(Z))), \ldots\}$. Mais par ailleurs, cet ensemble infini N vérifie la propriété P. Puisque $N \subseteq \Gamma$ et que Γ est par hypothèse le plus petit ensemble vérifiant P, on en déduit que $\Gamma = N$. Finalement, c'est cet ensemble que nous décidons d'appeler $\mathbb{N}$, en introduisant des notations pour chaque entier $0 := Z, 1 := S(Z)$, $2 := S(S(Z))$, $3 := S(S(S(Z)))$, ...

Construction des entiers. Il est important de remarquer que notre définition inductive des entiers est constructive, au sens qu'elle donne une procédure permettant d'énumérer (de lister) tous les entiers. Cette procédure fonctionne par étape, à l'étape 0, nous ne connaissons que l'entier Z dont l'existence est directement postulée par la définition, et nous posons donc $X_0 = Z$. À l'étape 1, nous appliquons la règle du successeur sur chaque entier dans X_0 pour découvrir d'autres entiers, nous découvre ainsi $S(Z)$, ce qui nous donne

$$X_1 := X_0 \cup \{S(i) : i \in X_0\} = \{Z, S(Z)\}.$$

Ainsi de suite, en calculant $X_{n+1} := X_n \cup \{S(i) : i \in X_n\}$, nous construisons

$$\begin{aligned}
X_2 &:= \{Z, S(Z)\} \cup \{S(Z), S(S(Z))\} = \{Z, S(Z), S(S(Z))\} \\
X_3 &:= \{Z, S(Z), S(S(Z))\} \cup \{S(Z), S(S(Z)), S(S(S(Z)))\} = \{Z, S(Z), S(S(Z)), S(S(S(Z)))\} \\
X_4 &:= \{Z, S(Z), S(S(Z)), S(S(S(Z)))\} \cup \{S(Z), S(S(Z)), S(S(S(Z))), S(S(S(S(Z))))\} \\
&\quad = \{Z, S(Z), S(S(Z)), S(S(S(Z))), S(S(S(S(Z))))\} \\
&\ldots
\end{aligned}$$

```java
public sealed interface Nat permits Zero, Successor {
  public static final Nat ZERO = new Zero();
  default Nat successor() {
    return new Successor(this);
  }
}
public record Zero() implements Nat {}
public record Successor(Nat n) implements Nat {}
```

```haskell
data Nat
  = Zero
  | Successor of Nat
```

FIGURE 4 – Définition d'entiers inductifs en Java (en haut) et en Haskell (en bas). La constante `ZERO` et la méthode par défaut `successor()` ne sont pas requises pour définir `Nat`, mais l'ajout de constante et de méthodes fabriquants les valeurs permettent souvent de simplifier l'usage des types inductifs.

On a alors $\mathbb{N} = \bigcup_{n\geq 0} X_n$ [7]. Cet aspect constructif rend les définitions inductives adaptées pour définir de nouvelles données en programmation, comme dans la Figure 4. De plus, la procédure décrite ci-dessus pour générer toutes les valeurs qui sont des entiers naturels est algorithmique. Elle est donc implémentable dans un programme, par exemple en réalisant un générateur des entiers naturels en Python, réalisé en Figure 5. Ce générateur construit ainsi tous les entiers naturels, un par un à chaque `yield`.

3.3 Deuxième exemple : les expressions arithmétiques

L'exemple des entiers naturels ne montre pas clairement toutes les possibilités offertes par les définitions inductives. Intéressons-nous maintenant à un cas d'usage plus typique : la représentation d'expressions arithmétiques. Le contexte est celui d'une calculatrice, d'une feuille de calcul type *excel*, ou bien d'un langage de programmation, pour lequel nous souhaitons décrire une formule mathématique, comprenant des variables. Cette formule pourra être évaluée, en faisant varier les valeurs des variables. Nous allons nous restreindre aux formules comprenant des littéraux entiers (relatifs), des variables consistant en une lettre de l'alphabet, et les opérations d'addition et de multiplication. Les parenthèses permettent de préciser les opérandes de chaque opérateur arithmétique, mais peuvent être omises à l'écriture comme il en est typiquement l'usage. Voici des exemples de telles formules :

$$z, 3 \times x + 1, 42, x \times y + y \times z, (x + 3) \times (z + 2).$$

Attention, dans cette section, nous n'évaluons pas les expressions : la formule $2 + 2$ **n'est pas égale** à la formule 4 (néanmoins leurs valeurs sont égales), et la formule $x + 1$ n'est pas égale à la formule $1 + x$ non plus. Les formules sont des constructions syntaxiques auxquelles nous ne donnons pas (encore) un sens.

Définition inductive des formules. Nous savons qu'un littéral entier, comme 42 ou -5, constitue déjà une formule, de même qu'une variable comme x ou y. Par ailleurs, nous pouvons faire des additions : l'addition se fait entre deux formules, et est une formule. Le principe est aussi vrai pour la multiplication. Nous pouvons donc poser :

7. Nous ne sommes pas autorisés à utiliser cette égalité comme définition de $\mathbb{N}$ car cela serait une définition cyclique, l'indice n de l'union étant pris dans $\mathbb{N}$.

```python
from typing import Generator

class Nat:
  def succ(self) -> Nat:
    return Succ(self)

class Zero(Nat):
  pass

class Succ(Nat):
  def __init__(self, predecessor: Nat):
    self.predecessor = predecessor

def nat_generator(n: Nat) -> Generator[Nat, None, None]:
  current = n
  while True:
    yield current
    current = current.succ()
```

FIGURE 5 – Une définition inductive des entiers naturels en Python, avec un générateur qui produira tous les entiers naturels un par un.

Définition 3.2. L'ensemble $\mathcal{F}$ des *formules arithmétiques* est le plus petit ensemble tel que :

(i) pour tout entier $n \in \mathbb{Z}$, $n \in \mathcal{F}$;

(ii) pour toute lettre $\alpha \in \{a, b, c, \ldots, z\}$, $\alpha \in \mathcal{F}$;

(iii) pour toutes formules $f_1 \in \mathcal{F}$ et $f_2 \in \mathcal{F}$, $(f_1 + f_2)$ et $(f_1 \times f_2)$ sont des formules.

Nous retrouvons la même forme de définition : le plus petit ensemble vérifiant la propriété P, avec

$$P(X) := (\forall n \in \mathbb{Z}, n \in X) \wedge (\forall \alpha \in \{a, b, c, \ldots, z\}, \alpha \in X) \wedge (\forall f_1, f_2 \in X, (f_1 + f_2) \in X \wedge (f_1 \times f_2) \in X)$$

Construction des formules. La même procédure que pour les entiers naturels peut être utilisée pour énumérer toutes les formules possibles. Nous commençons par les éléments explicitement dans $\mathcal{F}$ d'après la définition.

$$\mathcal{F}_0 = \mathbb{Z} \cup \{a, b, c, \ldots, z\}$$

Puis nous construisons toutes les formules de la forme $f_1 + f_2$ et $f_1 \times f_2$, pour f_1, f_2 deux formules de $\mathcal{F}_0$. Il s'agit de toutes les formules avec exactement un opérateur :

$$\mathcal{F}_1 = \mathcal{F}_0 \cup \{0 + 0, 0 + 1, 1 + 0, \ldots, 0 + x, x + 0, \ldots, x \times y, x \times z, \ldots\}$$

De même $\mathcal{F}_2$ se construit avec toutes les formules de la forme $f_1 + f_2$ et $f_1 \times f_2$, pour f_1, f_2 deux formules de $\mathcal{F}_1$ (ayant donc au plus un opérateur).

$$\mathcal{F}_2 = \mathcal{F}_1 \cup \{0 + (0 + 0), (1 \times 0) + (0 \times 1), (x \times x) + (y \times y), \ldots\}$$

$\mathcal{F}_2$ contient toutes les formules de *hauteur* au plus 2. Ainsi de suite, nous obtenons à tour de rôle tous les $\mathcal{F}_n$ contenant les formules de hauteur au plus n, et $\mathcal{F} = \bigcup_{n \geq 0} \mathcal{F}_n$.

Implémentation. Nous pouvons mettre en œuvre cette définition. Dans un langage objet, nous utilisons une classe par règle de construction dans la définition inductive, voir en Figure 6. Les

```java
public sealed interface Expression permits Litteral, Variable, Operation {}
public record Litteral(int n) implements Expression {}
public record Variable(char c) implements Expression {}
public record Operation(
  Operator op,
  Expression leftOperand,
  Expression rightOperand
) implements Expression {}
public enum Operator { SUM, PRODUCT; }
```

```ocaml
type operator = Addition | Product
type expression =
  | Litteral of int
  | Variable of char
  | Operation of operator * expr * expr
```

FIGURE 6 – Définitions des expressions arithmétiques dans un langage objet (Java, en haut), et dans un langage fonctionnel (Ocaml, en bas).

langages fonctionnels permettent d'exprimer bien plus succintement les définitions inductives, ce qui se justifie par leur usage très commun dans cette famille de langages.

L'énumération des formules peut se réaliser en procédant par hauteur croissante, cependant le nombre de formules de chaque hauteur est infini puisqu'il y a une infinité de littéraux. Même en limitant les littéraux à un sous-ensemble fini d'entiers, le nombre de formules de hauteur h croît exponentiellement en h.

3.4 Formes des définitions inductives

Nous généralisons maintenant les deux exemples, en donnant la forme générale des définitions inductives. Nous montrons ensuite que ces définitions sont valides ; pour cela il faut démontrer qu'il existe effectivement un plus petit ensemble parmi tous ceux ayant la propriété utilisée par la définition inductive. Ce n'est vrai que parce que nous utilisons des propriétés d'une forme très particulière :

- ▷ certaines règles postulent l'appartenance de valeurs explicitement citées dans l'ensemble défini, que ce soit Z dans le cas des entiers naturels, ou les entiers et les variables dans le cas des expressions arithmétiques ;
- ▷ d'autres règles proposent des méthodes d'assemblage ou de construction, de nouvelles valeurs à partir de valeurs déjà répertoriées. Par exemple $S(n)$ à partir de n pour les entiers naturels, $(f_1 + f_2)$ et $(f_1 \times f_2)$ pour les expressions arithmétiques.

Ainsi,

Définition 3.3 (Définition inductive)**.** Soit E un ensemble arbitraire. Une *définition inductive* d'une partie de E est une paire (B, R) telle que

- ▷ $B \subseteq E$ est un ensemble non-vide, appelé *ensemble de base* ou *d'axiomes* ;
- ▷ R est un ensemble de fonctions partielles sur E, appelées *règles d'induction*. Chaque règle peut avoir une arité (un nombre d'arguments) arbitraire, donc être de type $E^n \to E$ avec $n \geq 1$.

Pour rappel, une fonction partielle d'arité n est une relation $\mathcal{R}$ à $n+1$ termes sur E, telle que si $(a_1, a_2, \ldots, a_n, x) \in \mathcal{R}$ et $(a_1, a_2, \ldots, a_n, y) \in \mathcal{R}$, alors $x = y$. Comme pour une fonction, le dernier terme représente l'image par la fonction partielle des n premiers termes, et est complètement

déterminé par les n premiers termes. Nous le notons donc en général $a_1, a_2, \ldots, a_n \to x$. Contrairement à une fonction totale, tout choix de n premiers termes ne donne pas forcément une image.

Notons que dans cette définition, nous supposons l'existence d'un ensemble E, qui contient l'ensemble à définir. En général, E est une algèbre de termes suffisamment générale, ou simplement l'ensemble des séquences de caractères. Dans un langage de programmation ce serait peut-être l'ensemble des constructions valides de ce langage. En pratique, E n'est souvent pas explicité car ce n'est tout simplement pas utile.

Exemple des entiers naturels. Dans ce cas, il n'y a qu'un seul axiome, $B = \{Z\}$, et une seule règle, la fonction (totale) $n \to S(n)$, donc $R = \{n \to S(n)\}$.

Exemple des expressions arithmétiques. Dans ce cas, il y a un axiome par nom de variable possible et un axiome par littéral possible, ce qui donne $B = \mathbb{N} \cup \{a, b, c, \ldots, z\}$. La première règle est celle de l'addition, la relation $\{(f_1, f_2, (f_1 + f_2)) : f_1, f_2 \in E\}$, que nous notons comme la fonction (totale) $f_1, f_2 \to (f_1 + f_2)$. La deuxième régle, similaire est celle de la multiplication. Ainsi, $R = \{(f_1, f_2 \to (f_1 + f_2)), (f_1, f_2 \to (f_1 \times f_2))\}$.

Justification des définitions inductives. En quoi une telle paire (B, R) d'axiomes et de règles définit une partie de E ? Comme dans les exemples, nous considérons le plus petit sous-ensemble de E qui contienne B les axiomes, et qui soit clos par R.

Définition 3.4. Un ensemble X est *clos* par une fonction partielle f d'arité n si pour tout choix de $x_1, x_2, \ldots, x_n$, si $f(x_1, x_2, \ldots, x_n)$ existe est vaut y, alors $y \in X$.

Un ensemble X est *clos* par un ensemble de fonctions partielles R s'il est clos par toute fonction partielle $f \in R$.

Autrement dit, l'ensemble X est clos par R si, quelque soit la règle que j'applique sur des éléments de X, la valeur que j'obtiens est elle-même dans X : il n'est pas possible de trouver de nouveaux éléments par application des règles. Par exemple, l'ensemble des nombres pairs n'est pas clos par la règle de la définition inductive des entiers naturels, puisque $1 = S(Z)$ s'obtient en appliquant la règle du successeur sur $0 = Z$.

Pour valider nos définitions inductives, il faut montrer qu'un tel plus petit ensemble clos existe.

Théorème 3.5 (Théorème du point fixe)**.** Soit (B, R) une définition inductive sur un ensemble E. Il existe un plus petit ensemble $X \subseteq E$, contenant B et clos par R. Autrement dit, tout ensemble $Y \subseteq E$ tel que $B \subseteq Y$ et Y est clos par R, contient X.

Démonstration. Nous définissons par induction sur $\mathbb{N}$ les ensembles $X_0 := B$ et

$$X_{n+1} := X_n \cup \{y : \text{il existe } f \in R \text{ d'arité } k, \text{ et } x_1, x_2 \ldots, x_k \in X_n \text{ avec } f(x_1, x_2, \ldots, x_k) = y\},$$

et finalement $X = \bigcup_{n \in \mathbb{N}} X_n$. X_{n+1} s'obtient donc en ajoutant toutes les valeurs obtenues par l'application de toutes les règles sur les éléments de X_n. Nous montrons que X est clos par R, et que X est contenu dans tout sous-ensemble de E clos par R et contenant B.

Clôture par R. Soit $f \in R$ une règle d'arité k, et $x_1, x_2, \ldots, x_k \in X$ des éléments de X, et supposons que $f(x_1, x_2, \ldots, x_k)$ existe et vaut y. Il nous faut montrer que $y \in X$. Pour tout $i \in \{1, 2, \ldots, k\}$, puisque $x_i \in X$ et par définition de X, il existe n_i tel que $x_i \in X_{n_i}$. Soit $n = \max_{i \in \{1,2,\ldots,k\}} n_i$, nous avons alors $x_1, x_2, \ldots, x_k \in X_n$. Par définition de X_{n+1} cela implique que $y \in X_{n+1}$, et donc $y \in X$.

Contenance dans tout autre sous-ensemble clos. Soit Y un sous-ensemble de E, contenant B et clos par R. Nous démontrons que $X \subseteq Y$. Pour cela, il suffit de montrer que pour tout $n \in \mathbb{N}$, $X_n \subseteq Y$, ce que nous allons faire par induction sur n. Comme $X_0 = B$, par hypothèse $B \subseteq Y$, donc $X_0 \subseteq Y$, l'hypothèse est vrai au rang 0. Soit un entier arbitraire $n \in \mathbb{N}$, supposons notre hypothèse vraie au rang n, c'est-à-dire $X_n \subseteq Y$. Soit $y \in X_{n+1}$. Ou bien $y \in X_n$, et par hypothèse de récurrence $X_n \subseteq Y$ donc $y \in Y$. Ou bien, par définition de X_{n+1}, il existe une fonction partielle f d'arité k et $x_1, x_2, \ldots, x_k \in X_n$ tels que $f(x_1, x_2, \ldots, x_k) = y$. Mais dans ce

cas, puisque $X_n \subseteq Y$ par hypothèse de récurrence, $x_1, x_2, \ldots, x_n \in Y$. Puisque par hypothèse sur Y, Y est clos par R, nous en déduisons que $y \in Y$. Ainsi, $X_{n+1} \subseteq Y$, l'hypothèse est démontrée au rang $n+1$, terminant notre preuve par récurrence.

Conclusion. Puisque X contient B et est clos par R, et que tout ensemble contenant B et clos par R contient X, X est le plus petit ensemble contenant V et clos par R. □

Cette preuve suit l'intuition algorithmique de la définition inductive, consistant à calculer une clôture de B par R en générant toujours plus de valeurs. Il existe une autre preuve classique mais plus abstraite, qui consiste à dire que (1) il existe bien au moins un ensemble contenant B et clos par R, à savoir E, et (2) l'intersection de tous les ensembles contenant B et clos par R est elle-même un ensemble contenant B et clos par R. Cette intersection est alors le plus petit ensemble ayant cette propriété.

Peu importe la preuve, le théorème nous montre qu'une définition inductive (B, R) est bien interprétable comme la définition d'un ensemble.

À retenir. Un ensemble défini par définition inductive est la plus petite clôture d'un ensemble de base par des règles d'induction.

Exercice 3.2. Considérons la définition inductive de l'écriture des entiers en binaires (B, R), avec $B = \{0, 1\}$, et R comprenant deux règles, les fonctions partielles $1x \to 1x0$ et $1x \to 1x1$ (x peut être vide). Calculer X_0, X_1, X_2 et X_3 tels que définis dans la preuve du Théorème 3.5.

3.5 Représentation des définitions inductives

Il est d'usage de représenter les définitions inductives avec des notations adaptées. Nous en détaillons trois que nous réutiliserons dans le reste du cours.

La notation standard. Les définitions inductives (B, R) sont souvent notées sous la forme

- ▷ (Base) $b \in X$, pour tout $b \in B$;
- ▷ (Induction) si $x_1, \ldots x_n \in X$ alors $r(x_1, \ldots, x_n) \in X$.

Il est possible d'ajouter plus de lignes de base ou d'induction, ainsi que des conditions supplémentaires à chaque ligne, selon les besoins, pour exprimer la fonction partielle de la règle de construction.

Illustrons la notation avec nos exemples. La définition des entiers naturels $\mathbb{N}$ se note ainsi :

- ▷ (Base) $Z \in \mathbb{N}$;
- ▷ (Induction) si $n \in \mathbb{N}$, alors $S(n) \in \mathbb{N}$.

La définition des expressions arithmétiques $\mathcal{F}$ se note :

- ▷ (Base) $n \in \mathcal{F}$ pour tout $n \in \mathbb{Z}$;
- ▷ (Base) $\alpha \in \mathcal{F}$ pour tout $\alpha \in \{a, b, c, \ldots, z\}$;
- ▷ (Induction) si $f_1, f_2 \in \mathcal{F}$, alors $(f_1 + f_2) \in \mathcal{F}$;
- ▷ (Induction) si $f_1, f_2 \in \mathcal{F}$, alors $(f_1 \times f_2) \in \mathcal{F}$;

La notation par des règles. Cette notation graphique est particulièrement appréciée en logique. On utilise une barre horizontale, au-dessus de laquelle sont placées les *prémisses* de la règle, et en-dessous la conclusion qui découle des prémisses. Nous pouvons ajouter une étiquette à droite de la barre horizontale, pour apporter des informations supplémentaires ou bien simplement pour nommer la règle.

Avec cette notation, un axiome ou une règle de base n'a pas de prémisse et a une conclusion. La barre horizontale n'a donc pas d'élément au-dessus, et présente un élément de B en-dessous. Ainsi, nous avons pour tout $b \in B$:

$$\frac{}{b}$$

Une règle d'induction, dénotant une fonction partielle r d'arité k, et k éléments $x_1, x_2, \ldots, x_k$ tels que $r(x_1, x_2, \ldots, x_k)$ existe, a pour prémisses les éléments $x_1, x_2, \ldots, x_k$, et pour conclusion $r(x_1, x_2, \ldots, x_k)$. Ainsi pour tout règle d'arité k, nous noterons :

$$\frac{x_1 \qquad x_2 \qquad \ldots \qquad x_k}{r(x_1, x_2, \ldots, x_k)}$$

Pour illustrer cette notation, redonnons la définition des entiers naturels :

$$\frac{}{Z} \qquad \frac{n}{S(n)}$$

La règle de gauche s'interprète comme *Z est un entier naturel*, et celle de droite comme *si n est un entier naturel, alors $S(n)$ est un entier naturel*. Voyons maintenant la définition des expressions arithmétiques :

$$\frac{}{n}\ n \in \mathbb{Z} \qquad \frac{}{\alpha}\ \alpha \in \{a, b, c, \ldots, z\} \qquad \frac{e_1 \qquad e_2}{e_1 + e_2} \qquad \frac{e_1 \qquad e_2}{e_1 \times e_2}$$

Notons l'emploi d'étiquettes, utilisées ici pour préciser l'ensemble des valeurs autorisées dans chaque cas de base. Comme pour les entiers naturels, les deux règles de gauche s'interprètent comme *tout $n \in \mathbb{Z}$ et tout $\alpha \in \{a, b, c, \ldots, z\}$ est une formule*, et les règles de droite comme *si e_1 et e_2 sont des formules, alors $e_1 + e_2$ et $e_1 \times e_2$ aussi.*

La notation BNF. Cette notation est très utilisée en informatique, notamment pour spécifier les grammaires des langages de programmation. Nous la retrouvons plus ou moins dans la syntaxe pour définir un type inductif dans des langages variés comme Rust ou Haskell. Le principe est d'écrire une définition récursive, en séparant chaque cas par une barre verticale. Notons $R = \{r_1, r_2, \ldots, r_n\}$ les règles, $B = \{b_1, \ldots, b_l\}$ les cas de bases. Alors la définition inductive (B, R) est notée

$$\begin{aligned} e ::= \ & r_1(e, e, \ldots, e) \\ & \mid r_2(e, e, \ldots, e) \\ & \mid \ \ldots \\ & \mid r_n(e, e, \ldots, e) \\ & \mid b_1 \mid b_2 \mid \ldots \mid b_l. \end{aligned}$$

Il est d'usage d'écrire une règle par ligne, mais compacter les cas de base sur une seule ligne comme ici est envisageable. La barre $\mid$ signifie une alternative entre plusieurs constructions possibles (souvenons-nous de l'opérateur booléen de disjonction *ou*, noté `||` en programmation). Dans la règle pour r_1, la notation $r_1(e, e, \ldots, e)$ doit se comprendre comme r_1 appliqué à k valeurs de l'ensemble e (où k est l'arité de r_1). e étant l'ensemble en cours de définition, c'est bien une notation récursive, mais qui représente ici la définition inductive (B, R).

Reprenons nos exemples pour illustrer, en commençant par les entiers naturels :

$$\begin{aligned} n ::= \ & Z \\ & \mid S(n). \end{aligned}$$

L'ordre des règles n'importe pas, ici le cas de case est donné en ligne 2 et la règle d'induction en ligne 3. Pour les expressions arithmétiques, nous écrivons :

$$\begin{aligned} e ::= \ & n \qquad (\text{pour tout } n \in \mathbb{Z}) \\ & \mid \alpha \qquad (\text{pour tout } \alpha \in \{a, b, c, \ldots, z\}) \\ & \mid e + e \\ & \mid e \times e. \end{aligned}$$

Nous pouvons assez libéralement ajouter des indications, par exemple dans ce cas pour préciser les ensembles de cas de base.

Cette notation BNF, pour *Backus-Naur Form* du nom des deux chercheurs l'ayant introduite, peut aussi être utilisée pour définir plusieurs ensembles simultanément. Ce sera notamment le cas lors de l'étude des grammaires formelles et des compilateurs. Voici par exemple une autre version des expressions arithmétiques, mais reprenant aussi la définition d'entiers relatifs et naturels.

$$\begin{aligned} expr ::= \ & var \\ & \mid int \\ & \mid expr\ op\ expr. \end{aligned}$$

$$var ::= \ a \mid b \mid c \mid \ \ldots \ \mid z$$

$$op ::= \ + \ \mid \ \times .$$

$$int ::= \ nat \mid \ -S(nat).$$

$$nat ::= \ Z \mid S(nat).$$

Nous pouvons reconnaître la définition des entiers naturels *nat*, puis *int* représente les entiers relatifs, *op* les opérateurs binaires, et *var* les noms de variables. *expr* représente alors toutes les expressions arithmétiques. Cet exemple ne peut pas donc pas être directement écrit avec une seule définition inductive. Pour écrire l'équivalent avec des règles, nous pouvons utiliser le symbole $\in$ sur les prémisses et les conclusions. Nous pouvons alors avoir plusieurs définitions inductives simultanée, et chaque règle appartient à la définition inductive de l'ensemble auquel sa conclusion appartient. Ainsi notre exemple se réécrit :

$$\frac{}{\alpha \in \mathcal{F}}\ \alpha \in \{a, b, c, \ldots, z\} \qquad \frac{}{n \in \mathcal{F}}\ n \in \mathbb{Z} \qquad \frac{e_1 \in \mathcal{F} \qquad e_2 \in \mathcal{F}}{e_1 \oplus e_2 \in \mathcal{F}}\ \oplus \in \{+, \times\}$$

$$\frac{}{n \in \mathbb{Z}}\ n \in \mathbb{N} \qquad \frac{}{-S(n) \in \mathbb{Z}}\ n \in \mathbb{N}$$

$$\frac{}{Z \in \mathbb{N}} \qquad \frac{n \in \mathbb{N}}{S(n) \in \mathbb{N}}$$

Nous avons ainsi trois règles pour définir $\mathcal{F}$, deux règles pour $\mathcal{Z}$, et les deux dernières règles pour $\mathcal{N}$.

À retenir. Diverses notations co-existent pour les définitions inductives. Nous emploierons l'une ou l'autre pour profiter des avantages de chacune, notamment les formes par règles et BNF. Nous devons donc être capable de les comprendre, et de passer des définitions d'une notation à l'autre.

Exercice 3.3. Donner la notation par règle de la définition inductive des représentations des entiers en binaire, introduite dans l'Exercice 3.2.

3.6 Arbre de dérivation

Souvenons-nous de la preuve du Théorème 3.5, l'ensemble X défini par une définition inductive (B, R) est égal à $\bigcup n \geq 0 X_n$, avec $X_0 = B$ et

$$X_{n+1} := X_n \cup \{y : \text{il existe } f \in R \text{ d'arité } k, \text{ et } x_1, x_2 \ldots, x_k \in X_n \text{ avec } f(x_1, x_2, \ldots, x_k) = y\}.$$

Ainsi, pour toute valeur $x \in X$ définie par la définition inductive appartient à X_n pour un certain entier n. x se construit donc à partir des éléments de B en appliquant les règles de R. Nous allons

capturer ce phénomène à travers la notion d'arbre de dérivation : x se dérive de $x_1, x_2, \ldots, x_k$ par une règle de R, x_1, x_2,..., x_k se dérivent aussi d'autres éléments de X par des règles de R, et ainsi de suite jusqu'aux éléments de B.

Prenons des exemples avant de donner une définition formelle des arbres de dérivation. 4 est un entier naturel, l'abbréviation de $S(S(S(S(Z))))$. Pour prouver que 4 est un entier naturel, il suffit de montrer qu'il est *dérivable* depuis Z avec la règle $n \to S(n)$. Cette dérivation se représente graphiquement :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{Z}}{S(Z)}}{S(S(Z))}}{S(S(S(Z)))}}{S(S(S(S(Z))))}}$$

En lisant de haut en bas, nous déduisons successivement que Z est naturel, donc $S(Z)$ est naturel, donc $S(S(Z))$ est naturel, donc $S(S(S(Z)))$ est naturel, et finalement $S(S(S(S(Z))))$ est naturel.

Le cas des entiers naturels est très monotone car il n'y a qu'une seule règle, d'arité 1. Un exemple plus pertinent consiste en une preuve que $(3 + x) \times (y + (5 \times x))$ est une expression arithmétique.

$$\dfrac{\dfrac{\overline{3} \quad \overline{x}}{3 + x} \qquad \dfrac{\overline{y} \qquad \dfrac{\overline{5} \quad \overline{x}}{5 \times x}}{y + (5 \times x)}}{(3 + x) \times (y + (5 \times x))}$$

En enchaînant graphiquement les règles de la définition inductive depuis les axiomes, nous obtenons effectivement un arbre[8]. Un tel arbre s'appelle un *arbre de dérivation* ou une *preuve*, puisqu'il démontre que la conclusion finale est bien un élément de l'ensemble défini par induction.

Définition 3.6. Pour une définition inductive (B, R) définissant un ensemble X, et une valeur $x \in X$ ainsi définie, un *arbre de dérivation* (ou *preuve*) de x est un arbre dont les nœuds sont des éléments de X, tel que les feuilles sont des axiomes (des éléments de B), et pour tout nœud interne y dont les enfants sont x_1, x_2, ..., x_n, il existe une règle $r \in R$ d'arité n telle que $y = r(x_1, x_2, \ldots, x_n)$.

Non seulement l'arbre de dérivation prouve que la racine est un élément définie par la définition inductive, mais réciproquement, tout élément effectivement défini est racine d'un arbre de dérivation.

Proposition 3.7. Un élément x est défini par une définition inductive (B, R) si et seulement s'il existe un arbre de dérivation ayant x pour racine.

Démonstration. Commençons par démontrer que si x est racine d'un arbre de dérivation, alors x est bien élément de l'ensemble X défini par (B, R). Nous le faisons en démontrant par récurrence sur l'entier h, que pour tout arbre de dérivation valide de hauteur au plus h, sa racine est dans X. Pour le cas de base, nous considèrons un arbre de hauteur 0,

$$\overline{b}$$

C'est arbre est valide du moment que $b \in B$, auquel cas $b \in X$, prouvant l'hypothèse au rang 0. Soit donc un entier h arbitraire et supposons notre hypothèse vraie au rang h : tout arbre de dérivation de hauteur au plus h a pour racine un élément de X. Soit un arbre de hauteur $h + 1$.

8. c'est une blague fameuse d'informaticien : les logiciens dessinent leurs arbres à l'envers, ils mettent la racine en bas.

Notamment, au niveau de sa racine, nous avons

$$\frac{x_1 \qquad x_2 \qquad \dots \qquad x_n}{y}$$

Chacun des x_i est lui-même racine d'un sous-arbre de hauteur au plus h, donc par hypothèse de récurrence, $x_i \in X$. Puisque $x_1, x_2, \dots, x_n \in X$, l'application de la règle la plus basse nous donne que $y \in X$. Ainsi, la racine de tout arbre de hauteur $h+1$ est dans X, et tout comme les racines des arbres de hauteur au plus h par hypothèse de récurrence au rang h. Nous en déduisons l'hypothèse de récurrence au rang $h+1$, ce qui conclut notre preuve par récurrence : les racines des arbres de dérivations sont dans X.

Pour la réciproque, nous voulons montrer que tout élément y de X est racine d'un arbre de dérivation. Nous montrons par récurrence sur l'entier n que pour tout $y \in X_n$ (tel que défini dans la preuve du Théorème 3.5), y est racine d'un arbre de dérivation. Pour le cas de base, soit $b \in X_0 = B$, alors $\frac{}{b}$ est un arbre de dérivation valide dont b est racine. Pour le cas d'induction, soit n un entier arbitraire, supposons que tout élément de X_n est racine d'un arbre de dérivation. Soit y un élément arbitraire de X_{n+1}. Par définition de X_{n+1},

- ▷ ou bien $y \in X_n$, auquel cas y est par hypothèse de récurrence racine d'un arbre de dérivation ;
- ▷ ou bien $y = r(x_1, x_2, \dots, x_k)$ pour $r \in R$ une règle d'arité k, et $x_1, x_2, \dots, x_k \in X_n$. Par hypothèse de récurrence, chaque x_i est racine d'un arbre de dérivation T_i. Alors,

$$\frac{T_1 \qquad T_2 \qquad \dots \qquad T_k}{y}$$

est une arbre de dérivation ayant y pour racine.

Ainsi l'hypothèse de récurrence est vrai au rang $n+1$, terminant notre preuve par récurrence : pour tout $n \in \mathbb{N}$, les élements de X_n sont racines d'arbres de dérivation. Puisque $X = \bigcup_{n \geq 0} X_n$, c'est aussi vrai de tout élément de X. □

Ambiguïté. Les arbres de dérivation vont se révéler très utiles pour travailler avec des valeurs issues de définitions inductives, pour définir des fonctions ou des algorithmes, ce que nous ferons en Section 5, ou pour démontrer des propriétés de ces valeurs, en Section 5.4. Néanmoins pour cela, il nous faudra un propriété supplémentaire : l'unicité des arbres de dérivation.

Considérons la définition inductive suivante :

$$\frac{}{\varepsilon} \qquad \frac{}{\mathrm{a}} \qquad \frac{}{\mathrm{b}} \qquad \frac{x \qquad y}{x.y}$$

avec $x.y$ la concaténation des mots x et y. Nous pouvons nous convaincre assez facilement que l'ensemble ainsi défini est $\{a, b\}^\star$, l'ensemble des mots sur l'alphabet $\{a, b\}$. Par exemple, le mot abbbab est valide, et nous pouvons le démontrer de plusieurs façons par des arbres de dérivation :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{a}} \quad \dfrac{}{\mathrm{b}}}{\mathrm{ab}} \quad \dfrac{\dfrac{}{\mathrm{b}} \quad \dfrac{}{\mathrm{b}}}{\mathrm{bb}}}{\mathrm{abbb}} \quad \dfrac{\dfrac{}{\mathrm{a}} \quad \dfrac{}{\mathrm{b}}}{\mathrm{ab}}}{\mathrm{abbbab}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{a}} \quad \dfrac{}{\mathrm{b}}}{\mathrm{ab}} \quad \dfrac{}{\mathrm{b}}}{\mathrm{abb}} \quad \dfrac{\dfrac{}{\mathrm{b}} \quad \dfrac{}{\mathrm{a}}}{\mathrm{ba}}}{\mathrm{abbba}} \quad \dfrac{}{\mathrm{b}}}{\mathrm{abbbab}}$$

Définition 3.8. Une définition inductive (B, R) d'un ensemble X est *ambiguë* s'il existe un élément $x \in X$ tel que x est racine de deux arbres de dérivation distincts.

À l'inverse, pour une définition inductive non-ambiguë, il existe un seul arbre de dérivation possible, une seule preuve possible. Nous aurions pu donner une définition inductive non-ambiguë des mots sur l'alphabet $\{a, b\}$ ainsi :

$$\frac{}{\varepsilon} \qquad \frac{x}{\mathrm{a}.x}\,\mathrm{a} \qquad \frac{x}{\mathrm{b}.x}\,\mathrm{b}$$

Alors notre mot abbbab a un seul arbre de dérivation,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\varepsilon}\,\mathrm{b}}{\mathrm{b}}\,\mathrm{a}}{\mathrm{ab}}\,\mathrm{b}}{\mathrm{bab}}\,\mathrm{b}}{\mathrm{bbab}}\,\mathrm{b}}{\mathrm{bbbab}}\,\mathrm{a}}{\mathrm{abbbab}}$$

Pour montrer qu'une définition est non-ambiguë, nous pouvons nous appuyer sur la proposition suivante.

Proposition 3.9. Soit (B, R) une définition inductive telle que

(i) aucun axiome (de B) ne peut être produit par une règle (de R) ;

(ii) chaque élément peut être produit par au plus une règle ;

(iii) chaque règle est injective ;

alors (B, R) est non-ambiguë.

Démonstration. Soit x racine d'un arbre de dérivation T. Nous montrons par récurrence sur h que, pour tout sous-arbre S de T de hauteur au plus h, S a pour racine un élément admettant un seul arbre de dérivation. Pour $h = 0$, S est réduit à un axiome $\overline{\ b\ }$, et par (i), b n'est dérivable par aucune règle, donc possède un seul arbre de dérivation. Soit un entier naturel h arbitraire, et supposons l'hypothèse de récurrence vraie au rang h. Considérons un sous-arbre S de T de hauteur $h + 1$. Soit S' un arbre de dérivation arbitraire de même racine que S. Nous pouvons décrire les deux arbres

$$S = \dfrac{A_1 \quad A_2 \quad \dots \quad A_k}{\mathrm{y}} \qquad \text{et } S' = \dfrac{B_1 \quad B_2 \quad \dots \quad B_l}{\mathrm{y}}$$

Par (ii), c'est la même règle $r \in R$ qui produit y dans les deux arbres, et donc $k = l$. Par (iii), r est injective, ce qui implique que les racines de A_1, A_2, $\dots A_k$ coïncident respectivement avec les racines de B_1, B_2, …, B_k. Mais puisque A_1, A_2,…, A_k ont pour hauteur au plus h et sont des sous-arbres de T, par hypothèse de récurrence leurs racines admettent un seul arbre de dérivation. Donc $A_1 = B_1$, $A_2 = B_2$,…, $A_k = B_k$ et $S = S'$, prouvant l'hypothèse de récurrence au rang $n + 1$.

Ainsi, en appliquant l'hypothèse de récurrence à T, sa racine x admet un unique arbre de dérivation. □

À retenir.

- ▷ Une définition inductive est la donnée de *cas de base* ou *axiomes*, et de *règles d'induction* permettant de construire des termes de plus en plus complexes.
- ▷ Toute valeur définie par une définition inductive est *dérivable*, en appliquant les règles d'induction depuis les axiomes. Un *arbre de dérivation* est une trace de ce processus, prouvant l'appartenance de l'élément à l'ensemble ; toute valeur définie par la définition inductive est racine d'un arbre de dérivation.
- ▷ Nous préférons les définitions inductives *non-ambiguës*, celles pour lesquelles l'arbre de dérivation est toujours unique.
- ▷ Nous avons vu trois notations différentes pour les définitions inductives. Chacune possède ses avantages, et nous serons amener à utiliser les trois, selon le contexte.

Exercice 3.4. Considérons l'ensemble d'entiers définis inductivement par

$$\frac{}{0} \qquad \frac{n}{n+3} \qquad \frac{n}{n+5} \qquad \frac{n}{n+7}$$

Montrer que cette définition inductive est ambiguë, en donnant deux arbres de dérivation de l'entier 10.

```java
public sealed interface Sequence<T>
  permits EmptySequence, NonEmptySequence {}

public record EmptySequence<T>()
  implements Sequence<T> {}
public record NonEmptySequence<T>(T head, Sequence<T> tail)
  implements Sequence<T> {}
```

FIGURE 7 – Une implémentation des séquences d'objets de type `T` en Java. Nous retrouvons deux cas : soit la séquence est vide, soit elle contient un élément de tête `head`, et une séquence de queue `tail`.

4 Quelques définitions inductives classiques

4.1 Les séquences (ou listes)

Les séquences, ou mots, ou chaînes, ou listes, consistent en prendre un nombre arbitraire et fini k d'éléments d'un ensemble de base, possiblement avec des répétitions, et à les organiser selon un ordre total. Dans un ordre total, tous les éléments sont deux-à-deux comparables et distincts (si $x \neq y$, alors $x < y$ ou $y < x$). Choisir un tel ordre revient à indicer les k éléments de 1 à k (du plus petit au plus grand), ou de façon équivalente à placer les éléments sur une ligne de gauche à droite. Ainsi, $3, 7, 3, 2, 5, 5, 10, 0, 1$ est une séquence d'entiers (séparés par des virgules), et abbabab est un mot sur l'alphabet $\{a, b\}$ (conformément à l'usage, nous ne séparons pas les lettres d'un mot).

Nous pouvons distinguer deux cas, selon que la séquence vide est autorisée ou non. Si la séquence vide est autorisée, une séquence sur un ensemble Σ est définie par

$$\frac{}{\varepsilon}\ \text{vide} \qquad \frac{w}{\alpha w}\ \alpha \in \Sigma$$

Nous avons un cas de base, lorsque la séquence est vide, et une règle d'induction, consistant en l'ajout de n'importe quelle lettre en début de séquence. Pour $\Sigma = \{a, b\}$, nous avons vu que cela définit l'ensemble des mots sur cette alphabet. Mais nous pouvons l'appliquer sur un ensemble infini, par exemple $\mathbb{N}$, pour produire l'ensemble des séquences finies d'entiers.

Proposition 4.1. La définition inductive des séquences est non-ambiguë.

Démonstration. Nous utilisons la Proposition 3.9. Il n'y a qu'un axiome, produisant la séquence vide, et les règles d'induction génèrent toujours des séquences comportant au moins une lettre α, donc (i) est vrai. Pour une séquence $a_1 a_2 \dots a_k$, seule la règle d'induction pour a_1 appliquée à $a_2 \dots a_k$ peut l'avoir produite donc (ii) et (iii) sont vrais. □

Les séquences peuvent être réalisées en programmation, par exemple en Java selon la Figure 7.

Lorsque nous souhaitons nous restreindre à des séquences non-vides, il est possible d'utiliser cette définition plutôt.

$$\frac{}{\alpha}\ \alpha \in \Sigma \qquad \frac{w}{\alpha.w}\ \alpha \in \Sigma$$

Cette fois-ci, nous avons pour chaque élément de Σ, un axiome et une règle d'induction. L'axiome pour l'élément α donne la séquence singleton, de longueur un, contenant α. La règle d'induction ajoute α en tête de la séquence.

Proposition 4.2. La définition inductive des séquences non-vides est non-ambiguë.

```java
public sealed interface Sequence<T>
  permits SingletonSequence, LongSequence {}

public record SingletonSequence<T>(T element)
  implements Sequence<T> {}
public record LongSequence<T>(T head, Sequence<T> tail)
  implements Sequence<T> {}
```

FIGURE 8 – Une implémentation des séquences non-vides d'objets de type `T` en Java. Nous retrouvons deux cas : soit la séquence est un singleton contenant un élément unique `element`, soit elle contient un élément de tête `head`, et une séquence de queue `tail`.

Démonstration. Nous utilisons la Proposition 3.9. Notons que tout séquence définie sont de longueur au moins 1, puisque les conclusions de chaque règle et axiome contiennent toutes un élément de tête. Les axiomes, produisent des séquences de longueur 1, alors que les règles d'induction, ayant pour prémisses des séquences de longueur au moins 1, produisant des séquences de longueur au moins 2, donc (i) est vrai. Pour une séquence $a_1 a_2 \dots a_k$ avec $k \geq 2$, seule la règle d'induction pour a_1 appliquée à $a_2 \dots a_k$ peut l'avoir produite donc (ii) et (iii) est vrai. □

Les séquences non-vides peuvent aussi être réalisées en programmation, par exemple en Java selon la Figure 8.

Nous pouvons aussi remarquer que la définition inductive des séquences ressemble beaucoup à celle des entiers naturels : un axiome, plus un règle d'induction ajoutant un élément, tout comme le successeur ajoute 1 à l'entier naturel. C'est l'idée de la représentation en unaire des entiers : la plus naïve des écritures pour les entiers, dans laquelle l'entier n est écrit comme une séquence de n points.

4.2 Les algèbres de termes

Lors de la présentation de la Définition 3.3 des définitions inductives, nous avons préciser que E, l'ensemble dans lequel nous construisons des valeurs, peut être en toute généralité une algèbre de termes bien choisies. Une *algèbre de terme* est l'ensemble des formules sur une signature donnée ; une *signature* étant un ensemble de symboles de fonction ayant chacun une arité fixée. Par exemple, considérons les symboles a, b, c d'arité 0, f d'arité 1 et g d'arité 2. En respectant ces arités, nous pouvons construire des termes variées, comme b, $f(f(c))$, $g(f(a), g(b, f(f(c))))$. Donnons donc une définition inductive de l'ensemble de ces termes valides.

$$\frac{}{\mathrm{a}} \qquad \frac{}{\mathrm{b}} \qquad \frac{}{\mathrm{c}} \qquad \frac{x}{\mathrm{f}(x)} \qquad \frac{x \qquad y}{\mathrm{g}(x,y)}$$

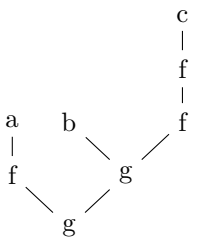
et une dérivation d'un des termes

$$\dfrac{\dfrac{\dfrac{}{\mathrm{a}}}{\mathrm{f(a)}} \qquad \dfrac{\dfrac{}{\mathrm{b}} \qquad \dfrac{\dfrac{\dfrac{}{\mathrm{c}}}{\mathrm{f(c)}}}{\mathrm{f(f(c))}}}{\mathrm{g(b,f(f(c)))}}}{\mathrm{g(f(a),\ g(b,f(f(c))))}}$$

Notons que la définition inductive est non-ambiguë, et que si nous gardons dans cet arbre de dérivation, uniquement le symbole ajouté par chaque règle, nous obtenons l'arbre suivant

```java
public sealed interface Term permits A, B, C, F, G {}
public record A() implements Term {}
public record B() implements Term {}
public record C() implements Term {}
public record F(Term x) implements Term {}
public record G(Term x, Term y) implements Term {}
```

FIGURE 9 – Une implémentation en Java d'une algèbre de terme.



qui est l'*arbre syntaxique* du terme.

Les algèbres de terme sont fréquemment utilisées en informatique, car elles permettent de donner une représentation de nombreux langages. Les expressions arithmétiques en sont un exemple ; les constantes et les entiers sont les termes d'arité 0, alors que $+$ et $\times$ sont les deux termes d'arité 2. Seulement nous avons l'habitude de la notation infixe pour ces deux opérateurs : nous écrivons $2 \times (x+1)$, plutôt que $\times(2, +(x, 1))$. Un autre cas d'usage est pour représenter les arbres binaires, essentiels en algorithmique. Pour les arbres binaires, nous avons deux symboles, le premier pour les feuilles d'arité 0, et le second pour les nœuds d'arité 2. Bien sûr, cela peut s'étendre pour définir des arbres ternaires ou plus ; ce sont tous des cas d'algèbre de terme. Nous reviendrons par contre sur les arbres d'arité quelconque, dans lesquels chaque nœud peut avoir un nombre d'enfants arbitraire.

Définition 4.3. Soit un ensemble de symbole de fonctions T et $a : T \to \mathbb{N}$ une fonction d'arité, tel que au moins un symbole $t \in T$ a une arité $a(t) = 0$. Alors l'algèbre de terme pour T et a est défini inductivement par les règles : pour tout t d'arité $k = a(t)$,

$$\frac{x_1 \qquad x_2 \qquad \dots \qquad x_k}{\mathrm{t}(x_1, x_2, \dots, x_k)}$$

Proposition 4.4. Les algèbres de terme sont non-ambiguës.

Démonstration. Chaque symbole de fonction ne peut être obtenu que par sa règle associée, y compris les axiomes, c'est-à-dire les symboles d'arité 0. En conséquence, (i), (ii) et (iii) sont vrais dans la Proposition 3.9, donc l'algèbre de terme est non-ambiguë. □

Programmer une algèbre de terme. Les algèbres de terme s'implémentent assez simplement dans les langages objets, en utilisant une interface, avec une implémentation par symbole possible. Notre exemple de début de section est présenté en Figure 9, un autre exemple est l'implémentation des expressions arithmétiques en Figure 6.

Dans les langages supportant des types algébriques (ou types unions), comme Swift ou Rust, chaque règle donne lieu à un *variant*, c'est-à-dire un cas dans le type algébrique. Ainsi le même exemple s'implémente en Ocaml avec 5 alternatives, comme le montre la Figure 10.

```
type term =
  | A
  | B
  | C
  | F of term
  | G of term * term
}

let value = G (F B, G (A,C))
```

FIGURE 10 – Une implémentation en Ocaml d'une algèbre de terme, avec les 5 variants possibles, puis un exemple de terme.

4.3 Les arbres d'arité non-fixe

Terminons en introduisant une définition inductive des arbres avec une arité quelconque, qui sont aussi classiquement utilisés. Par exemple, un document html est divisé en sections, elle-mêmes composées de sections et de paragraphes. La structure est hiérarchique, mais le nombre d'enfants à chaque niveau n'est pas limité. Nous ne sommes donc pas dans le cas d'une algèbre de terme de la section précédente, pour lesquels les termes ont une arité fixée, ou en tout cas bornée.

Prenons un exemple simple, avec comme axiomes les entiers naturels, et un unique terme f d'arité arbitraire. Nous pouvons donc écrire des formules comme $f(1)$, $f(f(1),0)$, $f(1,3,5,2,4)$ ou $f(0,f(2,f(3)),f(1))$. Voici une définition inductive pour ces formules :

$$\frac{}{\mathrm{n}}\ n \in \mathbb{Z} \qquad \frac{x}{\mathrm{f}(x)}\ \text{intro-f} \qquad \frac{x \qquad \mathrm{f}(y)}{\mathrm{f}(x,y)}\ \text{cons-f}$$

et un exemple de dérivation :

$$\dfrac{\dfrac{}{0} \qquad \dfrac{\dfrac{\dfrac{}{2} \qquad \dfrac{\dfrac{}{3}}{\mathrm{f}(3)}}{\mathrm{f}(2,\mathrm{f}(3))} \qquad \dfrac{\dfrac{}{1}}{\mathrm{f}(1)}}{\mathrm{f}(\mathrm{f}(2,\mathrm{f}(3)),\mathrm{f}(1))}}{\mathrm{f}(0,\mathrm{f}(2,\mathrm{f}(3)),\mathrm{f}(1))}$$

Notons que nous avons un ensemble d'axiomes pour les entiers, et deux règles d'induction. La première règle permet d'augmenter la hauteur de la hiérarchie, en ajoutant un nouvel ancêtre en racine. La deuxième règle permet d'ajouter un enfant supplémentaire à la racine actuelle.

Proposition 4.5. La définition inductive de l'arbre d'arité quelconque est non-ambiguë.

Démonstration. De nouveau nous utilisons la Proposition 3.9. Les axiomes sont les entiers, alors que les régles fournissent des formules de type $f(\ldots)$, donc (i) est vrai.

Considérons une valeur arbitraire v dans l'ensemble défini. Si v est un axiome, il n'est donc pas produit par une règle. Sinon il est de la forme $f(x_1, x_2, \ldots, x_k)$. Si $k = 1$, il ne peut être produit que par la règle intro-f, puisque la conclusion de règle cons-f a plusieurs arguments ($k > 2$). Ainsi, une seule règle a pu produire v, (ii) est vrai.

Finalement chaque règle est bien injective, notamment parce qu'elle sont toutes les deux inversibles : si la conclusion de intro-f est $\mathrm{f}(x)$, sa prémisse est nécessairement x, et si la conclusion de cons-f est $\mathrm{f}(x,y)$, ses prémisses sont x et $f(y)$. Donc (iii) est vrai. □

```java
public sealed interface Tree
  permits Node, Leaf {}
public sealed interface Node extends Tree
  permits SingletonNode, GeneralNode {}

public record SingletonNode(Tree child)
  implements Node {}
public record GeneralNode(Tree childHead, Node children)
  implements Node {}
public record Leaf(int n)
  implements Tree {}
```

FIGURE 11 – Une implémentation peu pratique d'un arbre d'arité non-fixe, avec des entiers aux feuilles.

```java
public sealed interface Tree permits Node, Leaf {}

public record Node(NonEmptySequence<Tree> children)
  implements Node {}
public record Leaf(int n)
  implements Tree {}
```

```haskell
data NonEmptyList element
  = SingletonList element
  | LongerList element (NonEmptyList element)

data Tree
  = Leaf Int
  | Node (NonEmptyList Tree)
```

FIGURE 12 – Une implémentation raisonnable d'un arbre d'arité non-fixe, utilisant une structure de données pour la séquence des enfants d'un nœud, en Java puis en Haskell.

Cette définition et la preuve de la proposition peut s'étendre à des arbres avec des termes plus variés, et il est même possible de mixer nœuds d'arité fixe et nœuds d'arité arbitraire.

La définition inductive peut être adaptée directement en un programme, comme nous le montrons en Figure 11. Cependant, il est plus pratique de s'éloigner un peu de la définition inductive, et d'adopter une séquence pour représenter les enfants d'un nœud. Cette séquence peut être elle-même issue d'une définition inductive ou reposer sur une collection fournie par le langage. Dans la Figure 12, nous choisissons de réutiliser les séquences non-vides définies en Section 4.1.

Exercice 4.1. Écrire une définition inductive pour représenter les formules propositionnelles en logique, permettant d'écrire des formules comme

$$\forall x.\exists y.(x \wedge \neg z) \vee (\text{vrai} \implies \neg x).$$

$\vee$, $\wedge$, $\implies$ sont les opérateurs binaires de disjonction (*ou*), conjonction (*et*) et implication. $\neg$ est l'opérateur unaire de négation. $\exists$ et $\forall$ sont les quantificateurs *il existe* et *pour tout*, permettant d'introduire une variable.

5 Fonctions et algorithmes sur les structures inductives

5.1 Définitions inductives et ordres bien-fondés

Lorsque nous avons expliqué le principe des définitions inductives, et notamment la construction du plus petit ensemble clos par les règles d'induction, nous avons introduit X_n l'ensemble des valeurs obtenus après n opérations. Ainsi $X_n \setminus X_{n-1}$ sont les termes construits à l'étape n ; nous pouvons appelé cet ensemble la n^{e} génération. Lorsque la définition inductive est non-ambiguë, la n^{e} génération est en fait la hauteur de l'unique arbre de dérivation. Cette partition en générations nous donne un ordre sur les termes : certains sont construits plus tôt que d'autres. Un terme est plus petit s'il apparait avant dans notre construction de tous les termes. Non seulement c'est un ordre (partiel), mais comme il est induit par une fonction vers $\mathbb{N}$, c'est un ordre bien-fondé, que nous pouvons exploiter par le principe de récurrence.

Cependant, nous pouvons faire presque aussi bien, sans avoir à parler de hauteur d'arbre de dérivation ou de génération, en montrant plus directement l'existence d'un ordre bien-fondé directement basé sur les règles d'induction, et qui sera plus adapté pour l'usage que nous ferons du principe de récurrence.

Définition 5.1. Soit X un ensemble défini par une définition inductive (B, R) **non-ambiguë**. On définit la relation de *dérivation* $x \preceq_X y$, pour $x, y \in X$, lorsque x apparait comme conclusion d'une règle dans l'arbre de dérivation de y.

Lorsque l'ensemble est clairement identifié, nous noterons simplement $\preceq$ cette relation de dérivation.

Exemple des entiers naturels. En considérant l'arbre de dérivation de $4 = S(S(S(S(Z))))$, nous obtenons

$$Z \preceq S(Z) \preceq S(S(Z)) \preceq S(S(S(Z))) \preceq S(S(S(S(Z)))).$$

Plus généralement, $\preceq$ coïncide avec l'ordre naturel $\leq$ des entiers.

Exemple des expressions arithmétiques. Considérons la formule $(2+x)\times(3+y)$, avec son arbre de dérivation

$$\dfrac{\dfrac{\dfrac{}{2} \quad \dfrac{}{x}}{2+x} \quad \dfrac{\dfrac{}{3} \quad \dfrac{}{y}}{3+y}}{(2+x)\times(3+y)}$$

Nous observons que $2 \preceq 2+x$ et $x \preceq 2+x$, $3 \preceq 3+y$ et $y \preceq 3+y$, et enfin pour toute formule ϕ dans $\{2, x, 3, y, 2+x, 3+y\}$, $\phi \preceq (2+x)\times(3+y)$. Une formule est plus grande que toutes les sous-formules qu'elle contient.

Notons que nous exigeons que la définition inductive soit non-ambiguë. Pour comprendre cette restriction, nous pouvons prendre la définition inductive suivante, qui définit le sous-ensemble des entiers relatifs impairs :

$$\dfrac{}{1} \qquad \dfrac{n}{n-2} \qquad \dfrac{n}{n+2}$$

Les entiers impairs s'obtiennent en ajoutant ou soustrayant 2 suffisamment de fois depuis 1. Voici un arbre de dérivation de 3 :

$$\dfrac{\dfrac{\dfrac{}{1}}{-1}}{\dfrac{1}{3}}$$

Nous remarquons que 1 apparaît dans l'arbre de dérivation de -1, donc $1 \preceq -1$. Mais par ailleurs, -1 apparaît aussi dans un arbre de dérivation de 1 (en considérant l'occurence de 1 qui est prémisse de la dernière règle dont la conclusion est 3). Donc $-1 \preceq 1$. Il en découle que $\preceq$ n'est pas antisymétrique, donc n'est pas un ordre. Avec l'exigence de non-ambiguïté, nous avons

Proposition 5.2. La relation de dérivation $\preceq$ pour une définition inductive non-ambiguë est un ordre bien-fondé.

Cet ordre peut être partiel, comme dans le cas des formules (x et 3 sont incomparables par exemple), ou total, comme dans le cas des entiers naturels.

Démonstration. Soit X l'ensemble défini par une définition inductive (B, R) non-ambiguë, et $\preceq$ sa relation. Pour un élément $x \in X$, nous notons T_x son arbre de dérivation, qui est uniquement défini par l'hypothèse de non-ambiguïté. Nous aurons besoin de l'affirmation suivante.

Fait 5.3. Pour tous x et y dans X, si $x \preceq y$, alors T_x est sous-arbre de T_y.

Démonstration. Puisque $x \preceq y$, T_y contient une règle dont la conclusion est x. Alors le sous-arbre de T_y enraciné en ce nœud x est un arbre de dérivation pour x, donc est T_x par non-ambiguïté. Ainsi T_x est sous-arbre de T_y. □

Reprenons la preuve principale, et vérifions que $\preceq$ est un ordre :

- ▷ réflexivité : tout x est conclusion de la dernière règle dans l'arbre de dérivation de x, donc $x \preceq x$;
- ▷ antisymétrie : considérons deux éléments x, y de X, tel que $x \preceq_X y$ et $y \preceq_X x$. Nous devons montrer que $x = y$. Puisque $x \preceq y$, T_x est contenu dans T_y. De même, puisque $y \preceq x$, T_y est contenu dans T_x. Puisque T_x et T_y sont finis, cela signifie que T_x et T_y ont le même nombre de nœuds, et comme l'un est sous-arbre de l'autre, ils sont égaux. Leur racine est alors $x = y$.
- ▷ transitivité : soient x, y, z trois éléments de X, avec $x \preceq y$ et $y \preceq$. Nous devons montrer que $x \preceq z$. Puisque $x \preceq y$, x apparaît en conclusion d'une règle dans T_y. Puisque $y \preceq z$, T_y est sous-arbre de T_z. Or T_y contient x, donc T_z contient aussi x, prouvant que $x \preceq z$.

Nous terminons en montrant que $\preceq$ est bien-fondé. Soit $U \subseteq X$ un sous-ensemble de X, et soit $x \in U$ minimisant le nombre de nœuds dans T_x. Alors x est un élément minimal de U : en effet, pour tout $y \in U$, $y \preceq x$ implique que T_y est un sous-arbre de T_x, or T_x a un nombre minimum de nœud, donc T_y est un sous-arbre de T_x de même taille que T_x, autrement dit $T_y = T_x$, et donc leur racine est $x - y$. Puisque tout sous-ensemble de X contient un élément minimal, X est bien-fondé. □

Puisque l'ensemble défini inductivement de manière non-ambiguë est bien-fondé, nous pouvons exploiter l'ordre bien-fondé et en particulier le principe de récurrence associé à cet ordre bien fondé. Cela va nous permettre d'écrire des preuves, ce que nous ferons en Section 5.4. Dans l'immédiat, nous allons en profiter pour proposer une méthode pour définir simplement des fonctions sur les ensembles définis inductivement, et la propriété de fondation de l'ordre nous fournira un argument de terminaison pour nos fonctions qui seront donc facilement réalisables algorithmiquement.

À retenir. Une définition inductive non-ambiguë engendre un ordre bien-fondé sur les valeurs définies. La contrainte de non-ambiguïté est importante. L'ordre est que $x \preceq y$ si x apparaît dans l'arbre de dérivation de y.

Exercice 5.1. Reprenons la définition de la représentation binaire des entiers de l'Exercice 3.2. Donner la liste de toutes les paires x, y avec $x \preceq y$, parmi les éléments

$$0, 1, 10, 11, 101, 110, 111, 1001.$$

Vous pouvez représenter le résultat sous la forme d'un graphe orienté, avec un arc $x \to y$ lorsque $x \preceq y$.

5.2 Fonctions définies inductivement

Nous montrons maintenant comment utiliser les définitions inductives pour définir des fonctions ayant pour domaine un ensemble défini inductivement. Puisque les définitions inductives reposent sur les règles d'induction, que nous espérons être individuellement assez simple, nous souhaitons nous appuyer sur ces règles pour définir nos fonctions.

Exemples introductifs. Afin d'introduire la technique, examinons un premier exemple sur la définition inductive des entiers naturels, en définissant la classique fonction factorielle. Pour cela nous réutilisons la notation en règle, en rajoutant la syntaxe $x : r$ signifiant que l'image de x par la fonction est r. Voici comment nous présentons la factorielle en tant que fonction définie inductivement.

$$\frac{}{\mathrm{Z} : 1} \qquad \frac{n : r}{\mathrm{S}(n) : (n+1) \times r}$$

La première règle exprime que l'image de Z par la fonction factorielle est 0. La deuxième règle exprime que si l'image de n par la fonction factorielle est r, alors l'image de $S(n)$ par cette même fonction est $(n+1) \times r$.

En utilisant l'arbre de dérivation, et en l'annotant intégralement avec cette règle sur les images par la fonction factorielle, nous obtenons, pour la factorielle de 4,

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathrm{Z} : 1}}{\mathrm{S}(\mathrm{Z}) : 1 \times 1}}{\mathrm{S}(\mathrm{S}(\mathrm{Z})) : 2 \times (1 \times 1)}}{\mathrm{S}(\mathrm{S}(\mathrm{S}(\mathrm{Z}))) : 3 \times (2 \times (1 \times 1))}}{\mathrm{S}(\mathrm{S}(\mathrm{S}(\mathrm{S}(\mathrm{Z})))) : 4 \times (3 \times (2 \times (1 \times 1)))}$$

ce qui après simplification nous donne que $4! = 24$.

Prenons un deuxième exemple avec les formules arithmétiques, et définissons la fonction qui calcule la valeur d'une expression arithmétique lorsque la valeur des variables est donnée par $\phi : \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots, \mathrm{z}\} \to \mathbb{Z}$.

$$\frac{}{n : n}\ n \in \mathbb{Z} \qquad \frac{}{\alpha : \phi(\alpha)}\ \alpha \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots, \mathrm{z}\} \qquad \frac{e_1 : n_1 \qquad e_2 : n_2}{e_1 + e_2 : n_1 + n_2} \qquad \frac{e_1 : n_1 \qquad e_2 : n_2}{e_1 \times e_2 : n_1 \times n_2}$$

Ici l'opérande droite du symbole **:** doit être évaluée comme un entier. Par exemple, pour $\phi(x) = 3$ et $\phi(y) = -1$, l'évaluation de l'expression $(x+2) \times (3+y)$ se représente ainsi :

$$\dfrac{\dfrac{\dfrac{}{x : 3} \quad \dfrac{}{2 : 2}}{x + 2 : 5} \qquad \dfrac{\dfrac{}{3 : 3} \quad \dfrac{}{y : -1}}{3 + y : 2}}{(x+2) \times (3+y) : 10}$$

Définition de fonctions. Pour que la définition de la fonction soit purement locale à chaque règle d'induction, l'image de la conclusion ne doit dépendre que des prémisses et de leurs images. Par exemple, pour la règle du successeur dans la définition de la factorielle, l'image est $(n+1) \times r$, avec n une prémisse et r l'image de la prémisse r. L'image de la conclusion est donc une fonction des prémisses et des images des prémisses. Ceci est capturé par la fonction g_r dans la définition suivante.

Définition 5.4 (Fonction définie inductivement)**.** Soit (B, R) une définition inductive **non-ambiguë** d'un ensemble X. Une *fonction définie inductivement* f de domaine X vers l'ensemble Y est la donnée de deux familles γ et g avec

- ▷ (Base) pour chaque axiome $x \in B$, une image $f(x) := \gamma_x \in Y$;
- ▷ (Induction) pour chaque règle $r \in R$ d'arité k, une fonction $g_r : X^k \times Y^k \to Y$, de sorte

que
$$f(r(x_1, x_2, \ldots, x_k)) := g_r(x_1, x_2, \ldots, x_k, f(x_1), f(x_2), \ldots, f(x_k)).$$

Cette définition correspond à la représentation suivante

$$\frac{}{x : \gamma_x}\, x \in B \qquad \frac{x_1 : v_1 \qquad x_2 : v_2 \qquad \ldots \qquad x_k : v_k}{x : g_r(x_1, x_2, \ldots, x_k, v_1, v_2, \ldots, v_k)}$$

Comme nous l'avons annoncé, avoir adossé notre définition de fonction à un ordre bien-fondé permet de prouver que la fonction f est bien définie.

Proposition 5.5. f est bien définie : pour tout $x \in X$, il existe une unique image $f(x)$ dans Y.

Démonstration. Nous le prouvons par récurrence sur l'ordre bien-fondé $\preceq$ de X. Les éléments minimaux de X sont les axiomes. Puisque (B, R) est non-ambiguë, un axiome $x \in B$ n'est dérivable que par la règle de base qui lui est associé, et donc $f(x)$ est bien définie une et une seule fois.

Soit $y \in X$ un élément qui n'est pas un axiome, et supposons que pour tout $x \preceq y$, $f(x)$ est uniquement définie. Il existe une unique règle d'induction $r \in R$ permettant d'obtenir y depuis des prémisses $x_1, x_2, \ldots, x_r$, ce qui s'écrit

$$\frac{x_1 \qquad x_2 \qquad \ldots \qquad x_k}{y}\, r$$

Par définition de $\preceq$, $x_1 \preceq y$, $x_2 \preceq y$,..., $x_k \preceq y$. Par hypothèse d'induction bien-fondée, $f(x_1)$, $f(x_2)$,..., $f(x_n)$ sont uniquement définis. Alors le cas d'induction dans la Définition 5.4 affirme l'existence d'une fonction g_r associée à cette règle, définissant donc

$$f(y) := g_r(x_1, x_2, \ldots, x_k, f(x_1), f(x_2), \ldots, f(x_k)).$$

Par unicité de l'arbre de dérivation, $f(y)$ est ainsi uniquement défini. Ceci termine notre preuve par récurrence bien-fondée. □

Notation pour les définitions inductives de fonctions. Nous pourrons utiliser la notation plus claire, consistant à donner directement dans les règles le nom de la fonction. Reprenant nos exemples, cela nous donne pour la factorielle :

$$\frac{}{\text{fact}(Z) : 1} \qquad \frac{n : r}{\text{fact}(S(n)) : (n+1) \times r}$$

et pour l'évaluation d'une expression arithmétique :

$$\frac{}{\text{eval}(n) : n}\, n \in \mathbb{Z} \qquad \frac{}{\text{eval}(\alpha) : \phi(\alpha)}\, \alpha \in \{\text{a}, \text{b}, \text{c}, \ldots, \text{z}\}$$

$$\frac{\text{eval}(e_1) : n_1 \qquad \text{eval}(e_2) : n_2}{\text{eval}(e_1 + e_2) : n_1 + n_2} \qquad \frac{\text{eval}(e_1) : n_1 \qquad \text{eval}(e_2) : n_2}{\text{eval}(e_1 \times e_2) : n_1 \times n_2}$$

La fonction est en fait définie inductivement sur son argument, et nous pourrons plus généralement définir des fonctions ayant plusieurs arguments, avec une induction sur un argument particulier seulement.

Dans tous les cas, cette notation, faisant clairement apparaître le nom de la fonction, a l'avantage d'être plus lisible, et nous pourrons donc la privilégier.

Calculer une fonction définie inductivement. Les fonctions définies inductivement ont l'avantage d'être simple à décrire, puisque localement définies pour chaque règle d'induction. Un autre avantage est que cela les rend calculable avec un algorithme simple, pourvu qu'on sache pour un élément $y \in X$, calculer la règle r est les prémisses $x_1, x_2, \ldots, x_k$ tels que $r(x_1, x_2, \ldots, x_k) = y$.

factorielle(n)
Entrée: un entier $n \geq 0$;
Sortie: la factorielle $n!$ de n.
1: **si** $n = Z$ **alors** ▷ $n = 0$
2: **renvoyer** 1
3: **soit** p tel que $S(p) = n$ ▷ $p = n - 1$
4: **renvoyer** $n \times$ factorielle(p)

Algorithme 5.1 – L'algorithme inductif pour le calcul de la factorielle.

évaluer(e, ϕ)
Entrée: une expression arithmétique e, une affectation $\phi : \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots, \mathrm{z}\} \to \mathbb{Z}$;
Sortie: la valeur de e lorsque les valeurs des variables sont données par ϕ.
1: **si** $e \in \mathbb{Z}$ **alors**
2: **renvoyer** e
3: **si** $e \in \{\mathrm{a}, \mathrm{b}, \mathrm{c}, \ldots, \mathrm{z}\}$ **alors**
4: **renvoyer** $\phi(e)$
5: **si** $e = e_1 + e_2$ **alors**
6: **renvoyer** évaluer(e_1, ϕ) + évaluer(e_2, ϕ)
7: **soit** $e = e_1 \times e_2$
8: **renvoyer** évaluer(e_1, ϕ) $\times$ évaluer(e_2, ϕ)

Algorithme 5.2 – L'algorithme inductif d'évaluation d'une expression arithmétique.

Les Algorithmes 5.1 et 5.2 calculent respectivement la factorielle d'un entier naturel, et la valeur d'une expression arithmétique selon l'affectation des variables. Une suite de conditionnelle permet de repérer si l'élément est un axiome ou bien dérive d'une règle d'induction. Dans le cas d'un axiome, l'algorithme termine avec la valeur associée à cet axiome. Dans le cas d'une règle d'induction, il faut déterminer les premisses, ce qui dans ces deux exemples est facile. Puis l'algorithme procède en s'appelant récursivement pour chaque prémisse, calculant leurs images respectives. Enfin, il retourne l'image par la fonction g_r associée à la règle inductive, en utilisant les prémisses et leurs images selon les besoins de g_r

$f(x)$
Entrée: un élément x d'un ensemble X défini inductivement par (B, R) non-ambiguë, une fonction f définie inductivement par $(\gamma_{b \in B}, g_{r \in R})$.
Sortie: l'image $f(x)$.
1: **si** $x \in B$ **alors**
2: **renvoyer** γ_x
3: **soit** $x_1, x_2, \ldots, x_k$ et $r \in R$ avec $r(x_1, x_2, \ldots, x_k) = x$
4: **pour** $i \in \{1, 2, \ldots, k\}$ **faire**
5: **soit** $y_i = f(x_i)$
6: **renvoyer** $g_r(y_1, y_2, \ldots, y_k)$

Algorithme 5.3 – L'algorithme générique d'évaluation d'une fonction définie inductivement.

L'Algorithme 5.3 est la forme générale de cette procédure. Il convient de préciser comment trouver $r \in R$ et les prémisses $x_1, x_2, \ldots, x_n$ au cas par cas pour chaque définition récursive qui nous intéresse. Ainsi, le calcul d'une fonction définie inductivement est réduit au problème de trouver la règle de dérivation permettant d'obtenir l'élément x de l'ensemble défini inductivement.

Proposition 5.6. Pour une fonction définie inductivement sur un ensemble X elle-même définie inductivement de façon non-ambiguë, l'Algorithme 5.3 termine, sous la condition que

```java
public static int factorial(int n) {
  return n == 0? 1:
                n * factorial(n-1);
}
```

FIGURE 13 – L'implémentation de l'algorithme inductif de la factorielle sur le type primitif des entiers en Java.

r et $x_1, x_2, \dots x_n$ soient calculables.

Démonstration. Puisque l'ensemble X défini inductivement est bien fondé, cela se déduit directement du théorème de terminaison des algorithmes récursifs sur les ensembles bien-fondés, en remarquant que les appels récursifs pour $x \in X$ se font sur les prémisses de la règle produisant x, qui sont bien plus petites que x dans l'ordre $\preceq$. □

Programmer une fonction définie inductivement. En pratique, les définitions inductives dans nos programmes suivent structurellement les règles d'induction, comme dans le cas des algèbres de terme, en ayant un variant ou une classe par règle possible. Retrouver la règle d'induction produisant une valeur est donc trivial, et les algorithmes définis inductivement sont donc implémentables sans difficulté. Nous l'illustrons avec nos deux exemples.

Réaliser la factorielle avec le type primitif des entiers est assez simple. La fonction se définit inductivement ainsi

$$\frac{}{0 : 1} \qquad \frac{n : r}{n+1 : (n+1) \times r}$$

et s'implémente par une fonction récursive, par exemple en Java en Figure 13.

Plus intéressant, implémentons la factorielle pour notre type inductif des entiers naturels, avec pour type de retour le type primitif `int`. Dans ce cas, nous avons aussi besoin de convertir un entier naturel défini inductivement vers le type de donnée entier natif de notre langage pour pouvoir effectuer les multiplications. Cette conversion peut aussi se décrire comme une fonction définie inductivement.

$$\frac{}{\mathrm{Z} : 0} \qquad \frac{n : r}{\mathrm{S}(n) : r+1}$$

Les deux fonctions peuvent alors être implémentées. En Java, deux options se présentent. Celle ressemblant le plus à l'algorithme générique (l'Algorithme 5.3) consiste à écrire une méthode par défaut pour chaque fonction, au niveau de l'interface définissant les entiers naturels. Chaque méthode cherche ensuite la règle d'induction à appliquer en utilisant la structure de contrôle `switch`. Cette dernière permet de déterminer la classe implémentée par l'objet courant. C'est aussi cette option que l'on retrouve dans des langages non-objets, par exemple en Haskell, qui a l'élégance de nous autoriser à filtrer les cas directement sur les paramètres de la fonction. Les deux implémentations sont données en Figure 14.

L'autre option est d'implémenter chaque règle de la fonction définie inductivement, dans la classe correspond à cette règle d'induction. Cela implique que la fonction n'est plus définie en un seul bloc d'instructions, dans un fichier du programme. Au contraire, elle se retrouve répartie sur toutes les classes implémentant le type inductif. Une fois passée nos éventuelles réticences à séparer notre fonction entre plusieurs fichiers, cette solution est finalement plus idiomatique dans les langages objets, et nous évite l'usage du `switch`. La sélection de la règle d'induction appropriée est effectuée par les mécanismes de polymorphisme de la programmation objet. Pour la factorielle, cela nous donne l'implémentation en Figure 15. Notons les encadrés, qui sont exactement les mêmes que ceux en Figure 14 : l'algorithme est le même, seule l'organisation du programme est différente.

```java
public sealed interface NaturalNumber permits Zero, Successor {
  default int factorial() {
    return switch (this) {
      case Zero z ->  1 ;
      case Successor s ->  this.toInt() * s.n().factorial() ;
    }
  }
  default int toInt() {
    return switch (this) {
      case Zero z ->  0 ;
      case Successor s ->  1 + s.n().toInt() ;
    }
  }
}
public record Zero() implements NaturalNumber {}
public record Successor(NaturalNumber n) implements NaturalNumber {}
```

```haskell
data Nat -- rappel du type des entiers naturels inductifs
  = Z
  | Successor Nat

toInt Z = 0
toInt (Successor n) = 1 + toInt n

factorial Z = 1
factorial (Successor n) = toInt (Successor n) * factorial n
```

FIGURE 14 – En haut, une implémentation de la factorielle et de la conversion vers le type `int` de Java pour les entiers naturels définis inductivement. Cette implémentation utilise un `switch` pour trouver la règle d'induction correspondant à la valeur de `this`. En bas, une implémentation équivalente en Haskell, avec la définition du type inductif des entiers, puis la conversion vers le type natif `Int` et la factorielle.

```java
public sealed interface NaturalNumber permits Zero, Successor {
  int factorial();
  int toInt();
}
public record Zero() implements NaturalNumber {
  public int factorial() { return  1 ; }
  public int toInt() { return  0 ; }
}
public record Successor(NaturalNumber n) implements NaturalNumber {
  public int factorial() { return  this.toInt() * n.factorial() ; }
  public int toInt() { return  n.toInt() + 1 ; }
}
```

FIGURE 15 – Une implémentation de la factorielle et de la conversion vers le type `int` de Java pour les entiers naturels définis inductivement. Chaque méthode est implémentée dans chaque classe permise par l'interface.

```java
public sealed interface Expression
  permits Litteral, Variable, Sum, Product
{
  int eval(ToIntFunction<Character> context);
}
public record Litteral(int n) implements Expression {
  public int eval(ToIntFunction<Character> context) {
    return n;
  }
}
public record Variable(char c) implements Expression {
  public int eval(ToIntFunction<Character> context) {
    return context.applyAsInt(c);
  }
}
public record Sum(Expression leftOperand, Expression rightOperand)
  implements Expression
{
  public int eval(ToIntFunction<Character> context) {
    return leftOperand.value(context) + rightOperand.value(context);
  }
}
public record Product(Expression leftOperand, Expression rightOperand)
  implements Expression
{
  public int eval(ToIntFunction<Character> context) {
    return leftOperand.value(context) + rightOperand.value(context);
  }
}
```

FIGURE 16 – Une implémentation en Java de l'évaluation d'une formule arithmétique. Il est possible de factoriser les deux opérateurs binaires d'addition et de multiplication, en introduisant un `enum`, qui définit une méthode `double eval(double,double)` permettant de réaliser l'opération arithmétique représentée, voir aussi la Figure 6.

Implémentons maintenant l'évaluation d'une expression arithmétique dans un contexte donnant les valeurs des variables. Ce contexte, qui correspond à l'affectation ϕ dans l'Algorithme 5.2, est représenté par une fonction du type `char` vers le type `int`, pour cela nous utilisons comme type l'interface fonctionnelle `ToIntFunction<Character>`, qui définit une méthode `int applyAsInt(Character)`. Nous reprenons la technique d'implémenter la méthode `eval` dans chaque classe permise par l'interface `Expression`. Le programme est présenté en Figure 16.

À retenir. Nous pouvons décrire inductivement des algorithmes sur des données qui sont elles-mêmes définies inductivement. Ces algorithmes peuvent alors s'implémenter inductivement dans les principaux langages de programmation. La terminaison dérive automatiquement de l'ordre bien-fondé hérité de la définition inductive.

En Java, l'algorithme inductif est réparti dans les différentes implémentations possibles du type inductif.

Exercice 5.2. En utilisant la définition inductive des entiers naturels, proposer une définition de la fonction $n \to 2 \times n$. Ajouter cette fonction dans l'implémentation Java de la Figure 15, sous la forme d'une méthode de signature `NaturalNumber double()`.

Exercice 5.3. Définir une fonction inductivement, permettant de calculer l'entier correspondant à une représentation binaire telle que définie dans l'Exercice 3.2. Vous utiliserez le fait qu'ajouter un 0 en fin de notation multiplie l'entier représenté par 2.

Exercice 5.4. Donner une définition inductive de l'ensemble des mots sur l'alphabet $\{a, b, c\}$, puis de la fonction qui a un mot associe le mot obtenu en remplaçant chaque a par un b, et chaque b par un a.

5.3 Fonctions inductives à plusieurs arguments

Jusque là, toutes les fonctions que nous avons définie n'ont qu'un seul argument, et ont pour domaine l'ensemble défini inductivement. Nous pouvous néanmoins étendre les définitions inductives de fonctions pour supporter la définition de fonctions prenant deux arguments ou plus. Nous allons présenter plusieurs sortes de définitions inductives de fonctions à deux arguments, ces techniques pouvant facilement s'étendre et se mixer pour pouvoir définir des fonctions avec trois arguments ou plus.

Fonctions inductives avec un deuxième argument fixe. Il s'agit de définir une fonction à deux arguments, mais le deuxième ne joue aucun rôle dans l'induction. Le premier est pris dans l'ensemble défini inductivement, et est celui qui est utilisé par la définition inductive de la fonction. Le deuxième argument ne varie pas dans les règles d'induction : il est le même dans les prémisses et dans la conclusion de chaque règle. Voici un exemple, permettant de définir l'addition de deux entiers naturels :

$$\frac{}{0 + n : n} \qquad \frac{p + n : q}{S(p) + n : S(q)}$$

L'induction se fait sur le premier argument p, et le deuxième argument est toujours n. En fait, nous pouvons lire cette définition comme celle d'une fonction qui ajoute n à un entier, que nous aurions pu définir tout-à-fait légalement

$$\frac{}{0 : n} \qquad \frac{p : q}{S(p) : S(q)}$$

Nous sommes donc toujours dans le cas, certes un peu masqué, de la définition d'une fonction à un seul argument.

Fonctions inductives avec un deuxième argument qui varie. Dans ce cas, l'induction se fait aussi sur le premier argument, pris dans l'ensemble défini inductivement. Mais par contre, le deuxième argument n'est pas dans les prémisses et la conclusion de chaque règle. Nous pouvons l'illustrer avec une autre définition de l'addition.

$$\frac{}{0 + n : n} \qquad \frac{p + S(n) : q}{S(p) + n : q}$$

Cette définition n'est pas légale au vu de notre définition des fonctions inductives. Néanmoins, de telles définitions sont courantes en programmation, nous aimerions donc les justifier mathématiquement. Nous allons nous en sortir, de nouveau en nous ramenant à une fonction ayant un seul argument, ayant comme domaine l'ensemble défini inductivement.

Pour cela, nous utilisons une technique appelée *curryfication*, qui permet de transformer une fonction à deux arguments en une fonction à un argument. Soit une fonction $f : (x, y) \to f(x, y)$, de domaine $X \times Y$. Pour tout $x \in X$, nous définissons une fonction g_x par $g_x(y) = f(x, y)$. Ainsi $(g_x)_{x \in X}$ est une famille (infinie si X l'est) de fonctions à un seul argument. Nous posons donc

maintenant $h : x \to g_x$ une nouvelle fonction. C'est cette fonction h qui peut alors être définie inductivement, en transformant la règle

$$\frac{(x_1, \phi_1(x,y)) : z_1 \qquad (x_2, \phi_2(x,y)) : z_2 \qquad \dots \qquad (x_k, \phi_k(x,y)) : z_k}{(x,y) : \psi(x_1, \dots, x_k, y, z_1, \dots, z_k)}$$

en

$$\frac{x_1 : g_{x_1} \qquad x_2 : g_{x_2} \qquad \dots \qquad x_k : g_{x_k}}{x : (y \to \psi(x_1, \dots, x_k, y, g_1(\phi_1(x,y)), \dots, g_k(\phi_k(x,y))))}$$

Appliqué à notre exemple de l'addition des entiers naturels, nous obtenons

$$\frac{}{0 : n \to n} \qquad \frac{p : h}{S(p) : (q \to h(S(q)))}$$

Pour nos règles d'induction, la curryfication a tendance à rendre les notations plus confuses, nous nous autoriserons donc à écrire des définitions inductives de fonctions à plusieurs arguments sans curryfier. L'important est de bien identifier un argument sur lequel porte l'induction.

Voici un exemple plus subtil, une définition illégale de la fonction du minimum de deux entiers naturels :

$$\frac{}{\min(0,n) : 0} \qquad \frac{}{\min(n,0) : 0} \qquad \frac{\min(n,p) : q}{\min(S(n), S(p)) : S(q)}$$

Cette définition inductive n'est pas valide : nous n'avons pas de correspondance entre les règles de la définition de $\mathbb{N}$ (ayant un axiome et une règle d'induction) et celle de min (qui a deux cas d'axiomes et une règle d'induction). Pourtant elle semble parfaitement saine, et correspond même à l'Algorithme 5.4

$\min(n,p)$
Entrée: deux entiers naturels n et p ;
Sortie: le minimum de n et p.
1: **si** $n = 0$ **alors**
2: ⌊ **renvoyer** 0
3: **si** $p = 0$ **alors**
4: ⌊ **renvoyer** 0
5: **renvoyer** $\min(n-1, p-1) + 1$

Algorithme 5.4 – Un algorithme étrange pour calculer le minimum de deux entiers.

Dans cet exemple, les deux arguments décroissent lors de l'appel récursif à la dernière ligne. Une analyse un peu plus poussée montre cependant que cet exemple peut aussi se comprendre en l'analysant uniquement du point de vue du premier argument : nous pouvons l'écrire en curryfiant

$$\frac{}{0 : (n \to 0)} \qquad \frac{n : h}{S(n) : \begin{cases} 0 \to 0 \\ S(q) \to S(h(q)) \end{cases}}$$

Mais nous pouvons aussi analyser des fonctions où chaque argument subit sa propre induction, et qui ne peuvent être réduites à l'analyse d'un seul argument bien choisi.

Inductions doubles. Il est assez rare, mais cela arrive, d'écrire des programmes nécessitant une double induction : deux arguments proviennent d'un type inductif, et ce n'est pas toujours le même argument qui décroit lors des appels récursifs. C'est le cas de cette fonction de fusion de deux listes triées en ordre croissant. Donnons un type des listes triées d'entiers

$$\frac{}{[]} \qquad \frac{}{n :: []}\ n \in \mathbb{Z} \qquad \frac{h :: t}{n :: h :: t}\ n < h$$

et voici la fonction de fusion

$$\frac{}{\text{fusion}([], l) : l} \qquad \frac{}{\text{fusion}(l, []) : l}$$

$$\frac{\text{fusion}(t_1, h_2 :: t_2) : t}{\text{fusion}(h_1 :: t_1, h_2 :: t_2) : h_1 :: t}\ h_1 \leq h_2 \qquad \frac{\text{fusion}(h_1 :: t_1, t_2) : t}{\text{fusion}(h_1 :: t_1, h_2 :: t_2) : h_2 :: t}\ h_1 > h_2$$

Cette définition donne lieu à l'Algorithme 5.5. Ici les appels récursifs peuvent faire décroitre l'un ou l'autre des deux arguments. Nous ne pouvons plus nous ramener à une induction sur un des deux éléments. En revanche, nous pouvons revenir à la raison première qui nous a permis de définir des fonctions inductivement : une récursion bien-fondée, en considérant un ordre $\leq$ sur les paires de listes : $(l_1, l_2) \leq (l'_1, l'_2)$ si $l_1 \preceq l'_1$ et $l_2 \preceq l'_2$. La preuve de la Proposition 5.5 s'adapte alors en utilisant cet ordre-ci.

fusion(l_1, l_2)
Entrée: deux listes d'entiers l_1 et l_2 croissantes ;
Sortie: une liste croissante l, union de l_1 et l_2.

```
si l1 = [] alors
    renvoyer l2
si l2 = [] alors
    renvoyer l1
soit h1 :: t1 = l1 et h2 :: t2 = l2
si h1 ≤ h2 alors
    renvoyer h1 :: fusion(t1, l2)
renvoyer h2 :: fusion(l1, t2)
```

Algorithme 5.5 – Un algorithme doublement inductif pour la fusion de deux listes triées.

Dans cette exemple de la fusion, les deux arguments décroissent au fur et à mesure des appels récursifs. Il est aussi possible d'avoir des doubles inductions plus compliquées, où l'un des arguments peut augmenter. Un exemple typique est la fonction d'Ackermann sur les entiers naturels :

$$\frac{}{\text{ack}(0, n) : S(n)} \qquad \frac{\text{ack}(m, S(0)) : r}{\text{ack}(S(m), 0) : r} \qquad \frac{\text{ack}(S(m), n) : r \qquad \text{ack}(m, r) : q}{\text{ack}(S(m), S(n)) : q}$$

Le premier argument ne croit jamais, mais le deuxième argument peut augmenter lorsque le premier argument décroit : l'argument r dans la deuxième prémisse de la troisième règle est en général plus grand que n. Nous pouvons néanmoins nous en sortir, en définissant ici aussi un ordre bien-fondé adapté à la situation. Nous posons $(n, m) \leq (n', m')$ si $n \prec n'$ ou $(n = n'$ et $m \preceq m')$. Nous vérifions simplement que les prémisses sont toujours inférieures à la conclusion pour l'ordre $\leq$, justifiant de nouveau la bonne définition de cette fonction.

Les inductions doubles sont heureusement assez peu courantes. Nous en croiserons exceptionnellement, qui s'analyseront toutes en combinant les ordres bien-fondés de chaque argument en un ordre bien-fondé pour la paire d'arguments, tel que les appels récursifs respectent cet ordre bien-fondé.

5.4 Preuves par induction

Les ensembles définis inductivement de façon non-ambiguë, nous l'avons vu, induisent un ordre de dérivation bien-fondé. Cette ordre nous permet d'étendre le principe de récurrence que nous connaissons sur les entiers naturels, à tout ensemble défini inductivement, en un principe d'induction. Ce principe d'induction est un outil de base pour démontrer des propriétés sur les valeurs définies inductivement, et sur les fonctions définies inductivement.

Rappelons le principe de récurrence. Soit P une proposition sur les entiers, telle que

(i) $P(0)$ est vraie ;

(ii) pour tout entier n, si $P(n)$ est vraie, alors $P(n+1)$ est aussi vraie.

Bien sûr, le *principe de récurrence* affirme que $P(n)$ est vrai pour tout entier n. Mais oublions cette conclusion temporairement, et considérons l'ensemble X des entiers n tels que $P(n)$ est vraie : $X = \{n \in \mathbb{N} \ : \ P(n)\}$. Certainement nous pouvons définir cet ensemble par spécification. Remarquons maintenant que :

(1) $0 \in X$, par (i)

(2) pour tout $n \in X$, $S(n) \in X$, par (ii) (si $n \in X$, par définition de X, $P(n)$ est vraie, donc par (ii) $P(n+1)$ est vraie, donc par définition de X, $n+1 \in X$).

Souvenons-nous maintenant que, par le Théorème 3.5 du point fixe, $\mathbb{N}$ est défini inductivement comme le plus petit ensemble tel que (1) et (2) sont vrais, donc X est au moins aussi grand que $\mathbb{N}$, c'est-à-dire $\mathbb{N} \subseteq X$. Comme par définition de X, $X \subseteq \mathbb{N}$, nous avons que $X = \mathbb{N}$. Autrement dit, $P(n)$ est vraie pour tout entier $n \in \mathbb{N}$, nous avons démontré le principe de récurrence ![9]

Le cas des entiers naturels n'est pas spécial : tout ensemble inductif admet un équivalent du principe de récurrence, que nous appelons le principe d'induction.

Théorème 5.7 (Principe d'induction)**.** Soit X un ensemble défini inductivement par (B, R), et P une proposition sur X. Si

- ▷ (Base) pour tout $x \in B$, $P(x)$ est vraie ;
- ▷ (Induction) pour toute règle $\mathcal{R} \in R$ et $(x_1, x_2 \ldots, x_k) \in \mathcal{R}$, si $P(x_1)$, $P(x_2)$,..., $P(x_{k-1})$ sont vraies, alors $P(x_k)$ est vraie ;

alors pour tout $x \in X$, $P(x)$ est vraie.

Démonstration. Nous définissons l'ensemble $Z := \{z \in X : P(z)\}$. Notons que

- ▷ pour tout $b \in B$, $P(b)$ est vraie par l'affirmation (Base), donc $b \in Z$ par définition de Z ;
- ▷ pour tout $(x_1, x_2, \ldots, x_k) \in \mathcal{R} \in R$, si $x_1, x_2, \ldots, x_{k-1} \in Z$, alors $P(x_1)$, $P(x_2)$,..., $P(x_{k-1})$ sont vraies par définition de Z, donc par (Induction) $P(x_k)$ est vrai, et $x_k \in Z$.

Par le Théorème 3.5 du point fixe, X est un sous-ensemble de Z, qui par définition est un sous-ensemble de X. Donc $X = Z$, prouvant que pour tout élément $x \in X$, $P(x)$ est vraie. □

Application : preuve sur un ensemble défini inductivement. Considérons cette définition d'un ensemble X de mots sur l'alphabet $\{a, b\}$.

$$\frac{}{\varepsilon} \qquad \frac{}{\mathrm{a}} \qquad \frac{}{\mathrm{b}} \qquad \frac{w}{\mathrm{ab}.w}\,\mathrm{ab} \qquad \frac{w}{\mathrm{ba}.w}\,\mathrm{ba}$$

Montrons d'abord que pour tout mot $w \in X$, le nombre $|w|_\mathrm{a}$ de a diffère du nombre $|w|_\mathrm{b}$ de b d'au plus 1 :

$$\phi(w) := |w|_\mathrm{a} - |w|_\mathrm{b} \in \{-1, 0, 1\}.$$

Notre proposition est donc : $P(w) := (\phi(w) \in \{-1, 0, 1\})$. Nous prouvons P sur X par induction, en commençant par montrer que P est vraie pour les cas de bases : $\phi(\varepsilon) = 0$ et $\phi(\mathrm{a}) = \phi(\mathrm{b}) = 1$.

Nous considérons ensuite pour chaque règle d'induction. Soit $w \in X$ tel que $P(w)$ est vrai. la règle ab permet de produire $\mathrm{ab}.w$ depuis la prémisse w. Nous avons alors

$$\phi(\mathrm{ab}.w) = |\mathrm{ab}.w|_\mathrm{a} - |\mathrm{ab}.w|_\mathrm{b} = |w|_\mathrm{a} + 1 - |w|_\mathrm{b} - 1 = |w|_\mathrm{a} - |w|_\mathrm{b} = \phi(w) \in \{-1, 0, 1\}.$$

Ainsi $P(\mathrm{ab}.w)$ est vraie. Le raisonnement est symétrique pour la règle ba, $P(\mathrm{ba}.w)$ est aussi vraie.

Par principe d'induction, nous en concluons que pour tout $w \in X$, $\phi(w) \leq 1$.

Comme deuxième exemple, démontrons qu'effectivement les mots de X n'ont pas trois a ou trois b consécutifs. Cette fois, nous n'arriverons pas à le montrer directement par le principe d'induction. Nous sommes dans un de ces cas qui nécessitent d'établir une hypothèse d'induction plus forte. Nous utiliserons la proposition suivante Q sur les mots $w \in X$: $Q(w)$ est vraie lorsque ni $\mathrm{a}.w$, ni $\mathrm{b}.w$ ne contiennent trois lettres consécutives identiques. Clairement si $Q(w)$ est vraie, alors w ne contient pas trois lettres consécutives identiques. Démontrons que Q est vraie sur X par induction.

9. Notre raisonnement est circulaire, puisque nous utilisons le théorème du point fixe implicitement dans la définition de $\mathbb{N}$, que nous avons prouvé en utilisant le principe de récurrence. Cependant, comme nous l'avions mentionné, le théorème du point fixe peut se prouver sans utiliser le principe de récurrence, en considérant l'intersection de tous les ensembles X clos possibles. Ainsi nous évitons le cycle dans la preuve.

Pour les cas de bases, puisqu'ils contiennent au plus une lettre, en leur ajoutant a ou b devant, nous obtenons des mots de longueur au plus 2, ne pouvant donc pas contenir trois lettres consécutives (*a fortiori* identiques). Donc $P(\varepsilon)$, $P(\text{a})$ et $P(\text{b})$ sont vraies.

Passons au cas d'induction, et d'abord la règle ab. Soit donc $w \in X$ tel que $Q(w)$ est vraie. Notons $w_1 w_2 \dots w_k$ les lettres de w, avec $k \geq 0$. Les $k+1$ triplets de lettres consécutives du mot aab.w sont donc aab, ab.w_1, b.$w_1 w_2$, $w_1 w_2 w_3$ jusqu'à $w_{k-2} w_{k-1} w_k$. Puisque $Q(w)$ est vrai, b.w ne contient par de triplets de lettres consécutives identiques, donc les $k-1$ derniers triplets de notre liste ne sont pas des triplets de lettres identiques. Par ailleurs, aab et ab.w_1 ne sont pas des triplets de lettres identiques non plus. Ainsi aab.w ne contient pas de triplets de lettres consécutives identiques. Puisque bab.w possèdent les mêmes triplets de lettres consécutifs excepté le premier bab, bab.w non plus ne possède pas de triplets de lettres consécutives identiques. Donc $Q(\text{ab}.w)$ est vraie.

La règle ba se démontre de la même façon, par symétrie du rôle de a et de b, impliquant que $Q(\text{ba}.w)$ est vraie.

Par principe d'induction, pour tout $w \in X$, $Q(w)$ est vraie.

Application : preuves sur une fonction définie inductivement. Nous pouvons aussi utiliser le principe d'induction pour démontrer des propriétés relatives à une fonction définie inductivement. À titre d'exemple, reprenons la définition de la factorielle.

$$\frac{}{0! : 1} \qquad \frac{n! : r}{(n+1)! : (n+1) \times r}$$

Démontrons la proposition suivante : pour tout $n \in \mathbb{N}$, $n > 1 \implies n!$ est pair. Pour cela posons la proposition $P(n) := (n > 1 \implies n!$ est pair), qui porte sur l'entier n mais aussi sur l'image de n par la fonction définie inductivement.

Pour le cas de base, $0 > 1 \implies 0!$ est pair est vrai, puisque $0 > 1$ est faux.

Considérons le cas d'induction, soit $n \in \mathbb{N}$ et supposons que $P(n)$ est vraie. Il y a trois cas à traiter.

Cas A. Si $n = 0$, alors $n+1 = 1$, et donc $n+1 > 1$ est faux, donc $n+1 > 1 \implies (n+1)!$ est pair, c'est-à-dire $P(n+1)$, est vrai.

Cas B. Si $n = 1$, alors $(n+1)! = (n+1) \times n!$ par la règle d'induction dans la définition de la factorielle. En substituant, $2! = 2 \times 1!$ est pair car multiple de 2. Subséquemment, $P(n+1)$ est vraie puisque la conclusion de l'implication l'est.

Cas C. Si $n > 1$, par $P(n)$ nous avons que $n!$ est pair. Dans ce cas, par la règle d'induction, $(n+1)! = (n+1) \times n!$, et cette quantité est paire aussi puisque multiple d'un nombre pair. De nouveau la conclusion de l'implication dans $P(n+1)$ est vraie, donc $P(n+1)$ est vraie.

Dans chaque cas, $P(n+1)$ est vraie, par principe d'induction nous en concluons que P est vraie sur $\mathbb{N}$.

Application : substitution dans une formule. Voici un deuxième exemple de preuve sur la définition inductive d'une fonction, permettant de justifier une opération de substitution comme nous en recroiserons plus tard (et notablement en logique).

Partant d'une expression arithmétique comme $(x+3) \times x$, nous pouvons l'évaluer pour des valeurs de x et y donnée. De fait, en notant $e : \mathcal{I} \to \mathbb{Z}$ la fonction donnant la valeur de chaque variable ($\mathcal{I}$ représentant l'ensemble des identifiants de variables), nous pouvons donner une définition inductive de la valeur d'une expression arithmétique.

$$\frac{}{f_e(n) : n}\ n \in \mathbb{Z} \qquad \frac{}{f_e(x) : e(x)}\ x \in \mathcal{I}$$

$$\frac{f_e(\tau) : n \qquad f_e(\sigma) : p}{f_e(\tau + \sigma) : n + p} \qquad \frac{f_e(\tau) : n \qquad f_e(\sigma) : p}{f_e(\tau \times \sigma) : n \times p}$$

Chaque variable est évaluée selon la valeur donnée par e, et les opérateurs s'appliquent naturellement. Une substitution consiste à remplacer toutes les occurences d'une variable, par exemple x, par une autre expression arithmétique. Par exemple si dans $(x+3) \times x$, nous substituons $(x+y+2)$

à x, nous obtenons l'expression $((x+3)\times x)[x+y+2 \leftarrow x] = ((x+y+2)+3)\times(x+y+2)$. Nous pouvons définir une substitution inductivement sur la formule à laquelle elle est appliquée. Ainsi pour une variable $x \in \mathcal{I}$ et une formule σ, nous définissons

$$\frac{}{n[\sigma \leftarrow x] : n}\ n \in \mathbb{N} \qquad \frac{}{x[\sigma \leftarrow x] : \sigma} \qquad \frac{}{y[\sigma \leftarrow x] : y}\ y \in \mathcal{I} \setminus \{x\}$$

$$\frac{\rho[\sigma \leftarrow x] : \rho' \qquad \tau[\sigma \leftarrow x] : \tau'}{(\rho+\tau)[\sigma \leftarrow x] : \rho' + \tau'}$$

Si nous voulons évaluer cette expression $((x+y+2)+3)\times(x+y+2)$, pour $x=3$ et $y=-1$, nous pouvons bien sûr calculer $((3+(-1)+2)+3)\times(3+(-1)+2)$, c'est-à-dire calculer directement $f_e(\tau[\sigma \leftarrow x]) = f_e(((x+3)\times x)[x+y+2 \leftarrow x])$, avec $e(x)=3$ et $e(y)=-1$. Mais nous pouvons aussi d'abord évaluer $f_e(\sigma) = f_e(x+y+2)$, ici 4, puis évaluer $f_{e'}(\tau) = f_{e'}((x+3)\times x)$ avec $e'(x) = f_e(\sigma) = 4$ et $e'(y) = f_e(y) = -1$. Les résultats seront identiques, parce que

$$f_e(\tau[\sigma \leftarrow x]) = f_{e'}(\tau),$$

où e' est défini par $e'(x) = f_e(\sigma)$ et $e'(z) = e(x)$ pour tout $z \in \mathcal{I} \setminus \{x\}$. Cette égalité demande néanmoins une démonstration, que nous pouvons détailler par une induction sur τ.

▷ Si $\tau = n$ pour un entier $n \in \mathcal{N}$, alors

$$f_e(n[\sigma \leftarrow x]) = f_e(n) = n = f_{e'}(n).$$

▷ Si $\tau = x$, alors

$$f_e(x[\sigma \leftarrow x]) = f_e(\sigma) = e'(x) = f_{e'}(x),$$

la première égalité étant par définition de la substitution, la deuxième par définition de e', et la troisième par définition de l'évaluation.

▷ Si $\tau = z$ pour une variable $z \in \mathcal{I} \setminus \{x\}$, alors

$$f_e(z[\sigma \leftarrow x]) = f_e(z) = e(z) = e'(z) = f_{e'}(z),$$

la première égalité étant par définition de la substitution, les deuxième et quatrième par définition de l'évaluation, la troisième par définition de e'.

▷ Si $\tau = \rho + \rho'$, alors

$$\begin{aligned}
f_e((\rho+\rho')[\sigma \leftarrow x]) &= f_e(\rho[\sigma \leftarrow x] + \rho'[\sigma \leftarrow x]) && \text{(par définition de la subtitution)}\\
&= f_e(\rho[\sigma \leftarrow x]) + f_e(\rho'[\sigma \leftarrow x]) && \text{(par définition de l'évaluation)}\\
&= f_{e'}(\rho) + f_{e'}(\rho') && \text{(par hypothèse d'induction appliquée à } \rho \text{ et à } \rho')\\
&= f_{e'}(\rho+\rho') && \text{(par définition de l'évaluation)}
\end{aligned}$$

ce qui prouve ce cas, et le dernier cas, si $\tau = \rho \times \rho'$, est similaire.

Ainsi nous avons bien démontré que pour toute formule τ,

$$f_e(\tau[\sigma \leftarrow x]) = f_{e'}(\tau).$$

Conséquence. Une preuve utilisant un principe d'induction se rédige toujours ainsi : présentation de la proposition P à démontrer, démonstration des cas de base, démonstration de chaque règle d'induction et conclusion affirmant que P est toujours vraie. Pour démontrer une règle d'induction, d'abord supposer que la propriété est vraie pour les prémisses, puis en déduire la propriété pour la conclusion.

La démonstration d'une propriété d'une fonction définie inductivement, par exemple sa correction, s'établit en utilisant une propriété $P(x,v)$ liant l'élément x de l'ensemble défini inductivement à son image v par la fonction définie inductivement.

À retenir. À toute définition inductive correspond un principe d'induction, pouvant être utilisé pour démontrer des propriétés de l'ensemble défini inductivement, ou des fonctions définies inductivement.

La correction d'un algorithme inductif peut souvent être démontrée en utilisant le principe d'induction correspondant. Ainsi, l'induction forme un tout cohérent, permettant de modéliser des données, d'écrire des algorithmes sur ces données, et de démontrer tant la terminaison que la correction de ces algorithmes. Nous avons donc tout intérêt à nous appuyer sur cette technique lorsqu'elle est applicable.

Exercice 5.5. Montrer par une preuve inductive que la fonction définie dans l'Exercice 5.3 est correcte : elle calcule bien l'entier représenté en écriture binaire.

Exercice 5.6. Définir et démontrer inductivement la correction d'une fonction permettant d'incrémenter un entier représenté en binaire, défini dans l'Exercice 3.2. La fonction prend donc un entier représenté en binaire et retourne un entier représenté en binaire. Par exemple, l'image de 100101 est 100110.

6 Application : une introduction à la sémantique

Nous disposons maintenant des outils pour donner une définition formelle, non seulement des algorithmes, mais aussi de la sémantique des algorithmes. Ainsi, nous allons pouvoir manipuler les algorithmes mathématiquement afin de démontrer leurs propriétés, et en premier lieu les propriétés de correction. Nous commençons par donner une définition d'un langage minimaliste, esquissé en Section 1, et de sa sémantique. Définir un langage complet nécessiterait un effort supérieur à ce que nous pouvons nous permettre dans le cadre de ce cours. Cependant les techniques que nous allons introduire se généralise assez naturellement pour définir les langages complexes auxquels nous sommes habitués, et suffiront pour illustrer les méthodes permettant de raisonner formellement sur nos algorithmes.

Le langage de programmation que nous allons adopter pour cette section est celui déjà décrit en Section 1 : il se base sur l'instruction d'affectation, dispose du seul type des entiers, et de la boucle **tant que** comme unique structure de contrôle. Il n'autorise pas les appels de fonctions, et pour simplifier encore plus nous n'aurons pas besoin de déclarer les variables, toutes variables ayant par défaut la valeur 0. Ce petit langage est néanmoins *Turing-complet*, ce qui signifie qu'il a les capacités de résoudre les mêmes problèmes algorithmiques qu'une machine de Turing, tout comme les langages de programmation auxquels nous sommes habitués.

Identifiants. Notre langage permet de manipuler des entiers, et de les stocker dans des variables. Les variables seront représentés par des identifiants. Nous noterons $\mathcal{I}$ l'ensemble des identifiants, qui est défini comme Σ^* pour Σ l'alphabet latin. Autrement dit les identifiants sont des mots en caractères latins.

Expressions. Les entiers seront représentés par leur littéraux, qui sont des séquences de chiffres décimaux bien-formées, éventuellement précédées d'un signe + ou − (0019 n'est pas bien formées). Par un abus de notation, nous nous permettrons de noter l'ensemble des littéraux entiers $\mathbb{Z}$.

Puisque nous voudrons certainement réaliser des opérations arithmétiques sur les entiers, nous permettons d'écrire des expressions arithmétiques. Pour rester simple, nous n'autorisons que les 4 principaux opérateurs arithmétiques. Nous pouvons donc définir inductivement l'ensemble des expressions arithmétiques $\mathcal{E}$ dénotant des entiers.

$$\frac{}{n}\ n \in \mathbb{Z} \qquad \frac{}{id}\ id \in \mathcal{I} \qquad \frac{\sigma \qquad \tau}{(\sigma \oplus \tau)}\ \oplus \in \{+, -, \times, /\}$$

Nous pourrons omettre les parenthèses, lorsque les règles usuelles de l'arithmétique l'autorisent.

Sémantique dénotationnelle des expressions Nous rappelons la définition de l'état d'exécution d'un algorithme, qui permet de représenter la mémoire à une étape de l'exécution de l'algorithme. Un état d'exécution est une fonction partielle des identifiants de variable vers les valeurs que prennent ces variables, c'est-à-dire de $\mathcal{I}$ dans $\mathcal{Z}$. La valeur d'une expression contenant des variables dépend de l'état d'exécution de l'algorithme au moment d'évaluer l'expression. Si une des variables de l'expression n'est pas dans le domaine de l'état d'exécution, nous considérerons que sa valeur est 0.

En notant e l'état d'exécution de l'algorithme, nous pouvons définir inductivement la sémantique d'une expression τ, c'est-à-dire sa valeur par rapport à l'environnement e, que nous notons $[\![\tau]\!]_e$:

$$\frac{}{[\![n]\!]_e : n}\ n \in \mathbb{Z} \qquad \frac{}{[\![id]\!]_e : e(id)}\ id \in \mathcal{I}, id \in \mathrm{dom}(e) \qquad \frac{}{[\![id]\!]_e : 0}\ id \in \mathcal{I}, id \notin \mathrm{dom}(e)$$

$$\frac{[\![\tau]\!]_e : n_1 \qquad [\![\sigma]\!]_e : n_2}{[\![\tau \oplus \sigma]\!]_e : n_1 \oplus n_2}\ \oplus \in \{+, -, \times, /\}$$

Ici, la division est interprétée comme la division entière, les autres opérateurs arithmétiques ont leur signification usuelle. La sémantique d'une expression est donc la valeur naturellement calculée

à partir de cette expression. Toujours pour rester au plus simple, nous ignorerons la possibilité d'une division par 0 dans cette présentation [10]. Nous noterons $\mathcal{E}$ l'ensemble des expressions.

Exercice 6.1. Calculer :

1. $[\![(n+3)\times m]\!]_{[n\to 3, m\to 5]}$,
2. $[\![(n+m)\times(n-m)]\!]_{[n\to 2, m\to -3]}$,
3. $[\![((n/2)-(m/5))/3]\!]_{[m\to 25, n\to 16]}$,
4. $[\![(m-n)+(m+n)]\!]_{[n\to 8, m\to 1]}$.

Programmes. Nous définissons maintenant les programmes, qui sont essentiellement des listes d'instructions. Ou du moins, ils seraient des listes s'il n'y avait de structures de contrôle, qui créent une hiérarchie. Ce sont donc plutôt des arbres, que nous définissons inductivement. Notre langage de programmation minimaliste contient comme instructions la déclaration de variable et l'affectation. Il dispose de deux structures de contrôle,

- ▷ la séquence, que nous noterons avec le point-virgule ou un retour à la ligne, et qui permet d'enchaîner deux instructions ou plus généralement deux programmes ;
- ▷ et la boucle **tant que**, avec une égalité entre deux entiers comme condition d'arrêt.

Rappelons-nous que $\mathcal{I}$ est l'ensemble des identifiants de variables, et $\mathcal{E}$ l'ensemble des expressions arithmétiques pouvant contenir des variables. La définition inductive de l'ensemble des programmes est

$$\frac{}{id \leftarrow \phi}\ id\in\mathcal{I}, \phi\in\mathcal{E} \qquad \frac{p_1 \quad p_2}{p_1 ; p_2}$$

$$\frac{p}{\textbf{Tant que } \sigma \neq \tau \textbf{ faire } p}\ \sigma,\tau\in\mathcal{E}$$

Sémantique. Nous devons maintenant définir la sémantique du programme, c'est-à-dire quel est son sens, qu'est-ce qu'il calcule. Parmi les multiples façons de définir la sémantique d'un langage de programmation, nous choisissons de donner une sémantique opérationnelle, c'est-à-dire décrivant la façon dont le programme s'exécute, par ses effets sur son environnement. Notre réutilisons donc la notion d'état d'exécution du programme, que nous avons introduite en Section 2.4. Rappelons donc que l'état d'exécution d'un programme est donné par une fonction partielle des identifiants $\mathcal{I}$ vers les valeurs, qui pour notre langage sont des entiers ($\mathbb{Z}$). L'effet d'un programme p est alors de modifier un état d'exécution e en un état d'exécution e', ce que nous noterons ainsi :

$$\langle e\rangle p \langle e'\rangle$$

Pour décrire les changements d'états, pour tout entier n et tout identifiant id, nous noterons $e[n \leftarrow id]$ l'application partielle avec

$$e[n\leftarrow id](x) = \begin{cases} n & \text{si } x = id, \\ e(x) & \text{si } x \in \mathrm{Dom}(e)\setminus\{id\}, \end{cases}$$

et $e[n \leftarrow id](x)$ est indéfinie sinon.

Nous pouvons alors proposer une définition de la sémantique, en tant que fonction définie inductivement sur les programmes. Pour l'affectation, l'expression est évaluée dans l'état d'entrée, et l'état de sortie correspond à l'ajout ou la modification de l'image de l'identifiant pour devenir la valeur ainsi calculée.

$$\frac{}{\langle e\rangle id \leftarrow \tau \langle e[[\![\tau]\!]_e \leftarrow id]\rangle}\ id\in\mathrm{Dom}(e), \tau\in\mathcal{E}$$

La règle pour la séquence est immédiate.

10. Nous pourrions introduire un état d'erreur : si une division se produit, l'état d'exécution devient l'état d'erreur. L'état d'erreur se propagerait par les règles de la sémantique (qu'il faudrait donc dédoubler).

$$\frac{\langle e\rangle p_1\langle e'\rangle \qquad \langle e'\rangle p_2\langle e''\rangle}{\langle e\rangle p_1; p_2\langle e''\rangle}$$

Pour la boucle **Tant que**, nous distinguons deux cas, selon que la condition s'évalue en vrai ou en faux. Si la boucle continue, il faut évaluer le bloc d'instruction puis revenir au début de la boucle. C'est ce que décrivent les règles suivantes.

$$\frac{}{\langle e\rangle \textbf{Tant que } \sigma \neq \tau \textbf{ faire } p\langle e\rangle} \; \sigma, \tau \in \mathcal{E}, [\![\sigma]\!]_e = [\![\tau]\!]_e$$

$$\frac{\langle e\rangle p\langle e'\rangle \qquad \langle e'\rangle \textbf{Tant que } \sigma \neq \tau \textbf{ faire } p\langle e''\rangle}{\langle e\rangle \textbf{Tant que } \sigma \neq \tau \textbf{ faire } p\langle e''\rangle} \; \sigma, \tau \in \mathcal{E}, [\![\sigma]\!]_e \neq [\![\tau]\!]_e$$

Bien sûr, nous remarquons que la deuxième règle n'est pas une règle d'induction valide, puisque l'une des prémisses (celle de droite) est identique à la conclusion de la règle. En fait, cela ne devrait pas nous étonner, nous savons d'expérience qu'une boucle peut ne pas terminer, et il est normal de ne pas pouvoir donner à une boucle infinie une sémantique en terme de changement d'état d'exécution.

Nous nous en sortons néanmoins lorsque la boucle possède un variant. Dans ce cas, la paire (p, e), avec p le programme et e l'état d'exécution à l'entrée du programme, est lexicographiquement plus petite dans les prémisses que dans la conclusion de chaque règle. Ainsi, la sémantique du programme est bien définie lorsque les boucles ont des variants.

Algorithmes. Pour la suite, nous supposerons disposer d'un langage de programmation ayant un peu plus de fonctionnalités, instructions ou structures de contrôle, types, structures de donnée comme les tableaux. Notre sémantique pourrait être étendue sans difficulté pour supporter les fonctionnalités auxquelles nous sommes habitués.

Un algorithme, nous l'avons vu en Section 1, est la donnée d'un programme d'une pré-condition et d'une post-condition. Les pré-conditions et post-conditions sont des formules logiques portant sur l'état d'exécution initial (pour la pré-condition), et sur les deux états d'exécution initial et final du programme (pour la post-condition), respectivement. La pré-condition limite le domaine d'application de l'algorithme. La post-condition exprime une relation entre l'état d'exécution finale et l'état d'exécution initial, typiquement pour caractériser la sortie en fonction de l'entrée.

Notre sémantique permet de calculer l'état final à partir de l'état initial [11]. Mais si notre but est de certifier que notre algorithme est correct, cette sémantique opérationnelle ne suffit pas : nous pouvons démontrer que le résultat est correct pour une entrée en exécutant l'algorithme, comme lorsque nous écrivons un test en programmation, mais nous ne pouvons pas encore montrer que le résultat est correct pour toutes les entrées possibles. Pour cela, il nous faut un outil permettant, non pas de déduire l'état final d'exécution depuis l'état initial, mais plutôt de déduire la post-condition depuis la pré-condition de l'algorithme. C'est l'objectif du prochain chapitre.

Exercice 6.2. Évaluer, en dessinant l'arbre de dérivation, l'état final obtenu pour le programme suivant, à partir de l'état d'exécution e de domaine $\{n, m\}$, avec $e(n) = 3$ et $e(m) = 5$.

1: $s \leftarrow 0$
2: **tant que** $n \neq 0$ **faire**
3: $\quad n \leftarrow n - 1$
4: $\quad s \leftarrow s + m$

Exercice 6.3. Ajoutons le **Si** ... **alors** dans notre langage. Quelles seraient les deux règles permettant de définir sa sémantique, selon la valeur prise par la condition testée ?

11. Essentiellement, nous venons d'écrire un interpréteur pour notre langage de programmation.

7 Preuves de correction d'algorithmes

Avec l'exemple de l'algorithme du Zune étudié en Section 1, nous avons remarqué l'importance de se munir d'outils pouvant établir que nos algorithmes sont corrects. La Section 6 nous a permis de définir une sémantique précise pour (un extrait de) notre langage de programmation, qui nous autorise à raisonner formellement sur l'effet d'un programme. Dans cette section, nous introduisons plus formellement la notion de correction d'un algorithme, et proposons une technique générale de preuve de correction : la logique de Hoare.

7.1 Correction d'un algorithme : définition

Comme nous l'avons vu précédemment, un algorithme se définit par une pré-condition, une post-condition, en plus du bloc d'instructions décrivant le fonctionnement de l'algorithme. Ce que nous illustrons ainsi :

Entrée: pré-condition sur les entrées.
Sortie: post-condition sur les sorties.
1:
2: bloc d'instructions de l'algorithme
3:

La pré-condition indique la nature des entrées avec, lorsque cela est nécessaire des restrictions sur les valeurs d'entrées. Bien qu'en pratique, nous utilisons un langage naturel pour décrire la pré-condition, il faudrait pour être parfaitement formel, l'écrire comme une formule logique sur l'état d'exécution initial.

La post-condition indique quelle sont les données calculées et renvoyées par l'algorithme (les *sorties*), et les propriétés que ces données satisfont, notamment vis-à-vis des entrées. La post-condition spécifie donc le comportement attendu d'un appel à l'algorithme, indépendamment de son fonctionnement interne. La post-condition aussi doit pouvoir s'écrire formellement comme une formule logique, cette fois portant sur les deux états d'exécution, initial et final.

Ainsi, reprenant l'exemple de l'Algorithme 2.3 de la division entière, pour lequel la pré-condition est $n \in \mathbb{N}$ et $m \in \mathbb{N} \setminus \{0\}$, et la post-condition est : la valeur de sortie est le résultat de la division entière de n par m, ce qui pourrait être écrit plus formellement :

$$q_\star \in \mathbb{N} \wedge \text{il existe } r \in \{0, 1, \ldots, m_0 - 1\} \text{ tel que } n_0 = q_\star \times m_0 + r,$$

où les variables indicées par 0 représentent leurs valeurs dans l'état d'exécution initial, et celles indicées par $\star$ représentent leurs valeurs dans l'état d'exécution final (convention que nous réutiliserons régulièrement). Notons que puisque l'algorithme est une division de n_0 par m_0, il est nécessaire que m_0 soit non-nul, contrainte qui figure bien dans la pré-condition.

Dire que cet algorithme est correct, c'est essentiellement affirmer que pour toutes entrées qui satisfont la pré-condition, la sortie de l'algorithme satisfait la post-condition.

Définition 7.1. On dit qu'un algorithme est *partiellement correct*, lorsque pour toute exécution à partir d'entrées satisfaisant les pré-conditions :

si l'algorithme termine, alors les sorties vérifient les post-conditions.

Définition 7.2. On dit qu'un algorithme est *correct*, si il est partiellement correct et si toute exécution à partir d'entrées satisfaisant les pré-conditions termine.

Continuons l'exemple de l'Algorithme 2.3 de division naturelle. Nous avons déjà prouvé qu'il termine en exhibant un variant pour la boucle **tant que**. Ainsi, si nous prouvons la correction partielle, nous obtiendrons que l'algorithme est correct. Nous établirons bientôt cette preuve. D'un

autre côté, l'algorithme n'aurait pas été correct, si nous avions choisi comme pré-condition : n est un entier relatif et m est un entier relatif non-nul. En autorisant les valeurs négatives, plusieurs problèmes apparaissent, que nous illustrons par deux exemples d'entrées.

- ▷ Si $n = -6$ et $m = -3$, $n < m$ dès l'entrée dans la boucle, donc le corps n'est donc jamais évalué. Ainsi *quotient* reste nul, et la valeur de sortie est 0, qui n'est pas la réponse attendue (2).
- ▷ si $n = 6$ et $m = -3$, la condition de la boucle **tant que** est initialement satisfaite, et chaque itération augmente n de 3, sans changer m. Ainsi, la condition reste toujours vraie, et l'algorithme ne termine pas.

Exercice 7.1. Modifier l'algorithme de division entière par soustractions successives pour qu'il soit correct pour toutes entrées dans $\mathbb{Z}$.

7.2 Une logique minimale (lecture facultative).

Dans le cadre de notre mini-langage de programmation, nous n'aurons besoin que d'une fraction des constructions fréquemment utilisées en logique. Nous réutilisons l'ensemble $\mathcal{E}$ des expressions arithmétiques, définies en Section 6. Nous pouvons alors inductivement définir les formules logiques en se restreignant volontairement à quelques constructions seulement, afin de rester bref. Notamment nous utiliserons l'égalité comme seule relation.

$$\frac{}{\sigma = \tau}\ \sigma, \tau \in \mathcal{E} \qquad \frac{\phi}{\neg\phi} \qquad \frac{\phi \qquad \psi}{\phi \vee \psi} \qquad \frac{\phi \qquad \psi}{\phi \wedge \psi}$$

$$\frac{\phi}{\exists id.\phi}\ id \in \mathcal{I} \qquad \frac{\phi}{\forall id.\phi}\ id \in \mathcal{I}$$

Pour capturer les propriétés de l'affectation, nous définissons une notion de substitution dans les formules logiques. Précisément, nous souhaitons pouvoir substituer une expression arbitraire à la place d'une variable, comme par exemple remplacer x par $3 + y$. Dans un premier temps, nous définissons inductivement la substitution dans les expressions :

$$\frac{}{id[\sigma \leftarrow id] : \sigma} \qquad \frac{}{x[\sigma \leftarrow id] : x}\ x \in \mathcal{I} \setminus \{id\} \qquad \frac{}{n[\sigma \leftarrow id] : n}\ n \in \mathbb{N}$$

$$\frac{\tau_1[\sigma \leftarrow id] : \tau_1' \qquad \tau_2[\sigma \leftarrow id] : \tau_2'}{(\tau_1 \oplus \tau_2)[\sigma \leftarrow id] : \tau_1' \oplus \tau_2'}\ \oplus \in \{+, -, \times, /\}$$

La première règle exprime que la variable substituée est remplacée par σ. La deuxième règle exprime que hormis la variable substituée, les variables sont inchangées. La troisième règle exprime que les littéraux sont inchangés. La dernière règle propage la substitution aux opérandes d'un opérateur arithmétique.

Nous pouvons alors étendre la substitution aux formules, de nouveau inductivement sur les formules :

$$\frac{\sigma[e \leftarrow id] : \sigma' \qquad \tau[e \leftarrow id] : \tau'}{(\sigma = \tau)[e \leftarrow id] : \sigma' = \tau'} \qquad \frac{\phi[e \leftarrow id] : \phi'}{(\neg\phi)[e \leftarrow id] : \neg\phi'}$$

$$\frac{\phi[e \leftarrow id] : \phi' \qquad \psi[e \leftarrow id] : \psi'}{(\phi \vee \psi)[e \leftarrow id] : \phi' \vee \psi'} \qquad \frac{\phi[e \leftarrow id] : \phi' \qquad \psi[e \leftarrow id] : \psi'}{(\phi \wedge \psi)[e \leftarrow id] : \phi' \wedge \psi'}$$

$$\frac{}{(\exists id.\phi)[e \leftarrow id] : \exists id.\phi} \qquad \frac{}{(\forall id.\phi)[e \leftarrow id] : \forall id.\phi}$$

$$\frac{\phi[e \leftarrow id] : \phi'}{(\exists x.\phi)[e \leftarrow id] : \exists x.\phi'}\ x \in \mathcal{I} \setminus \{id\} \qquad \frac{\phi[e \leftarrow id] : \phi'}{(\forall x.\phi)[e \leftarrow id] : \forall x.\phi'}\ x \in \mathcal{I} \setminus \{id\}$$

Ainsi la substitution d'une variable par une expression consiste à remplacer (par l'expression donnée) toutes les occurences de la variable, sauf celles capturées par un quantificateur utilisant le même identifiant.

Enfin, nous pouvons donner une *interprétation* $\|\phi\|$ d'une formule $\|\phi\|$ dans le contexte d'un état d'exécution e, en se rappelant que e est une fonction partielle des identifiants dans les entiers. Nous adoptons la convention que 0 représente la valeur de vérité *faux*, et que 1 représente la valeur de vérité *vrai*. Ainsi, la notation $A(e)$ pour une assertion A signifie que $\|A\|_e = 1$.

$$\frac{[\![\sigma]\!]_e : n \qquad [\![\tau]\!]_e : n}{\|\sigma = \tau\|_e : 1} \qquad \frac{[\![\sigma]\!]_e : n \qquad [\![\tau]\!]_e : p}{\|\sigma = \tau\|_e : 0}\ n \neq p \qquad \frac{\|\phi\|_e : r}{\|\neg\phi\|_e : 1 - r}$$

$$\frac{\|\phi\|_e : r \qquad \|\psi\|_e : s}{\|\phi \vee \psi\|_e : \max\{|r|, |s|\}} \qquad \frac{\|\phi\|_e \qquad \|\psi\|_e}{\|\phi \wedge \psi\|_e : \min\{|r|, |s|\}}$$

$$\frac{\|\phi\|_{e[n \leftarrow id]} : r_n \quad \text{(pour tout } n \in \mathbb{Z})}{\exists id.\phi : \max\{|r_n| : n \in \mathbb{Z}\}} \qquad \frac{\|\phi\|_{e[n \leftarrow id]} : r_n \quad \text{(pour tout } n \in \mathbb{Z})}{\forall id.\phi : \min\{|r_n| : n \in \mathbb{Z}\}}$$

Les deux dernières règles ont la particularité d'utiliser un nombre infini de prémisses, ce qui n'est pas tout à fait dans le cadre que nous avons posé, mais ne pose pas de problème sauf pour une éventuelle implémentation de la fonction. La notation $e[n \leftarrow id]$ représente la fonction qui à id associe n et à toute autre identifiant x associe $e(x)$.

Nous aurons besoin du lemme suivant, qui montre que nous pouvons renommer une variable avec un nouvel identifiant sans changer l'interprètation, du moment que nous donnons au nouvel identifiant la même valeur que l'ancien.

Lemme 7.3. Pour tout état d'exécution e et pour tout identifiant x, pour toute formule ϕ telle que y n'apparait pas dans ϕ,

$$\|\phi\|_e = \|\phi[y \leftarrow x]\|_{e[e(x) \leftarrow y]}$$

Démonstration. Notre preuve commence par le cas des expressions arithmétiques. Nous prouvons donc que, pour toute expression arithmétique τ, pour tout état d'exécution e,

$$[\![\tau]\!]_e = [\![\tau[y \leftarrow x]]\!]_{e'},$$

avec $e' = e[e(x) \leftarrow y]$. La preuve est par induction sur τ :

- ▷ Si $\tau = x$ avec $x \in \mathcal{I} \setminus \{x\}$,
$$[\![x]\!]_e = e(x) = e'(x) = [\![x]\!]_{e'} = [\![x[y \leftarrow x]]\!]_{e'}$$
- ▷ Si $\tau = x$,
$$[\![x]\!]_e = e(x) = e'(y) = [\![y]\!]_{e'} = [\![x[y \leftarrow x]]\!]_{e'}$$
- ▷ si $\tau = n \in \mathbb{Z}$,
$$[\![n]\!]_e = n = [\![n]\!]_{e'} = [\![n[y \leftarrow x]]\!]_{e'}$$
- ▷ si $\tau = \sigma \oplus \rho$, avec $\oplus \in \{+, -, \times, /\}$, alors $[\![\sigma \oplus \rho]\!]_e = [\![\sigma]\!]_e \oplus [\![\rho]\!]_e$. Par hypothèse d'induction appliquée à σ et ρ, nous avons $[\![\sigma]\!]_e = [\![\sigma[y \leftarrow x]]\!]_{e'}$ et $[\![\rho]\!]_e = [\![\rho[y \leftarrow x]]\!]_{e'}$. Donc
$$[\![\sigma \oplus \rho]\!]_e = [\![\sigma[y \leftarrow x]]\!]_{e'} \oplus [\![\rho[y \leftarrow x]]\!]_{e'} = [\![\sigma[y \leftarrow x] \oplus \rho[y \leftarrow x]]\!]_{e'} = [\![(\sigma \oplus \rho)[y \leftarrow x]]\!]_{e'}$$

ce qui termine l'induction sur les expressions arithmétiques.

Nous étendons ce résultat aux formules pour prouver le lemme.

▷ si ϕ est la formule $\sigma = \tau$ pour deux expressions σ et $\tau \in \mathcal{E}$, alors

$$\begin{aligned} & \|\phi\|_e = 1 \\ \text{si et seulement si } & [\![\sigma]\!]_e = [\![\tau]\!]_e \\ \text{si et seulement si } & [\![\sigma[y \leftarrow x]]\!]_{e'} = [\![\tau[y \leftarrow x]]\!]_{e'} \\ \text{si et seulement si } & \|(\sigma = \tau)[y \leftarrow x]\|_{e'} = 1. \end{aligned}$$

▷ si $\phi = \neg\psi$,

$$\begin{aligned} \|\phi\|_e = \|\neg\psi\|_e &= 1 - \|\psi\|_e \\ &= 1 - \|\psi[y \leftarrow x]\|_{e'} && \text{(par hypothèse d'induction appliquée à } \psi) \\ &= \|\neg(\psi[y \leftarrow x])\|_{e'} \\ &= \|(\neg\psi)[y \leftarrow x]\|_{e'} = \|\phi[y \leftarrow x]\|_{e'}\,. \end{aligned}$$

▷ si $\phi = \psi_1 \vee \psi 2$,

$$\begin{aligned} \|\phi\|_e = \|\psi_1 \vee \psi_2\|_e &= \max\{\|\psi_1\|_e\,, \|\psi_2\|_e\} \\ &= \max\{\|\psi_1[y \leftarrow x]\|_{e'}\,, \|\psi_2[y \leftarrow x]\|_{e'}\} \\ &\qquad \text{(par hypothèse d'induction appliquée à } \psi_1 \text{ et } \psi_2) \\ &= \|\psi_1[y \leftarrow x] \vee \psi_2[y \leftarrow x]\|_{e'} \\ &= \|(\psi_1 \vee \psi_2)[y \leftarrow x]\|_{e'} = \|\phi[y \leftarrow x]\|_{e'}\,. \end{aligned}$$

▷ si $\phi = \psi_1 \wedge \psi_2$, le raisonnement est identique au cas précédent.

▷ si $\phi = \exists x.\psi$ avec $x \neq x$,

$$\begin{aligned} \|\phi\|_e = \|\exists x.\psi\|_e &= \max\{\|\psi\|_{e[n \leftarrow x]} : n \in \mathbb{Z}\} \\ &= \max\{\|\psi[y \leftarrow x]\|_{e[n \leftarrow x][e(x) \leftarrow y]} : n \in \mathbb{Z}\} \\ &\qquad \text{(par hypothèse d'induction appliquée à } \psi) \\ &= \max\{\|\psi[y \leftarrow x]\|_{e[e(x) \leftarrow y][n \leftarrow x]} : n \in \mathbb{Z}\} && \text{(car } y \neq x) \\ &= \|\exists x.(\psi[y \leftarrow x])\|_{e'} \\ &= \|(\exists x.\psi)[y \leftarrow x])\|_{e'} && \text{(car } x \neq x) \\ &= \|\phi[y \leftarrow x]\|_{e'} \end{aligned}$$

▷ si $\phi = \exists x.\psi$,

$$\begin{aligned} \|\phi\|_e = \|\exists x.\psi\|_e &= \|(\exists x.\psi)[y \leftarrow x]\|_e && \text{(par définition des substitutions)} \\ &= \|(\exists x.\psi)[y \leftarrow x]\|_{e[e(x) \leftarrow y]} && \text{(car } y \text{ n'apparait pas dans } \psi) \\ &= \|\phi[y \leftarrow x]\|_{e'} \end{aligned}$$

▷ si $\phi = \forall x.\psi$ ou $\phi = \forall x.\psi$, les preuves sont similaires aux deux cas précédents. □

7.3 La logique de Hoare

Pour démontrer qu'un algorithme est correct, nous allons dériver la post-condition à partir de la pré-condition et de la sémantique du programme. Ainsi le processus pourrait se décrire ainsi : partant de l'arbre de dérivation de la sémantique opérationnel pour un état initial, nous remplaçons les états, trop précis et liés à une entrée précise, par des assertions logiques, qui sont générales et pourront s'appliquer à toutes les entrées. L'étape principale devient alors de déduire l'assertion logique liée à un état d'exécution après une instruction, à partir de l'assertion logique liée à l'état d'exécution avant l'instruction.

Les assertions La logique minimale que nous avons définie peut être étendue pour inclure plus de constructions, notamment au niveau des expressions. Sans entrer dans les détails, nos assertions pourront être composées :

- ▷ des variables : celles du domaine de l'état d'exécution principalement, dénoté par leur identifiant, mais aussi celles de l'état d'exécution initial, que nous dénoterons avec l'indice 0, et les variables introduites par des quantificateurs ;
- ▷ des connecteurs logiques : $\neg$ (la négation), $\wedge$ (le *et* logique, ou *conjonction*), $\vee$ (le *ou* logique, ou *disjonction*), $\implies$ (l'implication) ;
- ▷ des quantificateurs $\forall$ (pour tout), $\exists$ (il existe), portant sur une variable. Nous nous plaçons dans la logique du premier ordre, ce qui signifie que les variables représentent des valeurs élémentaires, elles ne peuvent pas représenter un ensemble, une fonction ou une relation ;
- ▷ et des opérations et prédicats usuels sur les valeurs comme $(+, \times, <, =, \ldots)$.

Une *assertion* est une expression bien-formée écrite dans ce langage. Si A est une assertion, et e est un état d'exécution de l'algorithme, l'expression $A(e)$ signifie que A est vérifiée par l'état e.

Exemple 7.4. Pour $e = (n \mapsto 17,\ m \mapsto 3,\ \mathit{quotient} \mapsto 0)$, on a $A(e)$ pour les assertions A suivantes (entre autres) :

- ▷ $A : n \geq 0$
- ▷ $A : (n \geq m) \wedge (m \geq 0)$
- ▷ $A : (\mathit{quotient} = 0) \vee (n = 10)$
- ▷ $A : (n = n_0) \wedge (m = m_0) \wedge (n > 0) \wedge (m > 0)$

Les triplets de Hoare. Pour raisonner sur les algorithmes, nous avons besoin de décrire les propriétés d'un état et de son évolution au cours de l'exécution des instructions. Étant donné un programme p, nous allons utiliser des paires de propriétés liant les états avant et après l'exécution de p. Nous considèrerons des triplets de la forme $\{Pre\}p\{Post\}$, où Pre et $Post$ sont des assertions.

Définition 7.5. Soit Pre et $Post$ deux assertions et p un programme. Le triplet $\{Pre\}p\{Post\}$ est une *formule de Hoare* si pour tous états d'exécution e et e' tels que

(i) $\langle e\rangle p\langle e'\rangle$,

(ii) et $Pre(e)$,

alors $Post(e')$.

Voici quelques formules de Hoare valides (pour un état d'exécution initial ayant x et y dans son domaine et dont les images sont des entiers) :

1. $\{x = 4\}y = x + 1\{x = 4 \wedge y = 5\}$
2. $\{x \leq x_0\}x = x + 1\{x \leq x_0 + 1\}$
3. $\{x \leq x_0\}x = x + 1\{x \leq x_0 + 2\}$
4. $\{x \leq x_0\}y = x + 1\{x \leq x_0\}$
5. $\{y = y_0 \wedge z = y + x\}y = y + 1\{y = y_0 + 1 \wedge z = y - 1 + x\}$

Il sera pratique d'avoir une notation qui distingue les valeurs des variables dans l'état précédent et dans l'état suivant l'évaluation du programme. Ainsi pour une variable n, vis-à-vis de $\langle e\rangle p\langle e'\rangle$, nous noterons $\underline{n}$ pour $e(n)$ et $\overline{n}$ pour $e'(n)$.

Montrons ainsi que la formule 2 est vraie, en prenant e_0, e et e' vérifiant (i) et (ii). Par (ii), $\underline{x} \leq x_0$. Par la sémantique de l'affectation et (i), $e'(x) = e(x) + 1$, c'est-à-dire $\overline{x} = \underline{x} + 1$. Nous en déduisons $\overline{x} \leq x_0 + 1$, c'est-à-dire $e'(x) \leq e_0(x) + 1$, autrement dit $(x \leq x_0 + 1)(e')$.

Remarquons aussi, qu'à l'exemple des formules 2 et 3, il est possible d'avoir $\{Pre\}p\{Post_1\}$ et $\{Pre\}p\{Post_2\}$ avec $Post_1$ et $Post_2$ différentes. Dans le cas présent, l'assertion $Post_1$ implique $Post_2$, $Post_1$ est donc une assertion plus forte. C'est un exemple d'*affaiblissement*, dont il existe deux types.

Proposition 7.6. Soit p un programme.

▷ Si $\{Pre\}p\{Post_1\}$ et ($Post_1$ implique $Post_2$), alors $\{Pre\}p\{Post_2\}$.

▷ Si $\{Pre_1\}p\{Post\}$ et (Pre_2 implique Pre_1), alors $\{Pre_2\}p\{Post\}$.

Démonstration.

▷ Soient e et e' deux états d'exécution tels que $Pre(e)$ et $\langle e\rangle p\langle e'\rangle$. Puisque $\{Pre\}p\{Post_1\}$, nous déduisons de la définition des triplets de Hoare que $Post_1(e')$. Puisque $Post_1$ implique $Post_2$, $Post_2(e')$ est satisfaite. Par définition des triplets de Hoare, $\{Pre\}p\{Post_2\}$.

▷ Soient e et e' deux états d'exécution tels que $Pre_1(e)$ et $\langle e\rangle p\langle e'\rangle$. Puisque $\{Pre_1\}p\{Post\}$, par définition des triplets de Hoare, $Post(e')$. De plus, puisque Pre_2 implique Pre_1 et que $Pre_1(e)$, nous avons aussi $Pre_2(e)$ par définition de l'implication. Donc par définition des triplets de Hoare, $\{Pre_2\}p\{Post\}$. □

Ainsi il est toujours possible de renforcer l'assertion en prélude du programme, ou d'affaiblir l'assertion en sortie du programme. Désormais notre objectif est de dériver des formules de Hoare pour des programmes arbitraires, ce que nous allons faire en nous basant sur la sémantique du langage. Mais contrairement à notre sémantique, dont nous ne pouvons calculer la valeur que pour une entrée fixée, nous souhaitons dériver des assertions qui sont vraies simultanément pour toutes les exécutions. Si nous fixions les entrées, les assertions les plus fortes que nous pourrions avoir sont simplement une caractérisation précise de l'état d'exécution (de type telle variable prend telle valeur), ce qui n'est clairement pas généralisable. Nous allons donc travailler en affaiblissant les assertions, ce qui demandera de trouver le niveau d'affaiblissement adéquat, ni trop (car cela ne prouverait plus la correction), ni trop peu (car cela ne se généraliserait pas à toutes les entrées). Nous commençons par donner les règles pour l'instruction d'affectation, puis les structures de contrôle.

L'affectation. L'effet d'une affectation est de modifier la valeur contenue par une variable. Depuis une assertion Pre, pour $\langle e\rangle id \leftarrow \tau\langle e'\rangle$, nous devrions donc obtenir une égalité en plus de la forme $id = \tau$. Par ailleurs, Pre reste vrai vis-à-vis de l'ancienne valeur de id. Dans la situation suivante

$$\langle x \to 3, y \to 4\rangle x \leftarrow x + 1\langle x \to 4, y \to 4\rangle,$$

considérons l'assertion $\{x < y\}$, qui est vraie avant l'évaluation de l'affectation. Il n'est cependant pas vrai que $\{x < y \wedge x = 4\}$ après l'évaluation de l'affectation, puisque x sera égal à y. La raison est que la variable x ayant changé de valeur, une assertion ϕ dépendant de x qui pouvait être vraie avant l'affectation, ne le sera plus forcément après l'affectation. Pour cela, il faut distinguer l'ancienne valeur de la variable, de la nouvelle. Nous le faisons en introduisant une nouvelle variable $\underline{x}$, dont le rôle est de représenter l'ancienne valeur de x. Puisque l'assertion ϕ est vraie avant l'affectation, elle est vraie en renommant x en $\underline{x}$, c'est-a-dire $\phi[\underline{x} \leftarrow x]$ est satisfaite après l'affectation. Cela nous suggère la règle pour dériver la formule de Hoare d'une affectation.

Proposition 7.7. Pour tout identifiant id, toute assertion Pre et toute expression τ, tels que Pre et τ ne contiennent pas $\underline{id}$, nous avons

$$\{Pre\}id \leftarrow \tau\{\exists \underline{id}.Pre[\underline{id} \leftarrow id] \wedge id = \tau[\underline{id} \leftarrow id]\}$$

Démonstration. Intuitivement, la formule est vraie en prenant $\underline{id} = e(id)$, le membre gauche de la conjonction signifiant que Pre reste vraie avec l'ancienne valeur de id (maintenant dans $\underline{id}$), et le membre droit signifiant que la nouvelle valeur de id est donné par l'expression τ, évaluée en utilisant l'ancienne valeur de id.

La démonstration formelle est cependant assez technique, car elle nécessite de bien prendre en compte du renommage de id en $\underline{id}$. Soit $\langle e\rangle id \leftarrow \tau\langle e'\rangle$ l'exécution d'une affectation, avec

$Pre(e)$. Nous devons montrer

$$\exists \underline{id}.Pre[\underline{id} \leftarrow id] \wedge id = \tau[\underline{id} \leftarrow id](e')$$

sachant que $e' = e[[\![\tau]\!]_e \leftarrow id]$ par la régle de la sémantique de l'affectation.

Puisque $Pre(e)$ signifie $\|Pre\|_e = 1$, nous vérifions que

$$\begin{aligned}
\|Pre\|_e &= \|Pre[\underline{id} \leftarrow id]\|_{e[e(id)\leftarrow \underline{id}]} && \text{(par le Lemme 7.3)}\\
&= \|Pre[\underline{id} \leftarrow id]\|_{e[e(id)\leftarrow \underline{id}][[\![\tau]\!]_e \leftarrow id]} && \text{(car } id \text{ n'apparait pas dans } Pre[\underline{id} \leftarrow id])\\
&= \|Pre[\underline{id} \leftarrow id]\|_{e[[\![\tau]\!]_e \leftarrow id][e(id)\leftarrow \underline{id}]} && \text{(car } \underline{id} \neq id)\\
&= \|\exists \underline{id}.Pre[\underline{id} \leftarrow id]\|_{e[[\![\tau]\!]_e \leftarrow id]} && \text{(par définition de l'interprétation de } \exists)\\
&= \|\exists \underline{id}.Pre[\underline{id} \leftarrow id]\|_{e'} && \text{(par définition de } e')
\end{aligned}$$

et donc $\exists \underline{id}.Pre[\underline{id} \leftarrow id](e')$.

De plus, nous avons aussi par définition de l'interprétation de $\exists$:

$$\|\exists \underline{id}.id = \tau[\underline{id} \leftarrow id]\|_{e[[\![\tau]\!]_e \leftarrow id]} = \max_{n\in\mathbb{N}} \|id = \tau[\underline{id} \leftarrow id]\|_{e[[\![\tau]\!]_e \leftarrow id][n\leftarrow \underline{id}]}.$$

Or, pour $n = e(id)$,

$$\begin{aligned}
[\![\tau[\underline{id} \leftarrow id]]\!]_{e[[\![\tau]\!]_e \leftarrow id][e(id)\leftarrow \underline{id}]} &= [\![\tau[\underline{id} \leftarrow id]]\!]_{e[e(id)\leftarrow \underline{id}]}\\
&\qquad \text{(car } id \text{ n'apparait pas dans } \tau[\underline{id} \leftarrow id])\\
&= [\![\tau]\!]_e && \text{(par le Lemme 7.3)}\\
&= [\![id]\!]_{e[[\![\tau]\!]_e \leftarrow id][e(id)\leftarrow \underline{id}]},
\end{aligned}$$

donc $\|id = \tau[\underline{id} \leftarrow id]\|_{e[[\![\tau]\!]_e \leftarrow id][e(id)\leftarrow \underline{id}]} = 1$, c'est-à-dire e' satisfait

$$\exists \underline{id}.id = \tau[\underline{id} \leftarrow id]$$

Nous concluons des deux parties de notre preuve (puisque dans les deux cas, nous avons pris $\underline{id} = e(id)$) que

$$\exists \underline{id}.Pre[\underline{id} \leftarrow id] \wedge id = \tau[\underline{id} \leftarrow id](e') \qquad \square$$

Exemple 7.8. Ainsi dans la situation précédente, nous aurons

$$\{x < y\}x \leftarrow x + 1\{\exists \underline{x}.\underline{x} < y \wedge x = \underline{x} + 1\},$$

l'ancienne condition est vraie pour l'ancienne valeur de la variable modifiée, et nous ajoutons une nouvelle égalité. Il est ensuite fréquent d'appliquer un affaiblissement pour éliminer le quantificateur existentiel, en substituant $\underline{x}$ par $x - 1$ dans ce cas :

$$\{x < y\}x \leftarrow x + 1\{x - 1 < y\},$$

et ici nous pouvons même, par l'intégralité de x, le réécrire en

$$\{x < y\}x \leftarrow x + 1\{x \leq y\}.$$

Pour conclure sur l'affectation, regardons le cas où la variable affectée est une nouvelle variable, n'apparaissant donc ni dans l'assertion préalable ni dans l'expression du membre droit de l'affectation. La règle se simplifie alors ainsi, puisque les substitutions n'ont pas d'effet.

Proposition 7.9. Pour tout identifiant id, toute assertion Pre et toute expression τ, tels

que Pre et τ ne contiennent pas id, nous avons

$$\{Pre\}id \leftarrow \tau\{Pre \wedge id = \tau\}$$

La séquence d'instructions. L'opérateur de séquence permet d'enchaîner deux programmes, les effets des deux programmes vont se combiner, d'une façon assez naturelle.

Proposition 7.10. Pour tous programmes p et q et toutes assertions Pre, Mid et $Post$, si
(i) $\{Pre\}p\{Mid\}$,
(ii) et $\{Mid\}q\{Post\}$,
alors $\{Pre\}p;q\{Post\}$.

Démonstration. Considérons trois états d'exécutions e, e' et e'' tels que $\langle e\rangle p\langle e'\rangle$ et $\langle e'\rangle q\langle e''\rangle$. Remarquons que par la sémantique de la séquence $\langle e\rangle p;q\langle e''\rangle$. Supposons $Pre(e)$. Alors par (i), $Mid(e')$ et par (ii), $Post(e'')$. Donc $\{Pre\}p;q\{Post\}$. □

Exemple 7.11.

- ▷ Nous avons $\{x > 2\}x \leftarrow x{+}1; x \leftarrow x{+}2\{x > 5\}$, puisque $\{x > 2\}x \leftarrow x{+}1\{\underline{x} > 2 \wedge x = \underline{x} + 1\}$ que nous affaiblissons en $\{x > 2\}x \leftarrow x+1\{x > 3\}$, et $\{x > 3\}x \leftarrow x+2\{\underline{x} > 3 \wedge x = \underline{x} + 2\}$. que nous affaiblissons en $\{x > 3\}x \leftarrow x + 2\{x > 5\}$.
- ▷ Nous avons $\{x \geq 0\}x \leftarrow x{+}1; x \leftarrow x{\times}x\{x \geq 0\}$, puisque $\{x \geq 0\}x \leftarrow x{+}1\{\underline{x} \geq 0 \wedge x = \underline{x} + 1\}$ que nous affaiblissons en $\{x \geq 0\}x \leftarrow x+1\{x \geq 1\}$, et $\{x \geq 1\}x \leftarrow x \times x\{\underline{x} \geq 1 \wedge x = \underline{x} \times \underline{x}\}$, que nous affaiblissons en $\{x \geq 1\}x \leftarrow x \times x\{x \geq 1\}$.

La boucle tant que. De notre mini-langage, il ne reste que la boucle à analyser. Pour rappel le bloc d'instruction est répétée jusqu'à ce que l'expression de contrôle s'évalue en 0. Considérons donc le programme

$$\textbf{tant que } \sigma \neq \tau \textbf{ faire } p.$$

La sémantique était simple à définir, puisque connaissant l'état d'exécution initial, nous pouvons déterminer la valeur de σ et τ pour un état précis, et donc par récurrence, calculer tous les états successifs jusqu'à la sortie de la boucle. En revanche, partant d'une assertion arbitraire, nous ne pouvons pas nécessairement déterminer si $\sigma = \tau$, et plus généralement combien d'itérations de la boucle seront nécessaires.

Nous savons néanmoins que l'exécution sera équivalente à

$$p; p; p; \ldots; p,$$

mais ignorons le nombre de répétitions. Cela suggère de mettre en place une stratégie qui ne dépende pas du nombre de répétitions. Nous faisons cela en cherchant une assertion Mid vraie après chaque répétition. En abusant de la notation des formules de Hoare (normalement un triplet), cela ressemblerait donc à :

$$\{Pre\}p\{Mid\}p\{Mid\}p\{Mid\}\ldots\{Mid\}p\{Post\}.$$

À ceci près qu'il nous manque un élément important : le bloc d'instruction, dans bien des cas, n'est valide que parce que la condition de continuation est fausse. Par exemple dans l'Algorithme 2.3 de division naturelle, si la condition était fausse, le bloc d'instruction effectuerait une division par zéro. Ou encore, nous pouvons penser à un algorithme basé sur une pile, qui extrait un élément de la pile à chaque itération. Si la pile est vide, le bloc d'instruction provoque une erreur.

Ainsi nous souhaitons une assertion qui, si elle est satisfaite et que la condition de continuation du **tant que** est vraie, est encore vraie après l'exécution du bloc d'instruction, ce qui s'écrit :

$$\{Mid \wedge \sigma \neq \tau\}p\{Mid\},$$

et notre répétition deviendrait

$$\{Pre\}p\{Mid \wedge \sigma \neq \tau\}p\{Mid \wedge \sigma \neq \tau\}\ldots\{Mid \wedge \sigma \neq \tau\}p\{Post \wedge \sigma = \tau\}.$$

Définition 7.12. Soient un programme p et une expression de condition $\sigma \neq \tau$. Une assertion A est un *invariant* de p relativement à la condition $\sigma \neq \tau$ si $\{A \wedge \sigma \neq \tau\}p\{A\}$.

Proposition 7.13. Soit un programme **tant que** $\sigma \neq \tau$ **faire** p, et soit Pre un invariant de p relativement à $\sigma \neq \tau$. Alors

$$\{Pre\}\textbf{tant que } \sigma \neq \tau \textbf{ faire } p\{Pre \wedge \sigma = \tau\}.$$

Démonstration. Soit $\langle e\rangle$**tant que** ϕ **faire** $p\langle e'\rangle$ et supposons que e satisfait Pre. La preuve de la proposition est par induction sur cette dérivation sémantique.
Si $[\![\sigma]\!]_e = [\![\tau]\!]_e$, alors $e = e'$, et donc l'assertion $Pre \wedge \sigma = \tau$ est satisfaite par e.
Si $[\![\sigma]\!]_e \neq [\![\tau]\!]_e$, alors il existe un état e' tel que $\langle e\rangle p\langle e''\rangle$ et $\langle e''\rangle$**tant que** $\sigma = \tau$ **faire** $p\langle e'\rangle$. Puisque Pre est un invariant, $\{Pre \wedge \sigma \neq \tau\}p\{Pre\}$. Comme l'assertion $Pre \wedge \phi \neq 0$ est satisfaite par e, l'assertion Pre est satisfaite par e''. Par induction nous en déduisons que $Pre \wedge \sigma = \tau$ est satisfaite par e''. □

En pratique, nous devons prouver que l'invariant est satisfait en fin de bloc en partant de l'hypothèse que l'invariant est satisfait en début de bloc. Cela nécessite le plus souvent au moins un affaiblissement, afin de retomber sur exactement la même formule. Par ailleurs, si nous quittons notre mini-langage et supposons que notre langage supporte les expressions booléennes, ϕ devient une expression booléenne, et le test d'égalité n'est qu'un cas particulier. La règle générale pour la boucle **tant que** est donc

Définition 7.14. Soient un programme p et une expression de condition ϕ. Une assertion A est un *invariant* de p relativement à ϕ si $\{A \wedge \phi\}p\{A\}$.

Proposition 7.15. Soit un programme **tant que** ϕ **faire** p, et soit Pre un invariant de p relativement à ϕ. Alors

$$\{Pre\}\textbf{tant que } \phi \textbf{ faire } p\{Pre \wedge \neg\phi\}.$$

Exemple 7.16. Considérons le programme **tant que** $x < n$ **faire** $x \leftarrow x + 1$. Nous nous doutons que lorsque le programme terminera, nous aurons $x = n$. Nous le démontrons en prenant comme invariant de $x \leftarrow x + 1$ relativement à la condition $x < n$, l'assertion

$$Pre = x \leq n$$

.

Vérifions d'abord qu'il s'agit d'un invariant : la règle de l'affectation nous donne

$$\{Pre \wedge x < n\}x \leftarrow x + 1\{\underline{x} \leq n \wedge \underline{x} < n \wedge x = \underline{x} + 1\},$$

ce qui se simplifie par un affaiblissement en

$$\{Pre \wedge x < n\}x \leftarrow x + 1\{Pre\},$$

puisque $x = \underline{x} + 1 < n + 1$ implique $x \leq n$.
Ainsi, nous avons par la Proposition 7.15,

$$\{Pre\}\text{tant que } x < n \text{ faire } x \leftarrow x + 1\{Pre \wedge \neg(x < n)\},$$

qui s'affaiblit en

$$\{Pre\}\text{tant que } x < n \text{ faire } x \leftarrow x + 1\{x = n\}.$$

La preuve de correction partielle d'un algorithme comprenant une boucle **tant que** fera donc toujours apparaître ces trois étapes :

1. donner et prouver un invariant pour le corps de la boucle ;
2. montrer que l'invariant est satisfait par l'état d'exécution avant l'évaluation de la boucle ;
3. appliquer la Proposition 7.15 pour en déduire une assertion satisfaite pour les états d'exécution après l'évaluation de la boucle.

Un exemple significatif. Prouvons que l'Algorithme 2.3 de division naturelle est partiellement correct. Puisque nous savons qu'il termine, cela prouvera sa correction. Rappelons sa pré-condition

$$\mathit{Pre} = n \geq 0 \wedge m > 0,$$

et sa post-condition

$$\mathit{Post} = \exists r. 0 \leq r \wedge r < m \wedge m_0 = q \times n_0 + r.$$

Rappelons les instructions de l'algorithme :

$\mathit{quotient} \leftarrow 0$
tant que $n \geq m$ **faire**
  $\mathit{quotient} \leftarrow \mathit{quotient} + 1$
  $n \leftarrow n - m$

La partie la plus subtile consiste à trouver l'invariant de la boucle. Nous sommes guidés par le fait que le quotient augmente de 1 à chaque itération, alors que n décroit de m dans le même temps. Ainsi la quantité $n + \mathit{quotient} \times m$ ne varie pas. Et nous pouvons aussi noter que m ne change pas non plus. Nous devrions donc pouvoir démontrer que l'assertion suivante est un invariant relativement à la condition $n \geq m$:

$$\mathit{Mid} = (n_0 = \mathit{quotient} \times m_0 + n) \wedge (m = m_0) \wedge (n \geq 0).$$

Nous commençons par démontrer qu'il s'agit d'un invariant, c'est-à-dire

$$\{\mathit{Mid} \wedge n \geq m\}\, \mathit{quotient} \leftarrow \mathit{quotient} + 1; n \leftarrow n - m \{\mathit{Mid}\}.$$

L'application de la règle de l'affectation deux fois nous donne d'abord

$$\{\mathit{Mid} \wedge (n \geq m)\}\, \mathit{quotient} \leftarrow \mathit{quotient} + 1 \left\{ \begin{array}{c} \mathit{Mid}[\underline{\mathit{quotient}} \leftarrow \mathit{quotient}] \wedge (n \geq m) \\ \wedge (\mathit{quotient} = \underline{\mathit{quotient}} + 1) \end{array} \right\},$$

puis (en notant $\underline{\mathit{Mid}} = \mathit{Mid}[\underline{\mathit{quotient}} \leftarrow \mathit{quotient}]$)

$$\left\{ \begin{array}{c} \underline{\mathit{Mid}} \wedge (n \geq m) \\ \wedge (\mathit{quotient} = \underline{\mathit{quotient}} + 1) \end{array} \right\} n \leftarrow n - m \left\{ \begin{array}{c} \underline{\mathit{Mid}}[\underline{n} \leftarrow n] \wedge (\underline{n} \geq m) \\ \wedge (\mathit{quotient} = \underline{\mathit{quotient}} + 1) \wedge (n = \underline{n} - m) \end{array} \right\}$$

qui se simplifie en

$$\left\{ \begin{array}{c} \underline{\mathit{Mid}} \wedge (n \geq m) \\ \wedge (\mathit{quotient} = \underline{\mathit{quotient}} + 1) \end{array} \right\} n \leftarrow n - m \left\{ \begin{array}{c} (n_0 = \underline{\mathit{quotient}} \times m_0 + \underline{n}) \wedge (m = m_0) \\ \wedge (\underline{n} \geq 0) \wedge (\underline{n} \geq m) \\ \wedge (\mathit{quotient} = \underline{\mathit{quotient}} + 1) \wedge (n = \underline{n} - m) \end{array} \right\}.$$

Cette formule de Hoare s'affaiblit en

$$\left\{ \begin{array}{c} \underline{\mathit{Mid}} \wedge (n \geq m) \\ \wedge (\mathit{quotient} = \underline{\mathit{quotient}} + 1) \end{array} \right\} n \leftarrow n - m \left\{ \begin{array}{c} (n_0 = (\mathit{quotient} - 1) \times m_0 + n + m) \\ \wedge (m = m_0) \wedge (n \geq 0) \end{array} \right\}$$

et finalement

$$\left\{ \begin{array}{c} \underline{\mathit{Mid}} \wedge (n \geq m) \\ \wedge (\mathit{quotient} = \underline{\mathit{quotient}} + 1) \end{array} \right\} n \leftarrow n - m \left\{ \begin{array}{c} (n_0 = \mathit{quotient} \times m_0 + n) \\ \wedge (m = m_0) \wedge (n \geq 0) \end{array} \right\}$$

en substituant m par m_0. Par la règle de la séquence d'instructions, nous obtenons bien que Mid est un invariant.

Pour terminer la preuve de correction partielle, considérons d'abord la première instruction d'affectation. En tant que première instruction, nous savons que l'état d'exécution initial satisfait Pre et aussi $n = n_0 \wedge m = m_0$. Par la règle de l'association, nous en déduisons

$$\{Pre \wedge n = n_0 \wedge m = m_0\} quotient \leftarrow 0 \{Pre \wedge n = n_0 \wedge m = m_0 \wedge quotient = 0\},$$

ce qui s'affaiblit en

$$\{Pre \wedge n = n_0 \wedge m = m_0\} quotient \leftarrow 0 \{Mid\}.$$

Par la règle de la séquence, et en notant p le programme tout entier, nous en déduisons

$$\{Pre \wedge n = n_0 \wedge m = m_0\} p \{Mid \wedge n < m\},$$

c'est-à-dire

$$\{Pre \wedge n = n_0 \wedge m = m_0\} p \left\{ \begin{array}{c} (n_0 = quotient \times m_0 + n) \\ \wedge (m = m_0) \wedge (n \geq 0) \wedge (n < m) \end{array} \right\},$$

qui s'affaiblit par l'introduction d'un quantificateur existentiel en

$$\{Pre \wedge n = n_0 \wedge m = m_0\} p \{\exists r.\ (n_0 = quotient \times m_0 + r) \wedge (r \geq 0) \wedge r < m_0\ \}.$$

Ainsi la post-condition et prouvée.

7.4 Extensions

Notre traitement du mini-langage de programmation est terminé, mais nous voulons utiliser la logique de Hoare dans nos contextes habituels de langages possédant de nombreuses fonctionnalités, au premier rang desquels nous trouvons les conditionnelles et les appels de fonctions. Nous donnons maintenant les triplets de Hoare correspondant, sans leurs preuves qui utiliseraient les mêmes techniques que celles que nous avons étudiées.

Les conditionnelles La structure de contrôle conditionnelle permet de choisir entre l'exécution de deux blocs d'instructions. Les triplets de Hoare correspondant vont devoir exprimer une assertion de conclusion qui unifie les résultats des deux branches.

Proposition 7.17. Soient p et p' deux programmes et ϕ une expression de condition, et Pre, $Post$ deux assertions, telles que

(i) $\{Pre \wedge \phi\} p \{Post\}$,

(ii) et $\{Pre \wedge \neg\phi\} p' \{Post\}$.

Alors

$$\{Pre\} \textbf{si } \phi \textbf{ alors } p \textbf{ sinon } p' \{Post\}.$$

Remarquons comment nous pouvons utiliser que la condition est vraie ou fausse, selon la branche, pour atteindre la nouvelle assertion $Post$.En général, la subtilité dans le calcul d'un triplet de Hoare pour la conditionnelle est de trouver cet affaiblissemt $Post$ commun aux deux branches possibles p et p'. Il est toujours possible de prendre la disjonction des deux résultats (de la forme $Post_1 \vee Post_2$), même si nous voudrons souvent trouver une assertion plus simple.

Exemple 7.18. Considérons la fonction calculant la valeur absolue d'un entier relatif.

valeurAbsolue(n)
Entrée: un entier $n \in \mathbb{Z}$;
Sortie: un entier r tel que $r \geq 0$ et ($r = n$ ou $r = -n$).

```
1: si n < 0 alors
2:     r ← −n
3: sinon
4:     r ← n
5: renvoyer r
```

Nous prenons $Pre = (n = n_0)$ et $Post = (r \geq 0 \wedge (r = n_0 \vee r = -n_0))$. Pour la branche *vraie* de la conditionnelle, nous avons

$$\{Pre \wedge n < 0\} r \leftarrow -n \{Pre \wedge n < 0 \wedge r = -n\},$$

qui s'affaiblit en

$$\{Pre \wedge n < 0\} r \leftarrow -n \{r < 0 \wedge r = -n_0\}.$$

Dans la branche *fausse* de la conditionnelle, nous avons

$$\{Pre \wedge n \geq 0\} r \leftarrow n \{Pre \wedge n \geq 0 \wedge r = n\},$$

qui s'affaiblit en

$$\{Pre \wedge n \geq 0\} r \leftarrow n \{r \geq 0 \wedge r = n_0\}.$$

Les assertions résultantes des deux branches ont pour affaiblissement commun :

$$r \geq 0 \wedge (r = n_0 \vee r = -n_0),$$

Par application de la proposition, nous en déduisons que

$$\{Pre\} \textbf{si } n < 0 \textbf{ alors } r \leftarrow -n \textbf{ sinon } r \leftarrow n \{r \geq 0 \wedge (r = n_0 \vee r = -n_0),\}$$

démontrant ainsi la post-condition de l'algorithme.

L'appel de fonction. L'appel de fonction est possible lorsque les pré-conditions de la fonction sont réalisées. Dans ce cas le résultat renvoyé vérifie les post-conditions de la fonction. Pour rester dans un cas simple, nous considérons uniquement le cas où

- ▷ les arguments passés à la fonction sont des variables, dont les identifiants correspondent aux noms des paramètres,
- ▷ le résultat de la fonction est directement placé dans une variable,
- ▷ la fonction ne modifie pas ses arguments (par exemple, une fonction qui trie un tableau reçu en paramètre modifie le tableau).

Il n'est pas compliqué de transformer un programme pour mettre tous les appels de méthodes sous ce format, et la règle que nous allons donné pourrait donc être étendue à n'importe quel appel de fonction, au prix de renommages fastidieux, qui seront ici pris en charge par d'éventuelles affectations supplémentaires.

Proposition 7.19. Soit *Mid* une assertion arbitraire. Soit f une fonction de paramètres $t_1, \ldots, t_k$, ayant pour pré-condition *Pre* et pour post-condition une assertion *Post* telle que la variable r dénote la valeur renvoyée (n'apparaissant ni dans *Pre* ni dans *Mid*). Alors

$$\{Mid \wedge Pre\} r \leftarrow \mathrm{f}(t_1, \ldots, t_k) \{Mid \wedge Post\}.$$

Ainsi, si la pré-condition est vraie avant l'appel de la fonction, la post-condition est vraie au retour de la fonction. Les autres assertions restent satisfaites tant que la fonction ne modifie pas ses arguments.

Exemple 7.20. Considérons le problème de mettre une fraction sous forme réduite : étant donné un numérateur $n \in \mathbb{N}$ et un dénominateur $m \in \mathbb{N} \setminus \{0\}$, la fraction irréductible égale à $\frac{n}{m}$ est donné par cet algorithme, qui utilise un appel à la fonction pgcd du calcul du plus grand commun diviseur.

fraction(n, m)
Entrée: un entier $n \in \mathbb{N}$, un entier non-nul $m \in \mathbb{N} \setminus \{0\}$;
Sortie: un fraction $r = \frac{n'}{m'}$ en forme réduite.
1: $d \geq \mathrm{pgcd}(n, m)$
2: $n' \leftarrow n/d$
3: $m' \leftarrow m/d$
4: $r \leftarrow \frac{n'}{m'}$

Précisons : la pré-condition de cet algorithme est

$$Pre = n_0 \geq 0 \wedge m_0 \geq 1,$$

et la post-condition que nous voulons démontrer est

$$Post = \frac{n_0}{m_0} = \frac{n'}{m'} \wedge (\forall k \in \mathbb{N} \setminus \{0\}, ((k \text{ divise } n') \wedge (k \text{ divise } m')) \implies k = 1).$$

La pré-condition de l'algorithme pgcd(n, m) est

$$Pre_1 = n \geq 0 \wedge m \geq 0 \wedge (n, m) \neq (0, 0),$$

qui est une conséquence logique de Pre. Sa post-condition pour la valeur de retour d est

$$Post_1 = (d \text{ divise } n) \wedge (d \text{ divise } m) \wedge \forall q > d, \neg((q \text{ divise } n) \wedge (q \text{ divise } m)).$$

Ainsi, en applicant la règle de l'appel de fonction, nous obtenons le triplet de Hoare

$$\{Pre \wedge n = n_0 \wedge m = m_0 \wedge Pre_1\} d \geq \text{pgcd}(n, m) \{Pre \wedge n = n_0 \wedge m = m_0 Post_1\}$$

que nous affaiblissons en

$$\{Pre \wedge n = n_0 \wedge m = m_0\} d \geq \text{pgcd}(n, m) \{Pre \wedge Post_1\}$$

En nommant P l'ensemble du programme et en appliquant les règles de l'affectation et de séquence, nous en déduisons après quelques simplifications :

$$\{Pre \wedge n = n_0 \wedge m = m_0\} P \left\{ \begin{array}{c} Pre \wedge (d \text{ divise } n_0) \wedge (d \text{ divise } m_0) \\ \wedge(\forall q > d, \neg((q \text{ divise } n_0) \wedge (q \text{ divise } m_0))) \\ \wedge n' = n/d \wedge m' = m/d \wedge r = \frac{n'}{m'} \end{array} \right\}$$

Nous affaiblissons l'assertion de droite, en démontrant $Post$:

- ▷ d'abord $\frac{n_0}{m_0} = \frac{n_0/d}{m_0/d}$ puisque d divise n_0 et d divise n_0, et donc $\frac{n_0}{m_0} = \frac{m'}{n'} = r$;
- ▷ ensuite, soit $k \geq 1$ et supposons que k divise n' et k divise m'. Alors kd divise n_0 et kd divise m_0, puisque $n' = n_0/d$ et $m' = m_0/d$. Comme $\forall q > d, \neg((q \text{ divise } n_0) \wedge (q \text{ divise } m_0))$, nous en déduisons que $kd \leq d$, et donc $k = 1$. Ainsi la deuxième partie de la post-condition est vraie.

Nous pouvons donc affaiblir le dernier triplet de Hoare en

$$\{Pre \wedge n = n_0 \wedge m = m_0\} P \{Post\}.$$

Le cas des fonctions récursives. Les fonctions récursives se traitent de la même façon, mais en supposant de prime abord que la post-condition est correcte lors des appels récursifs. Le but est alors de montrer la post-condition est correcte pour l'appel principal, et par induction la correction partielle de l'algorithme suit.

Prenons l'exemple de la factorielle et démontrons que son résultat r, pour l'entrée n, vaut $\prod_{i=1}^{n} i$. Rappelons ses instructions, que nous avons légèrement modifié pour pouvoir appliquer la règle de l'appel de fonction.

factorielle(n)
Entrée: un entier $n \in \mathbb{N}$;
Sortie: un entier $r = n!$.

```
1: si n = 0 alors
2:     r ← 1
3: sinon
4:     n' ← n − 1
5:     f ← factorielle(n')
6:     r ← f × n
```

Exprimons la pré-condition et la post-condition :

$$Pre = n \geq 0,$$

$$Post = r = \prod_{i=i}^{n_0} i.$$

Dans le but d'appliquer la règle des conditionnelles, nous étudions chaque branche, l'une après l'autre. Pour la branche positive, lorsque $n = 0$, nous avons

$$\{Pre \wedge n = n_0 \wedge n = 0\} r \leftarrow 1 \{Pre \wedge n = n_0 \wedge n = 0 \wedge r = 1\}$$

qui s'affaiblit sans difficulté en

$$\{Pre \wedge n = n_0 \wedge n = 0\} r \leftarrow 1 \left\{ r = \prod_{i=1}^{n_0} i \right\}.$$

Pour la première instruction de la deuxième branche, nous avons

$$\{Pre \wedge n = n_0 \wedge n \neq 0\} n' \leftarrow n - 1 \{Pre \wedge n = n_0 \wedge n \neq 0 \wedge n' = n - 1\}$$

qui s'affaiblit en

$$\{Pre \wedge n = n_0 \wedge n \neq 0\} n' \leftarrow n - 1 \{Pre \wedge n = n_0 \wedge n' = n_0 - 1 \wedge n' \geq 0\}.$$

Puisque $n' \geq 0$ est la précondition nécessaire pour l'appel récursif de l'algorithme, nous en déduisons :

$$\{Pre \wedge n = n_0 \wedge n' = n_0 - 1 \wedge n' \geq 0\} f \leftarrow \text{factorielle}(n-1) \left\{ Pre \wedge n = n_0 \wedge n' = n_0 - 1 \wedge f = \prod_{i=1}^{n'} i \right\},$$

puis en appliquent les règles de séquences et d'affectation, nous obtenons

$$\{Pre \wedge n = n_0 \wedge n \neq 0\} \left| \begin{array}{l} n' \leftarrow n - 1 \\ f \leftarrow \text{factorielle}(n') \\ r \leftarrow f \times n \end{array} \right. \left\{ Pre \wedge n = n_0 \wedge n' = n_0 - 1 \wedge \left(f = \prod_{i=1}^{n'} i \right) \wedge r = f \times n \right\},$$

qui s'affaiblit en

$$\{Pre \wedge n = n_0 \wedge n \neq 0\} \left| \begin{array}{l} n' \leftarrow n - 1 \\ f \leftarrow \text{factorielle}(n') \\ r \leftarrow f \times n \end{array} \right. \left\{ r = \prod_{i=1}^{n_0} i \right\}.$$

Ainsi, notant P le programme tout entier, par la règle de la conditionnelle, nous obtenons

$$\{Pre \wedge n = n_0\} P \left\{ r = \prod_{i=1}^{n_0} i \right\},$$

ce qui prouve la correction partielle de l'algorithme.

7.5 Présentation de la logique de Hoare

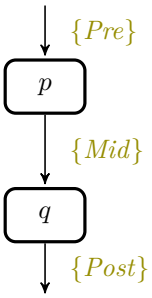
Nous avons présenté formellement la logique de Hoare sous la forme de triplets isolés. En pratique, lors de l'analyse de la correction d'un algorithme, il est souvent plus éloquent de représenter directement les assertions logiques au sein du programme ou de son graphe de flot de contrôle. Dans ce cas, chaque instruction ou bloc d'instruction se verra précédé et suivi par une assertion. Nous commençons par redonner quelques règles de la formule de Hoare sous le format du graphe de flot de contrôle, avant de donner des exemples complets.

La séquence d'instructions. Pour deux programmes p et q, avec

▷ $\{Pre\}p\{Mid\}$,

▷ et $\{Mid\}q\{Post\}$,

nous pouvons représenter les triplets de Hoare par ce graphe :

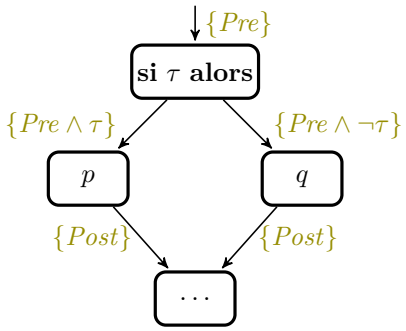


L'instruction conditionnelle Lors d'un branchement conditionnel, nous devons ajouter les assertions signifiant que la condition est ou n'est pas satisfaite au début de chaque branche. Ainsi, pour deux programmes p et q et une condition τ, telles que

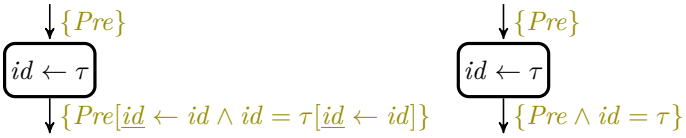
▷ $\{Pre \wedge \tau\}p\{Post\}$,

▷ et $\{Pre \wedge \neg\tau\}q\{Post\}$,

nous avons le graphe :



L'affectation Suivant que la variable existait déjà (et impose donc un renommage dans l'assertion) ou que l'identifiant de variable est nouveau, nous aurons une des deux notations :



qui peuvent aussi être présentées après un affaiblissement.

La boucle tant que. La boucle **tant que** se justifie par l'invariant : une assertion qui doit être vérifié avant l'entrée dans la boucle, et rester vraie après chaque itération. L'invariant doit donc être vrai avant l'exécution du bloc d'instructions interne à la boucle, et vrai après son exécution. Il convient donc de montrer

▷ $\{Inv \wedge \phi\}p\{Inv\}$,

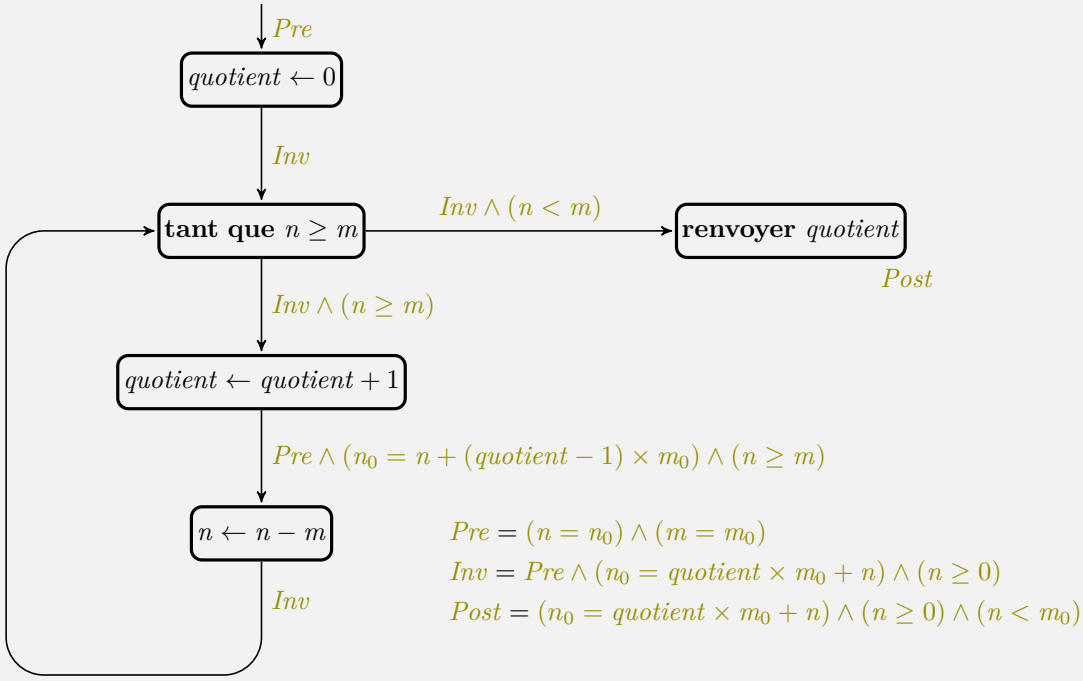
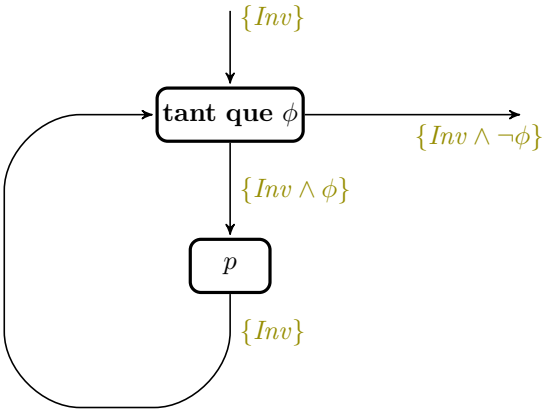


FIGURE 17 – L'algorithme de division entière, intégrant une preuve en logique de Hoare.

où ϕ est la condition de continuation de la boucle. Nous retrouvons donc l'invariant sur tous les arcs du graphe de flot de contrôle. Nous n'oublions pas d'ajouter la condition de continuation, positivement ou négativement, sur les deux arcs sortants du nœud correspondant au test de la condition. Nous obtenons le graphe ci-dessous :



Un exemple complet. Nous pouvons reprendre l'Algorithme 2.3 et en donnant une preuve en logique de Hoare, dont l'essentiel tient dans la Figure 17. Pour terminer la preuve, il reste à démontrer individuellement pour chaque instruction que le triplet de Hoare correspondant est correct, en partant de la règle de Hoare de l'instruction et en vérifiant qu'elle s'affaiblit bien dans le triplet requis.

Dans cet exemple, il convient donc de démontrer les triplets suivants :

- ▷ $\{Pre\}\, quotient \leftarrow 0 \{Inv\}$,
- ▷ $\{Inv \wedge n \geq m\}\, quotient \leftarrow quotient + 1 \{Pre \wedge (n_0 = n + (quotient - 1) \times m_0) \wedge (n \geq m)\}$,
- ▷ $\{Pre \wedge (n_0 = n + (quotient - 1) \times m_0) \wedge (n \geq m)\}\, n \leftarrow n - m \{Inv\}$,

ainsi que l'implication finale lors du **renvoyer** :

▷ $Inv \wedge n < m$ implique $Post$.

Une autre présentation consiste en l'ajout des assertions directement dans la description textuelle de l'algorithme, comme nous le faisons dans l'Algorithme 7.1.

divisionNat(n, m)
Entrée: n, un entier naturel, m, un entier naturel non nul.
Sortie: le résultat de la division entière de n par m

1: Pre
2: **soit** $quotient = 0$
3: Inv
4: **tant que** $n \geq m$ **faire**
5: $\quad Inv \wedge (n \geq m)$
6: $\quad quotient \leftarrow quotient + 1$
7: $\quad Pre \wedge (n_0 = n + (quotient - 1) \times m_0) \wedge (n \geq m)$
8: $\quad n \leftarrow n - m$
9: $\quad Inv$
10: $Inv \wedge (n < m)$
11: **renvoyer** $quotient$
12: $Post$

Algorithme 7.1 – Un algorithme pour la division entière, donné avec sa preuve en logique de Hoare. Les notations Pre, Inv et $Post$ sont les mêmes que dans la Figure 17.

Nous pouvons le constater sur cet exemple d'algorithme relativement simple, la logique de Hoare exige une extrême minutie. De fait, il s'agit d'un outil avant tout conçu pour permettre l'automatisation de la démonstration de correction des algorithmes, c'est-à-dire pour écrire des programmes capables de prouver qu'un algorithme est correct. Une automatisation complète n'est mathématiquement pas possible en toute généralité, mais de nombreux algorithmes peuvent être prouvés automatiquement pourvu qu'un humain fournisse suffisamment d'assertions logiques pour que l'ordinateur en déduise et prouve tous les triplets de Hoare.

Pour nous, nous retiendrons en premier lieu la technique de l'invariant pour démontrer la correction partielle d'une boucle : exhiber une propriété qui reste vraie à chaque itération de la boucle, et qui permet de montrer l'état final recherché. Combiné avec un variant bien-fondé, nous aurons à la fois la correction partielle et la terminaison, donc la correction de l'algorithme.

À retenir.

▷ La *logique de Hoare* consiste en l'établissement de triplets, comprenant une assertion, un programme et une autre assertion, de la forme $\{Pre\}p\{Post\}$. La deuxième assertion $Post$ décrit des propriétés sur l'état d'exécution après l'exécution du programme p, si la première assertion $Post$ était satisfaite avant l'exécution de p.

▷ Pour chaque instruction élémentaire, nous pouvons dériver automatiquement un triplet de Hoare.

▷ Un triplet de Hoare peut être *affaibli*, en remplaçant l'assertion $Post$ par une assertion plus faible, ou en remplaçant l'assertion Pre par une assertion plus restrictive.

▷ Le triplet de Hoare de la boucle **tant que** utilise un *invariant* : une assertion vraie avant la première itération, et qui reste vraie après chaque itération. Ainsi l'invariant est vraie à la fin de l'exécution de la boucle.

▷ L'appel d'une fonction nécessite que la pré-condition de la fonction soit une conséquence logique de Pre. Dans ce cas, $Post$ contient la post-condition de la fonction.

8 Complexité algorithmique

Dans cette dernière partie, nous nous intéressons à l'efficacité des algorithmes. Comment quantifier cette efficacité, quels sont les outils pour comparer les performances de deux algorithmes et en quoi l'efficacité importe, lorsque nous disposons de processeurs capables de réaliser des milliards d'opérations par seconde ?

8.1 Définir la complexité

Une efficacité se définit vis-à-vis d'une ressource, et en informatique nous distinguons en général deux grandes catégories de ressources :

- ▷ le nombre d'opérations élémentaires effectuées, c'est-à-dire la quantité de travail effectuée par les processeurs, ou de façon partiellement corrélée, l'énergie utilisée pour réaliser le calcul,
- ▷ l'espace mémoire utilisé, qui correspond pour sa part à la quantité de matière nécessaire pour réaliser le calcul.

En pratique, la plupart des applications algorithmiques ont une empreinte mémoire assez faible, d'autant que nous disposons de mémoires physiquement assez denses, ce qui fait que nous sommes plus souvent contraints par le nombre d'opérations élémentaires.

Les opérations élémentaires effectuées par un algorithme peuvent parfois être effectuées simultanément, au sein d'algorithmes exploitant les notions de concurrence et de parallélisme des calculs. D'autres fois les opérations élémentaires dépendent les uns des autres, par exemple la nature de la deuxième opération dépend du résultat de la première, et les opérations doivent donc être effectuées l'une après l'autre. De fait, les algorithmes qui nous intéressent dans ce cours sont tous écrits dans un modèle itératif, dans lequel les instructions sont effectuées l'une après l'autre, une à la fois. Dans ce cas, le nombre d'opérations élémentaires est alors corrélé à une autre notion physique : le temps nécessaire au calcul. Nous privilégions pour ce cours l'étude de l'efficacité temporelle des algorithmes, cette approche fournissant un cadre méthodologique qui pourra être étendu à l'analyse d'autres aspects de l'efficacité algorithmique.

Modèle de calcul. Une efficacité algorithmique se définit dans un modèle de calcul particulier. Une approche pragmatique consisterait à adopter comme modèle l'architecture d'un ordinateur moderne. L'efficacité pourrait alors être mesurée en chronométrant le temps de calcul de l'algorithme sur un échantillon représentatif de données d'entrée. Cependant, tout comme tester un programme ne permet pas de garantir l'absence de bugs, mesurer les temps de calculs d'un programme ne permet pas de garantir une efficacité temporelle dans tous les cas. Nous ne pouvons pas exclure que sur des instances particulières, l'algorithme puisse être beaucoup plus lent. Ainsi, si l'expérimentation est nécessaire pour évaluer l'efficacité d'un algorithme, elle ne peut pas entièrement se substituer à une analyse plus formelle. Ajoutons que les techniques formelles d'analyse de l'efficacité des algorithmes sont les seules à nous offrir un pouvoir prédictif quant à la performance de nos algorithmes.

Nous pourrions espérer calculer le temps de calcul d'un algorithme en listant la séquence de micro-instructions exécutées par le processeur, et en sommant le nombre de cycles consommés par chaque micro-instruction. Connaissant le nombre de cycles par seconde, nous obtiendrions le temps de calcul. Cette approche cumule plusieurs défauts :

- ▷ elle nécessite de compiler l'algorithme, pour une architecture précise,
- ▷ le nombre de cycles de chaque micro-instruction dépend du modèle de processeur,
- ▷ l'évaluation des micro-instructions ne se fait pas consécutivement : un processeur moderne évalue plusieurs micro-instructions simultanément, n'attendant pas la fin d'une instruction pour commencer l'évaluation de la suivante (quitte à devoir revenir en arrière),
- ▷ la fréquence du processeur s'adapte selon la charge de travail.

En bref, il n'existe rien de tel qu'une mesure exacte prédictible pour le temps d'évaluation d'un algorithme. De plus, nous voulons une méthode permettant d'ignorer les détails d'implémentation du langage de programmation ou de l'architecture matérielle, pour nous concentrer sur l'algorithme lui-même.

Nous devons donc adopter un modèle de calcul simple, adapté à l'analyse formelle. Il existe de nombreux modèles, offrant une variété de compromis entre simplicité du modèle et prise en compte de contraintes techniques comme les opérations supportées ou les formes de mémoire disponibles. Dans le cadre de ce cours, nous continuons simplement avec notre petit langage impératif et sa sémantique. Il est assez représentatif des principaux langages modernes et capture intuitivement la complexité des opérations telles qu'effectuées par le processeur d'un ordinateur, à une exception près. En effet, nous n'avons pas mis de contrainte sur les entiers contenus par les variables, là où un ordinateur moderne ne sait manipuler nativement que des entiers encodables sur un nombre fixe de bits (64 en général). Dans notre modèle de calcul, il est donc possible de faire l'addition de deux entiers de longueur arbitraire en une seule opération, ce qui n'est pas possible pour des entiers de plus de 64 bits sur une architecture moderne. Mise à part cette différence, que nous n'exploiterons pas, mesurer l'efficacité des algorithmes écrits dans notre langage impératif nous donnera des résultats réalistes au regard des mesures que nous pourrions être amenées à faire avec un ordinateur usuel. Dans ce cadre, nous choisissons comme mesure le nombre d'opérations arithmétiques (addition, multiplication, comparaison,...) et d'affectations effectuées par l'algorithme. En effet, ces opérations nous paraissent représentatives du travail effectué par l'algorithme. Cela reste un peu arbitraire, mais nous pourrons vérifier nos prédictions théoriques par rapport aux résultats empiriques.

Nombre d'opérations élémentaires Étant donnée un état initial d'exécution d'un programme, nous pouvons calculer le nombre d'opérations élémentaires en interprétant le programme comme le définit la sémantique de notre langage, en comptant le nombre d'opérations qui nous intéressent. Plus directement, nous en donnons une écriture inductive en se calquant sur celle de la sémantique.

Nous commençons par définir le coût d'évaluation $C(()\tau)$ d'une expression arithmétique, par induction sur τ :

$$\frac{}{C(n) : 0}\ n \in \mathbb{Z} \qquad \frac{}{C(id) : 0(id)}\ id \in \mathcal{I} \qquad \frac{C(\tau) : n \qquad C(\sigma) : p}{C(\tau \oplus \sigma) : 1 + n + p}\ \oplus \in \{+, -, \times, /\}$$

Accéder à la valeur d'une variable ne coûte rien, mais faire une opération arithmétique coûte une unité, peu importe quelle opération, plus le coût d'évaluation des opérandes gauche et droite. Ainsi nous simplifions en considérant qu'une addition et une division prenne le même temps de calcul. Nous étendons cette définition pour définir le coût d'évaluation $C(p, e)$ d'un programme depuis l'état d'exécution e, par induction sur p.

L'affectation coûte une unité, auquel il nous faut ajouter le coût d'évaluation de l'expression arithmétique.

$$\frac{}{C(id \leftarrow \tau, e) : 1 + C(()\tau)}\ id \in \mathrm{Dom}(e), \tau \in \mathcal{E}$$

La règle pour la séquence est immédiate, nous additionnons le coût des deux parties du programme.

$$\frac{C(p_1, e) : n \qquad C(p_2, e') : p}{C(p_1; p_2, e) : n + p}\ \langle e \rangle p_1 \langle e' \rangle$$

Enfin, pour la boucle **tant que**, nous devons distinguer selon le résultat de l'évaluation de la condition.

$$\frac{}{C(\textbf{tant que } \sigma = \tau \textbf{ faire } p, e) : 1 + C(\sigma) + C(\tau)}\ \sigma, \tau \in \mathcal{E}, [\![\sigma]\!]_e = [\![\tau]\!]_e$$

$$\frac{C(p, e) : n \qquad C(\textbf{tant que } \sigma = \tau \textbf{ faire } p, e') : m}{C(\textbf{tant que } \sigma = \tau \textbf{ faire } p, e) : 1 + C(\sigma) + C(\tau) + n + m}\ \sigma, \tau \in \mathcal{E}, [\![\sigma]\!]_e \neq [\![\tau]\!]_e, \langle e \rangle p \langle e' \rangle$$

Ceci nous définit effectivement le coût d'un calcul. Cependant, cette définition n'est pas immédiatement exploitable. Nous pouvons calculer le coût pour une entrée particulière, mais nous ne pouvons pas exprimer de propriété générale s'appliquant à toutes les entrées. Comme la logique de Hoare, que nous dérivons de la sémantique en remplaçant les états d'exécution par des assertions logiques vraies pour ces états, il nous faut généraliser en établissant non pas le coût précis, mais une approximation valable pour toutes les entrées admissibles.

Taille des entrées. Puisque nous voulons établir une relation entre le coût d'évaluation d'un algorithme et ses entrées, nous devons montrer une dépendance entre le coût d'évaluation et certaines caractéristiques des entrées. Un choix naturel et universel consiste en exprimer le coût de calcul en fonction de la taille de l'entrée. Par exemple, trier un tableau de longueur 1000 devrait nécessiter plus de travail que de trier un tableau de longueur 100. Ou calculer la racine carrée d'un nombre à 100 chiffres demande vraisemblablement plus de travail que de calculer la racine carré d'un nombre à 10 chiffres. Ainsi, nous donnerons systématiquement nos évaluations du temps de calcul en fonction de la taille de l'entrée.

Définition 8.1. La *taille* d'une entrée d'un algorithme est le nombre de bits dans sa représentation dans un encodage binaire (précisé par l'algorithme).

En pratique, nous n'aurons pas besoin d'être extrêmement précis dans le calcul de la taille, une estimation suffira. Nous considérerons donc pour simplifier, et sauf indication contraire, que :

- ▷ un entier n nécessite $\log_2 n$ bits,
- ▷ un nombre à virgule flottante avec une précision de k chiffres décimaux nécessite $4k$ bits,
- ▷ un tableau ou une liste de n valeurs, chaque valeur étant représentable avec b bits, nécessite $n \times b$ bits,
- ▷ un arbre de n nœuds, chaque nœud étant représentable avec b bits, nécessite $n \times b$ bits,
- ▷ un tuple de k éléments, prenant respectivement $b_1, b_2, \ldots, b_k$ bits, nécessite $b_1 + b_2 + \ldots + b_k$ bits.

Cela sous-estime un peu le nombre de bits nécessaires, mais, poursuivant notre approximation du coût des opérations élémentaires, nous pouvons aussi nous autoriser une petite approximation dans le calcul de la taille des entrées.

La complexité algorithmique. Notre objectif est donc désormais de lier le coût d'évaluation de l'algorithme avec la taille de ses entrées, autrement dit à exprimer une fonction qui, pour une taille d'entrée donnée, nous donne une estimation du coût d'évaluation de l'algorithme sur les entrées ayant cette taille. Comme une même taille d'entrée correspond à toute une famille d'entrées, et que chacune de ces entrées peut avoir un coût d'évaluation différent, il nous faut définir une façon d'agréger tous ces coûts d'évaluation en une seule mesure. Nous présentons deux définitions possibles.

Définition 8.2. La *complexité dans le pire des cas* d'un algorithme A est la fonction qui a un entier $n \in \mathbb{N}$, associe le maximum du coût d'évaluation de A sur une entrée de taille au plus n.

La complexité dans le pire des cas consiste donc en mesurer pour chaque taille d'entrée, le coût le plus élevé pour toutes les entrées de cette taille.

Définition 8.3. La *complexité moyenne* d'un algorithme A est la fonction qui a un entier $n \in \mathbb{N}$, associe la moyenne du coût d'évaluation de A sur toutes les entrées de taille n.

La complexité dans le pire de cas représente une garantie forte sur le coût d'évaluation d'un algorithme. Pour beaucoup d'algorithmes, le coût d'évaluation ne diffèrent qu'assez peu entre deux instances de même taille, et donc la complexité dans le pire des cas correspondra assez bien aux performances réelles de l'algorithme. Au contraire, pour certains algorithmes, cette mesure pourrait ne pas être très représentative des performances de l'algorithme, en particulier sur des entrées typiques que nous souhaiterions résoudre en pratique. Cet écart peut s'expliquer, soit parce que les entrées maximisant le coût d'évaluation sont très rares, soit parce que ces pires entrées sont très différentes de celles issues des instances réels. La complexité moyenne permet dans ce cas de donner une estimation peut-être plus fidèle du coût d'évaluation de l'algorithme. D'autres

phénomènes encore peuvent affectés la pertinence de ces deux notions de complexité algorithmique, comme l'utilisation d'aléa par l'algorithme, la prédominance des lectures/écritures sur des mémoires distances, l'utilisation du parallélisme ou de la concurrence. Pour ces cas particuliers, d'autres notions de complexité algorithmique plus adaptées peuvent être définies et utilisées. La pertinence de l'analyse dans le pire des cas n'est est pas moins réelle : elle prédit correctement les performances de nombreux algorithmes, et fourni toujours une borne supérieure pouvant être améliorée par des analyses plus raffinées.

8.2 Comparer les complexités d'algorithmes

Le comportement asymptotique des fonctions. Ayant établi qu'une complexité algorithmique est une fonction de la taille des entrées vers le coût d'évaluation maximum pour les entrées de la taille choisie, comment comparer l'efficacité relative de plusieurs algorithmes résolvant la même tâche ? Qu'est-ce qui constitue une complexité signalant un algorithme efficace, ou un algorithme inefficace ?

Nous devons prendre plusieurs facteurs en considération :

- ▷ Les ordinateurs sont désormais tellement rapides que la plupart des tâches sur des entrées de petite taille prennent un temps négligeable, même avec des algorithmes inefficaces. Nous pouvons garder comme point de référence qu'un million d'opérations s'effectuent quasi-instantanément, un milliard d'opérations nécessitent quelques secondes, et mille milliards plusieurs heures. Hormis dans certaines situations extrêmes, nous sommes donc surtout intéressés par des algorithmes qui sont efficaces pour des grandes tailles d'entrée.
- ▷ Un algorithme peut être très efficace pour des entrées de taille 100, mais devenir très inefficace pour des entrées de taille 1000 ou plus. Nous lui préférerons alors souvent un algorithme peut-être moins efficace pour les entrées de taille 100, mais qui sera capable de garder son efficacité pour des entrées de taille 1000 ou plus.
- ▷ Historiquement, la puissance et la vitesse de calcul des ordinateurs a augmenté à un rythme exponentiel. Cela signifie que pour un algorithme donné sur une entrée donnée, le temps de calcul effectif (ou l'énergie consommée) se réduit significativement année après année. L'enjeu algorithmique n'est donc pas temps d'améliorer les performances pour une taille d'entrée fixe, ceci étant réalisable par les progrès techniques ou par la mobilisation de plus de resources (plus de machines) jusqu'à atteindre des coûts négligeables. L'enjeu est plutôt de pouvoir traiter des données de plus en plus grosses.

Ainsi, pour ces raisons, nous sommes surtout intéressés par le comportement des algorithmes pour des données de taille plus en plus grandes, c'est-à-dire par le comportement de notre fonction de complexité quand n devient arbitrairement grand. C'est ce que nous appelons le *comportement asymptotique* de la fonction.

Définition 8.4. Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. Nous dirons que f est *asymptotiquement dominée* par g, et noterons $f = O(g)$, s'il existe $c \in \mathbb{R}$, et $N \in \mathbb{N}$, tels que pour tout $n \geq N$, $|f(n)| \leq c|g(n)|$.

Lorsque g ne s'annule pas, la condition peut se réécrire $\lim_{n\to+\infty} \left|\frac{f(n)}{g(n)}\right| < +\infty$. Une façon d'interpréter cette définition consiste à dire que $|f(n)|$ ne peut pas devenir arbitrairement plus grand que $|g(n)|$, lorsque n tend vers $+\infty$: $|f(n)|$ est limité par $c|g(n)|$.

La notation $f = O(g)$, qui se lit « f est en grand-O de g », est trompeuse, dans le sens qu'il ne s'agit pas d'une égalité, nous ne pouvons pas dire par exemple $O(g) = f$. Il s'agit en fait d'une relation de pré-ordre.

Proposition 8.5. La relation *être asymptotiquement dominée* est une relation de pré-ordre (elle est réflexive et transitive).

Démonstration. Soient f, g et h trois fonctions de $\mathbb{N}$ dans $\mathbb{R}$.

- ▷ (Réflexivité) Prenons $c = 1$ et $N = 0$, alors pour tout $n \geq N$, $|f(n)| \leq 1 \times |f(n)| = |f(n)|$, donc $f = O(f)$.
- ▷ (Transitivité) Supposons que $f = O(g)$ et $g = O(h)$, par définition il existe N_g, c_g et N_h, c_h tel que
  - — pour tout $n \geq N_g$, $|f(n)| \leq c_g \times |g(n)|$,
  - — pour tout $n \geq N_h$, $|g(n)| \leq c_h \times |h(n)|$.

  Prenons donc $N_f = \max\{N_g, N_h\}$ et $c_f = c_g \times c_h$. Soit $n \geq N_f$. Puisque $n \geq N_f \geq N_g$, nous avons que
  $$|f(n)| \leq c_g \times |g(n)|.$$
  Et puisque $n \geq N_f \geq N_h$, nous avons aussi que
  $$|g(n)| \leq c_h \times |h(n)|.$$
  Combinant ses deux inégalités, nous obtenons
  $$|f(n)| \leq c_g \times c_h \times |h(n)| = c_f \times |h(n)|.$$
  Ainsi $f = O(h)$. □

En particulier, si $f = O(g)$ et $g = O(h)$, alors $f = O(h)$. La relation de domination possède plusieurs autres propriétés qui nous seront utiles :

Proposition 8.6. Soient f_1, f_2, g_1, g_2 quatre fonctions de $\mathbb{N}$ dans $\mathbb{R}$, avec $f_1 = O(g_1)$ et $f_2 = O(g_2)$. Alors

(i) $f_1 + g_1 = O(g_1)$,

(ii) $f_1 \times f_2 = O(g_1 \times g_2)$,

(iii) pour $k \in \mathbb{N}$, $kf_1 = O(f_1)$.

Démonstration. Puisque $f_1 = O(g_1)$ et $f_2 = O(g_2)$, il existe $N_1, N_2 \in \mathbb{N}$, $c_1, c_2 \in \mathbb{R}^+$ tels que

- ▷ pour tout $n \geq N_1$, $|f_1(n)| \leq c_1 \times |g_1(n)|$,
- ▷ pour tout $n \geq N_2$, $|f_2(n)| \leq c_2 \times |g_2(n)|$.

(i) Posons $N = N_1$ et $c = c_1 + 1$, alors pour tout $n \geq N$, nous avons
$$\begin{aligned} |(f_1 + g_1)(n)| = |f_1(n) + g_1(n)| &\leq |f_1(n)| + |g_1(n)| \\ &\leq c_1 \times |g_1(n)| + |g_1(n)| \\ &= c \times |g_1(n)|. \end{aligned}$$

(ii) Soit $N = \max\{N_1, N_2\}$ et $c = c_1 \times c_2$, alors pour tout $n \geq N$, puisque $n \geq N_1$ et $n \geq N_2$, nous avons
$$\begin{aligned} |(f_1 \times f_2)(n)| = |f_1(n) + f_2(n)| = |f_1(n)| \times |f_2(n)| \\ &\leq c_1 \times |g_1(n)| \times c_2 \times |g_2(n)| \\ &= (c_1 \times c_2) \times |g_1(n)| \times c \times |g_2(n)| \\ &= c \times |(|g_1 \times g_2|)|(n)|. \end{aligned}$$

(iii) Soit $N = 0$ et $c = k$, alors
$$|(kf_1)(n)| = k \times |f_1(n)| = c \times |f_1(n)|.$$ □

Ainsi, (i) donne une règle pour simplifier l'addition : le plus grand terme (dans le sens de la dominance asymptotique) l'emporte et nous pouvons supprimer les autres. L'addition se comporte donc comme un maximum. La règle (iii) nous permet de supprimer les constantes multiplicatives.

Notre postulat est qu'en présence de deux algorithmes de complexités respectives f et g, si $f = O(g)$, alors l'algorithme f est au moins aussi efficace que l'algorithme g. En effet, cela correspond au fait que pour des entrées de tailles suffisantes, f est au moins aussi efficace que cg pour une constante multiplicative $c \in \mathbb{R}^+$. D'un point de vue théorique, cette constante ne nous importe pas, puisque nous avons approximé le coût des opérations élémentaires à une constante près aussi ; cela correspond au manque de précision que nous avons délibérement introduit dans notre méthode, pour prendre en contre notre ignorance des architectures pouvant évaluer nos algorithmes et affectant le coût du calcul.

Il est d'usage en informatique d'abuser des notations, en notant $O(f(x))$ au lieu de $O(f)$, par exemple écrire $O(x^2)$ au lieu de $O(f)$ avec $f : x \to x^2$, ou même écrire $x^2 \log x = O(x^3)$ plutôt que le plus correct $f = O(g)$ avec $f : x \to x^2 \log x$ et $g : x \to x^3$.

Autres relations asymptotiques. Nous pourrons être amenés à utiliser deux autres notions relatives au comportement asympotique des fonctions.

Définition 8.7. Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. Nous dirons que f est *asymptotiquement négligeable* par rapport à g, et noterons $f = o(g)$ (prononcé « petit-O de g »), si pour tout $\varepsilon > 0$, il existe $N \in \mathbb{N}$ tel que pour tout $n > N$, $f(n) < \varepsilon g(n)$.

Lorsque g ne s'annule pas, la condition peut se réécrire $\lim_{n\to+\infty} \frac{f(n)}{g(n)} = 0$. Le principe de la définition est de dire que nous pouvons prendre une marge ε aussi petite que souhaitée, par exemple $\varepsilon = 1/1000000$, et alors, pour toutes les valeurs de n suffisamment grandes, $f(n)$ sera plus petit que $g(n)$ par un facteur ε : $g(n)$ vaut au moins 1000000 de fois plus que $f(n)$. La relation *être asymptotiquement négligeable* est transitive et asymétrique. De plus, en prenant $\varepsilon = 1$, nous obtenons que $f = o(g)$ implique $f = O(g)$. En quelque sorte, $o(g)$ est une version stricte de $O(g)$: f est asymptotiquement vraiment plus petit que g, f et g ne sont pas du même ordre de grandeur.

Définition 8.8. Soient f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. Nous dirons que f et g sont *de même ordre de grandeur*, noté $f = \Theta(g)$, s'il existe $k_1, k_2 \in \mathbb{R}^+$ et $N \in \mathbb{N}$ tels que pour tout $n > N$, $k_1|g(n)| \leq |f(n)| \leq k_2|g(n)|$.

Puisque nous ne sommes pas intéressés par les facteurs multiplicatifs, deux algorithmes dont les complexités sont du même ordre de grandeur ne sont pas distinguables par nos critères théoriques.

Les notations $f = o(g)$, $f = O(g)$ et $f = \Theta(g)$ sont appelés *notations de Landau*.

Une échelle des complexités. Puisque la relation de domination induit un pré-ordre partiel sur les fonctions de complexité, nous pourrons classer nos algorithmes du plus efficace au moins efficace. Le pré-ordre n'est pas total, mais la plupart du temps les fonctions de complexité que nous considérerons seront suffisamment simples et comparables entre elles. De fait, elle appartiendront aux quelques catégories que nous allons maintenant décrire, en ordre croissant de complexité. Nous noterons n la taille de l'entrée.

- ▷ La complexité constante $O(1)$ correspond aux algorithmes qui effectuent un nombre borné d'opérations, indépendamment de la taille de leur entrée. Par exemple, déterminer si un entier représenté en binaire est pair, puisqu'il suffit de vérifier que le bit de poids le plus faible est un 0.
- ▷ La complexité logarithmique $O(\log n)$, avec par exemple la recherche dichotomique et ses applications. Les algorithmes de complexité logarithmique n'ont pas le temps de lire toute leur entrée, en fait ils doivent même en lire une partie négligeable seulement, puisque $\log n = o(n)$. Ce sont souvent des algorithmes capables de diviser successivement leur espace de recherche en deux.
- ▷ Les complexités polylogarithmiques, de la forme $(\log^k n)$ pour $k > 1$.
- ▷ La complexité linéaire $O(n)$ regroupe des algorithmes qui ne sont pas beaucoup plus coûteux que la lecture complète de l'entrée. De fait, tout algorithme dont la réponse dépend de chaque bit de l'entrée aura une complexité au moins linéaire.

- ▷ La complexité quasi-linéaire $O(n \log n)$, qui correspond à celle des meilleurs algorithmes de tri par comparaison, et donc à beaucoup d'algorithmes nécessitant un tri. En pratique, $\log n$ étant rarement bien grand, ses algorithmes sont considérés comme assez efficace pour toutes les applications ne travaillant pas sur des volumes de données extrêmes.
- ▷ La complexité polynomiale $O(n^d)$. Comme $n \log n = o(n^d)$ dès lors que $d > 1$, ce sont des algorithmes en général moins efficace que les deux précédentes catégories. En pratique, d vaut rarement plus que 3, ce qui permet de travailler avec des instances de plusieurs millier de bits, ou même millions si $d = 1.5$ par exemple (c'est-à-dire $O(n\sqrt{n})$). La recherche d'une plus long séquence commune entre deux brins d'ADN est un exemple de problème ayant comme meilleures solutions des algorithmes de complexité (n^2).
- ▷ La complexité exponentielle $O(2^n)$ ou plus généralement $O(c^n)$ avec $c > 1$. Nous y trouvons des algorithmes de recherche par force brute : tester toutes les combinaisons possibles. Ces algorithmes sont considérés comme inefficaces, ne pouvant être raisonnablement exécutés que pour $n < 50$ environ. Cependant, puisque nous parlons complexité dans le pire des cas, et que les cas d'usage ne sont pas toujours les pires, certains algorithmes exponentiels peuvent afficher des performances décentes en pratique, à l'image des SAT-Solvers.

Nous admettrons le résultat suivant, qui permet de montrer que nos catégories sont bien d'efficacité décroissantes.

Proposition 8.9. Pour tous réels $c > 1$ et $d > 0$,
(i) si $f = O(1)$ alors $f = o(\log^d n)$,
(ii) si $f = O(\log n)$ alors $f = o(n^d)$,
(iii) si $f = O(n)$ alors $f = o(c^n)$.

Pour illustrer la pertinence de cette échelle de complexité, nous pouvons comparer le temps de calcul pour différentes tailles d'entrées selon les fonctions de complexités, en supposant l'utilisation d'un ordinateur effectuant 1 milliard d'opérations par seconde.

n	10	100	1000	10000	100000	1000000
$\log n$	4ns	8ns	11ns	14ns	17ns	20ns
$\log^2 n$	16ns	64ns	121ns	196ns	289ns	400ns
n	10ns	100ns	1µs	10µs	100ns	1ms
$n \log n$	40ns	800ns	11µs	140ns	1.7ms	20ms
n^2	100ns	10µs	1ms	100ms	10s	17min
n^3	1µs	1ms	1s	17min	12j	3 ans
2^n	1µs	10^{13} ans				

Sur le même ordinateur :

- ▷ pour une complexité de $O(\log n)$, résoudre un problème de taille double $n' = 2n$ demande un nombre constant d'opérations supplémentaires, n'influençant quasiment pas le temps de calcul ;
- ▷ pour une complexité linéaire $O(n)$, doubler la taille de l'entrée double aussi le temps de calcul ;
- ▷ pour une complexité quadratique $O(n^2)$, doubler la taille de l'entrée quadruple le temps de calcul ;
- ▷ pour une complexité cubique $O(n^3)$, doubler la taille de l'entrée multiplie par 8 le temps de calcul ;
- ▷ pour une complexité cubique $O(2^n)$, augmenter de 1 le nombre de bits de l'entrée double le temps de calcul, augmenter de 10 bits le multiplie par 1000, et doubler la taille de l'entrée est irréaliste sauf sur des entrées de toute petite taille.

Inversement, investir dans un ordinateur deux fois plus rapide permet d'utiliser des algorithmes linéaires sur des entrées deux fois plus grandes, et 40% plus grandes pour un algorithme en $O(n^2)$, mais ne permet d'augmenter que de 1 le nombre de bits des entrées d'un algorithme en $O(2^n)$.

8.3 Comment évaluer une complexité dans le pire des cas ?

Nous pouvons utiliser notre définition formelle du coût d'évaluation sur une entrée de taille n arbitraire, mais sans connaître précisement e, il n'est pas toujours possible de calculer ce coût d'évaluation. Nous donnons ici quelques règles permettant de calculer une sur-estimation du coût d'évaluation. Elle est guidée par deux principes :

- ▷ ignorer les spécificités de l'entrée pour ne retenir que sa taille d'une part,
- ▷ ignorer les termes négligeables et calculer une borne supérieure sur la complexité à une constante multiplicative près, en utilisant les notations de Landau.

La séquence d'instructions Si pour une entrée commune, la complexité du programme p est donnée par f_p et celle de q par f_q, alors $p;q$ a pour complexité $f_{p;q} = O(f_p + f_q)$. Ainsi la complexité d'une séquence fixe d'instruction est dominée par celle de l'instruction la plus chère.

La conditionnelle Avec les même notations et f_ϕ pour la fonction de complexité de l'évaluation de la formule ϕ, un programme de la forme si ϕ alors p sinon q aura une complexité $O(f_\phi + f_p + f_q)$. Ceci se simplifie généralement en $O(f_p)$, $O(f_q)$ ou $O(f_\phi)$ selon laquelle des 3 fonctions domine les 2 autres.

La boucle tant que. Pour évaluer une boucle **tant que** ϕ **faire** p, une technique simple consiste en borner le nombre d'itérations de la boucle. Cela se fait avec la même technique que le variant bien-fondé, excepté qu'il faut maintenant trouver une borne sur la plus longue suite décroissante possible depuis l'état d'exécution initial de la boucle. C'est naturellement plus facile lorsque le variant est un entier positif qui décroit de 1 à chaque itération : le nombre d'itération est alors la différence entre la valeur initiale de cet entier et sa valeur finale. Si $g(n)$ est le nombre d'itérations de la boucle, exprimé en fonction de la taille n de l'entrée de l'algorithme, alors la boucle a pour complexité $O(g \times (f_\phi + f_p))$.

Il est parfois nécessaire d'utiliser des techniques complexes pour borner le nombre des itérations. Dans d'autres cas, le coût d'évaluation de p dépend fortement de l'itération : par exemple quelques itérations vont être très longues, alors que les autres seront très rapides. Cela demande une analyse capable de distinguer précisément les coûts des itérations. Considérons donc une série d'itérations de la forme

$$\langle e_0 \rangle p \langle e_1 \rangle p \langle e_2 \rangle \dots \langle e_{k-1} \rangle p \langle e_k \rangle,$$

alors le coût d'évaluation est

$$\sum_{i=0}^{k-1} C(p, e_i).$$

Le calcul précis de la complexité demande donc de caractériser les états intermédiaires e_i, par exemple avec la logique de Hoare, puis de sommer les coûts $C(p, e_i)$ pour chaque i possible.

La boucle pour tout. Cette boucle peut se réécrire comme une boucle **tant que** sans difficulté, les mêmes techniques vont donc s'appliquer. Nous disposons par contre de l'avantage de pouvoir facilement borner le nombre d'itérations.

Un cas typique est de parcourir toutes les paires d'entiers i et j avec $1 \leq i < j \leq n$, comme pour le programme q suivant :

```
1: pour i ∈ {1, 2, ..., n} faire
2:     pour j ∈ {i + 1, i + 2, ..., n} faire
3:         p
```

Nous obtenons :

$$f_q = O\left(\sum_{i=1}^{n} \sum_{j=i+1}^{n} f_p\right) = O(n^2 f_p).$$

Les relations suivantes sont ainsi parfois utiles :

Proposition 8.10. Nous avons

$$\sum_{i=1}^{n} i = O(n^2) \qquad \sum_{i=1}^{n} i^2 = O(n^3) \qquad \dots \qquad \sum_{i=1}^{n} i^d = O(n^{d+1}).$$

L'appel de fonction. Le coût d'un appel de fonction devra être déterminé en fonction de la taille des arguments qui sont passées à la fonction. Les appels récursifs nécessitent des techniques spéciales pour calculer leur complexité.

Exemple de calcul. Nous reprenons l'exemple de l'algorithme de division entière, et analysons sa complexité. Nous pouvons annoter chaque ligne par la contribution de la ligne à la complexité. Pour les boucles, nous annotons avec le nombre d'itérations, puisque ici nous pouvons le calculer explicitement. Lorsque cela n'est pas possible, nous pouvons annoter la boucle avec une estimation, qui dans le cadre d'une analyse dans le pire des cas, doit être au moins aussi grande que le nombre d'itérations réel. Le résultat de ces annotations est visible dans l'Algorithme 8.1.

divisionNat(n, m)
Entrée: n, un entier naturel, m, un entier naturel non nul.
Sortie: le résultat de la division entière de n par m

```
1: soit quotient = 0                          ▷ O(1)
2: tant que n ≥ m faire                       ▷ n/m×
3:     quotient ← quotient + 1                ▷ O(1)
4:     n ← n − m                              ▷ O(1)
5: renvoyer quotient                          ▷ O(1)
```

Algorithme 8.1 – Analyse de la complexité d'un algorithme pour la division entière.

L'analyse se conclue en sommant toutes les lignes, en n'oubliant pas de multiplier les complexités à l'intérieur de la boucle par le nombre d'itérations (ou son estimation). Nous obtenons :

$$O(1) + (n/m) \times (O(1) + O(1)) + O(1) = O(n/m).$$

8.4 Calculer la complexité d'un algorithme récursif

Équations récursives de complexité. La méthode précédente ne s'applique pas directement aux algorithmes récursifs, puisque nous ne connaissons pas par avance le coût des appels récursifs. Pour cela, nous commencerons par introduire ce coût, sous la forme d'une fonction inconnue, puis nous conduisons l'anaalyse de complexité normalement. Cela va nous donner une équation (ou inéquation), dont la fonction de complexité est l'inconnue.

Prenons l'exemple de l'algorithme récursif du calcul de la factorielle d'un entier. Nous posons donc $T(n)$ le coût d'évaluation de cet algorithme sur l'entier n. Pour ajouter un peu de piquant, nous supposerons cette fois travailler avec des entiers de précision arbitraire, ce qui signifie que les opérations arithmétiques ont une complexité en $O(\log p)$ sur des entiers inférieurs à p. Nous annotons l'algorithme par les coûts, en utilisant T pour les appels récursifs, comme indiqué dans l'Algorithme 8.2.

amU Faculté des sciences Aix Marseille Université

factorielle(n)
Entrée: n un entier positif ;
Sortie: la factorielle de n.

```
1: si n = 0 alors                          ▷ O(1)
2:     renvoyer 1                          ▷ O(1)
3: soit r = factorielle(n − 1)             ▷ T(n − 1) + O(log n)
4: renvoyer n × r                          ▷ O(log((n − 1)!))
```

Algorithme 8.2 – Analyse de complexité pour l'algorithme récursif de calcul de la factorielle.

En ligne 3, le calcul de $n-1$ est une opération arithmétique de coût $\log n$. Nous y ajoutons le coût de l'évaluation de l'appel récursif $T(n-1)$. En ligne 4, nos devons compter le coût de la multiplication, qui est donné par le logarithme de la plus grand opérande, ici r. Puisque r vaut $(n-1)!$, cette ligne coûte $O(\log((n-1)!))$. Enfin nous sommons toutes les instructions, remarquant que l'évaluation du **si** ... **alors** est plus coûteuse lorsque la condition est fausse (si elle est vraie, l'algorithme s'arrête), nous obtenons donc :

$$T(n) = T(n-1) + O(\log n) + O(\log((n-1)!) = T(n-1) + O(\log(n-1)!).$$

Constatons que nous avons exprimé le coût d'évaluation pour n en fonction du coût d'évaluation pour $n-1$, ce qui nous donne cette équation de récurrence. Contrairement aux définitions de suites par récurrence, nous n'avons pas de cas de base explicite. Puisque nous cherchons un comportement asymptotique (c'est-à-dire quand n tend vers $+\infty$), et que nous ignorons les facteurs multiplicatifs constants, la valeur précise de $T(0)$ ou $T(1)$ n'a pas d'importance.

Résolution des équations de récurrence. Il nous reste à résoudre cette équation. Ici nous pouvons simplifier en utilisant que $\log(a \times b) = \log a + \log b$ et la croissance du logarithme :

$$\log((n-1)!) = \sum_{i=1}^{n-1} \log i \leq (n-1)\log(n-1) = O(n \log n).$$

Ainsi :

$$T(n) = T(n-1) + O(n \log n).$$

Autrement dit, il existe une constante $c > 0$ et un entier $N \in \mathbb{N}$ tels que pour tout $n \geq N$, $T(n) \leq T(n-1) + cn \log n$. En dépliant la récurrence, nous avons

$$T(n) \leq cn \log n + c(n-1)\log(n-1) + \ldots + cN \log N + T(N-1),$$

et donc $T(n) \leq cn(n-N)\log n + T(N-1)$. Comme $T(N-1)$ est une constante, nous en déduisons que $T(n) = O(n^2 \log n)$.

Plus généralement, nous adopterons la méthode de résolution suivante. Dans un premier temps, deviner la complexité cherchée, par exemple en dépliant comme nous l'avons fait l'équation de récurrence, il est souvent possible de comprendre quelle va être la complexité, mais sans nécessairement obtenir une preuve correcte telle que celle que nous venons d'écrire. Parfois, l'expérience nous suffira pour deviner la complexité, mais il faut ensuite la justifier.

Illustrons avec notre exemple, supposons que nous avons deviné que la complexité de la factorielle allait être $O(n^2 \log n)$. Nous commençons par l'écrire plus précisément, suivant la définition de la domination, faisons l'hypothèse qu'il existe une constante $d > 0$, et un entier $N \geq 0$ tels que pour tout $n \geq N$, $T(n) \leq dn^2 \log n$. Nous prouvons notre hypothèse par récurrence, en prenant N =??? et d =??? (nous ne pouvons pas deviner par avance les valeurs de N et d, nous les déterminerons pendant l'écriture de la preuve).

Pour cas de base, c'est-à-dire $n = N$, nous avons $T(n) = T(N) \leq dN^2 \log N = dn^2 \log N$ par choix de d (nous voyons ici qu'il faudra choisir $d \geq T(N)/(N^2 \log N)$ et $N > 1$).

Pour le cas général, soit $n > N$ et supposons que $T(n-1) \leq d(n-1)^2 \log(n-1)$. Alors, par l'équation de récurrence, nous avons

$$\begin{aligned} T(n) &= T(n-1) + cn\log n \\ &\leq d(n-1)^2 \log(n-1) + cn\log n \\ &\leq d(n-1)^2 \log n + cn\log n \\ &\leq dn^2 \log n + (cn - 2dn + d)\log n \\ &\leq dn^2 \log n, \end{aligned}$$

car $cn < (2n-1)d$ puisque $d > c$ et $n > 1$ (nous obtenons ici une deuxième contrainte pour d).

En posant rétrospectivement $d \geq \max\{T(N)/(N^2 \log N), c+1\}$ et $N = 2$, nous en concluons que l'hypothèse est démontrée.

En pratique, un choix approprié pour d permet toujours d'assurer le cas de base, et nous prendrons donc l'habitude de ne pas le traiter pour se concentrer plutôt sur le cas général, en considérant que n est un entier arbitraire suffisamment grand. La preuve se fait alors en 4 étapes :

1. trouver l'équation récursive de complexité $T(n) = h(T(n-1), T(n-2), \ldots)$, $T(n)$ pouvant dépendre de n'importe quel $T(p)$ pour $p < n$,
2. conjecturer la complexité, sous la forme $T(n) = O(f(n))$,
3. préciser en introduisant la constante multiplicative c, c'est-à-dire $T(n) \leq cf(n)$,
4. montrer le cas général en substituant dans l'équation récursive chaque $T(p)$, $p < n$ par $cf(p)$ (en s'assurant que h est bien croissante).

Analyse des algorithmes diviser-pour-régner . Les algorithmes utilisant de la dichotomie ou plus généralement des techniques de diviser-pour-régner, ont des complexités dont l'équation de récurrence est

$$T(n) = kT(n/d) + O(f(n)),$$

avec k et d des entiers strictement positifs, et f une fonction positive arbitraire. Par exemple une recherche dichotomique fait $k = 1$ appels récursifs sur un espace de recherche de taille $n/2$, donc $d = 2$, et fait $O(1)$ opérations élémentaires hors appel récursif, donc $f : n \to 1$. Un algorithme de tri par fusion fait $k = 2$ appels récursifs chacun sur la moitié des données $d = 2$, et effectue une fusion en temps $O(n)$, donc $f : n \to n$.

Pour ces algorithmes, nous disposons du théorème suivant.

Théorème 8.11. Soit une suite $(T(n))_{n\in\mathbb{N}}$ définie par la relation de récurrence :

$$T(n) = kT\left(\frac{n}{d}\right) + f(n)$$

où $k \geq 1$, $d > 1$, et f est une fonction positive. Posons $\alpha = \log_d k$. Alors :

(i) S'il existe $\varepsilon > 0$ tel que $f(n) \leq O(n^{\alpha-\epsilon})$, alors $T(n) = \Theta(n^\alpha)$

(ii) Si $f(n) = \Theta(n^\alpha)$, alors $T(n) = \Theta(n^\alpha \log n)$

(iii) S'il existe $\varepsilon > 0$ tel que $n^{\alpha+\epsilon} = O(f)$ pour un $\epsilon > 0$, et si $kf(\frac{n}{d}) \leq cf(n)$ pour une constante $c < 1$ et n suffisamment grand, alors $T(n) = \Theta(f(n))$.

Démonstration. Considérons l'arbre des appels récursifs : sa racine est l'appel pour l'entrée de taille n. À chaque nœud, le problème de taille n est divisé en k sous-problèmes de taille $\frac{n}{d}$, qui forme les enfants du nœud. Les racines correspondent aux appels à l'algorithme qui sont résolus sans appels récursifs supplémentaires, c'est-à-dire aux cas de base de la récursion.

À une profondeur i (la racine est de profondeur 0), nous avons :

▷ un nombre de nœuds égal à k^i,

▷ chaque nœud correspond à un appel récursif avec une entrée de taille $\frac{n}{d^i}$,

▷ le coût d'évaluation de chaque nœud est au plus $f\left(\frac{n}{d^i}\right)$

La hauteur de l'arbre est $h = \lceil \log_d(n) \rceil$ puisque alors $d^h \geq n$ et donc $\frac{n}{d^h} \leq 1$, et le nombre de nœuds dans l'arbre est $\sum_{i=0}^{h} k^i \geq k^h \approx k^{\log_d n} = n^{\log_d k} = n^\alpha$.

Le coût d'évaluation de l'algorithme pour une entrée de taille n est alors

$$T(n) \leq \sum_{i=0}^{h} k^i f\left(\frac{n}{d^i}\right).$$

Si nous comparons entre deux niveaux successifs, le nombre de nœuds est multiplié par k, mais la taille de chaque nœud est divisé par d. Il y a une tension entre ces deux quantités : l'une tendant à augmenter les coûts et l'autre à les baisser. Le comportement général dépend alors de f.

(i) Premier cas : f n'est pas cher, la complexité est dominé par le nombre de nœuds (autrement dit par le coût des feuilles).

Puisque $f(n) = O(n^{\alpha-\epsilon})$, il existe des constantes c et n_0 telles que pour tout $n \geq n_0$:

$$f(n) \leq cn^{\alpha-\epsilon}.$$

Le coût au niveau i est alors majoré par :

$$\begin{aligned} k^i f\left(\frac{n}{d^i}\right) &\leq k^i c \left(\frac{n}{d^i}\right)^{\alpha-\epsilon} \\ &\leq c(d^\alpha)^i \left(\frac{n}{d^i}\right)^{\alpha-\epsilon} && (\text{car } k = d^\alpha) \\ &= cn^\alpha \left(\frac{1}{(nd)^\epsilon}\right)^i \end{aligned}$$

La somme totale peut donc être majorée ainsi :

$$T(n) \leq cn^\alpha \sum_{i=0}^{h} \left(\frac{1}{(nd)^\varepsilon}\right)^i$$

Nous remarquons que $\frac{1}{(nd)^\varepsilon} < 1$ et donc $\sum_{i=0}^{h} \left(\frac{1}{(nd)^\varepsilon}\right)^i$ est une série géométrique de raison inférieure à 1. Ainsi cette somme converge vers une constante, et $T(n) = O(n^\alpha)$. La minoration qui permet de conclure que $T(n) = \Theta(n^\alpha)$ consiste en remarquer que le nombre de nœuds de l'arbre est au moins n^α et que chaque nœud correspond à un appel récursif donc demande au moins quelques opérations élémentaires.

(ii) Deuxième cas : le coût de f s'équilibre avec le nombre de nœuds. Dans ce cas, nous montrons que tous les niveaux ont le même coût.

Par hypothèse, $f(n) = \Theta(n^\alpha)$, et donc il existe des constantes c_1, c_2 et n_0 telles que pour tout $n \geq n_0$:

$$c_1 n^\alpha \leq f(n) \leq c_2 n^\alpha$$

Le coût au niveau i est alors :

$$k^i f\left(\frac{n}{d^i}\right) \geq k^i c_1 \left(\frac{n}{d^i}\right)^\alpha = c_1 n^\alpha$$

car $k^i = (d^\alpha)^i = d^{i\alpha}$

De même, ce coût est majoré par $c_2 n^\alpha$. Donc à chaque niveau, le coût est $\Theta(n^\alpha)$.

Comme il y a $h + 1 = \log_d(n) + 1$ niveaux, la somme totale est :

$$T(n) = \Theta(n^\alpha \log n)$$

(iii) Troisième cas : le coût de f est plus élevé que le nombre de nœuds. Dans ce cas, nous montrons que l'évaluation de la racine domine le coût d'évaluation total.
Dans ce cas $f(n) \geq cn^{\alpha+\epsilon}$ pour une constante $c > 0$ et $kf(\frac{n}{d}) \leq c'f(n)$ pour une constante $c' < 1$.
Le coût au niveau i est majoré par :

$$k^i f\left(\frac{n}{d^i}\right) \leq (c')^i f(n).$$

La somme totale est donc majorée par :

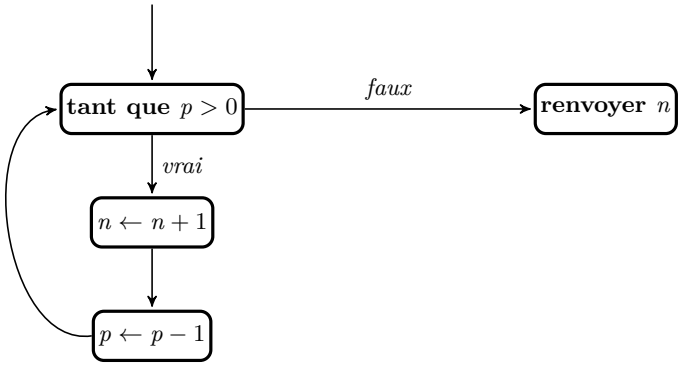
$$\begin{aligned} T(n) &\leq f(n) \sum_{i=0}^{h} (c')^i \\ &\leq f(n) \sum_{i=0}^{\infty} (c')^i \\ &= \frac{f(n)}{1-c'} = O(f(n)) \end{aligned}$$

La minoration par $\Omega(f(n))$ est immédiate car le premier niveau coûte déjà $f(n)$ opérations.

□

9 Correction des exercices

Exercice 1.1



Exercice 1.2 Pour simuler une conditionnelle **si** ... **alors** avec une boucle **tant que**, nous introduisons une variable supplémentaire, que nous appellerons *effectueInstructions*. L'idée est d'utiliser cette variable pour contrôler si les instructions du bloc doivent être effectuées ou non.

```
instructions_pre
effectueInstructions ← condition
tant que effectueInstructions faire
    instructions_corps
    effectueInstructions ← faux
instructions_post
```

Dès que les instructions sont effectuées, il ne faut pas les exécuter une deuxième fois, donc nous passons la variable à faux.

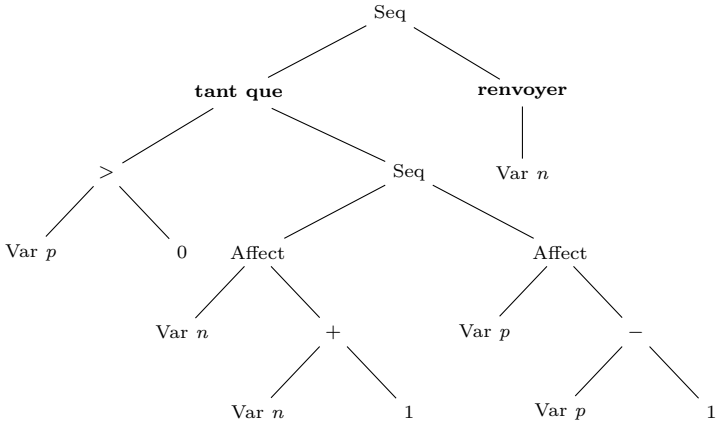
Pour la conditionnelle **si** ... **alors** ... **sinon**, il suffit d'utiliser une deuxième variable pour contrôler le deuxième bloc d'instructions.

```
instructions_pre
effectueInstructionsAlors ← condition
effectueInstructionsSinon ← non(effectueInstructionsAlors)
tant que effectueInstructionAlors faire
    instructions_corps
    effectueInstructionAlors ← faux
tant que effectueInstructionsSinon faire
    instructions_sinon
    effectueInstructionSinon ← faux
instructions_post
```

Exercice 1.3 La racine de l'arbre syntaxe abstrait est un nœud Seq, car le programme est une séquence d'instructions. Le premier enfant est un nœud **tant que**, avec deux enfants :

- ▷ la condition, formée de l'opérateur $>$ ayant pour opérandes la variable p et la constante 0 ;
- ▷ le corps de la boucle qui est lui-même un nœud Seq, groupant deux affectations : $n \leftarrow n + 1$ et $p \leftarrow p - 1$.

À l'issue de la boucle, le programme renvoie n, représenté par un nœud renvoyer, deuxième enfant de la racine.



Exercice 1.4 Voici le tableau de l'exécution du programme :

itération	0	1	2	$\cdots$	p-1	p
n	n	$n+1$	$n+2$	$\cdots$	$n+(p-1)$	$n+p$
p	p	$p-1$	$p-2$	$\cdots$	1	0
$p>0$	vrai	vrai	vrai	$\cdots$	vrai	faux

Nous en déduisons qu'il se produit p itérations, chacune effectuant deux instructions. Enfin une dernière instructions est exécutée pour retourner la valeur obtenue. Ainsi le nombre d'instructions total est $2p+1$.

Exercice 2.1 Nous vérifions les trois propriétés :

- ▷ (Réflexivité) Soit $n \in \mathbb{N}$. Alors $n = n+0$, donc en prenant $q = 0$, il existe $q \in \mathbb{N}$ tel que $n = n+q$, ce qui prouve que $n \leq n$.
- ▷ (Antisymétrie) Soient $n, p \in \mathbb{N}$ avec $n \leq p$ et $p \leq n$. Par définition de l'ordre, il existe q et q' entiers naturels tels que $p = n+q$ et $n = p+q'$. Alors $n = n+q+q'$ et donc $q+q' = 0$. Par positivité, $q = q' = 0$, et donc $n = p$.
- ▷ (Transitivité) Soient $n, p, q \in \mathbb{N}$, avec $n \leq p$ et $p \leq q$. Par définition de l'ordre, il existe r et r' tels que $p = n+r$ et $q = p+r'$, et donc $q = n+r+r'$. En prenant $s = r+r' \in \mathbb{N}$, $q = n+s$, et donc $n \leq q$.

Exercice 2.2 Nous vérifions les trois propriétés :

- ▷ (Réflexivité) Soit $n \in \mathbb{N}$. Alors $n \times 1 = n$, donc en prenant $q = 1$, il existe $q \in \mathbb{N}$ tel que $n \times q = n$, ce qui prouve que $n \mid n$.
- ▷ (Antisymétrie) Soient $n, p \in \mathbb{N}$ avec $n \mid p$ et $p \mid n$. Par définition de la relation, il existe q et q' entiers naturels tels que $p \times q = n$ et $n \times q' = p$. Alors $n \times q' \times q = n$ et donc $q \times q' = 1$. Ainsi $q = q' = 1$, et donc $n = p$.
- ▷ (Transitivité) Soient $n, p, q \in \mathbb{N}$, avec $n \mid p$ et $p \mid q$. Par définition de l'ordre, il existe r et r' tels que $n \times r = p$ et $p \times r' = q$, et donc $n \times r \times r' = q$. En prenant $s = r \times r' \in \mathbb{N}$, $n \times s = q$, et donc $n \mid q$.

Exercice 2.3 Notons que seule la définition de $\prec_{A\times B}$ est donnée, donc la définition de l'ordre strict. L'ordre $\preceq_{A\times B}$ s'obtient en rajoutant la réflexivité : $x \preceq_{A\times B} y$ si $x = y$ ou $x \prec_{A\times B} y$.

Ainsi, en présence d'une telle définition de la version stricte d'un ordre, la réflexivité de l'ordre non-strict est immédiate. L'antisymétrie se montre par contradiction : si $x \preceq y$, $y \preceq x$, par définition de $x \preceq y$ soit $x \neq y$ et c'est bon, soit $x \prec y$, et alors $y \prec x$ par définition de $y \preceq x$. Il suffit alors de démontrer que $x \prec y$ et $y \prec x$ entraînent une contradiction. Enfin pour l'antisymétrie, pour $x \preceq y$ et $y \preceq z$, si $x = y$ ou $y = z$, $x \preceq z$ s'obtient par une simple substitution. Il suffit donc de montrer que $x \prec y$ et $y \prec z$ implique $x \prec z$.

amU Faculté des sciences Aix Marseille Université

▷ (Antisymétrie) Soient (a_1, b_1) et (a_2, b_2) dans $A \times B$ tels que $(a_1, b_1) \prec_{A\times B} (a_2, b_2)$ et $(a_2, b_2) \prec_{A\times B} (a_1, b_1)$. Par définition de l'ordre, puisque $(a_1, b_1) \preceq_{A\times B} (a_2, b_2)$, $a_1 \prec_A a_2$ ou $a_1 = a_2$, c'est-à-dire $a_1 \preceq_A a_2$. De même, parce que $(a_2, b_2) \prec_{A\times B} (a_1, b_1)$, nous avons $a_2 \preceq_A a_1$. Nous en déduisons que $a_1 = a_2$. Reprenant la définition de l'ordre, appliquée à $(a_1, b_1) \prec_{A\times B} (a_2, b_2)$ nous obtenons $b_1 \prec_B b_2$, et appliquée à $(a_2, b_2) \prec_{A\times B} (a_1, b_1)$ nous obtenons $b_2 \prec_B b_1$. Ceci contredit le fait que $\preceq_B$ est un ordre.

▷ (Réflexivité) Soient (a_1, b_1), (a_2, b_2) et (a_3, b_3) dans $A \times B$ tels que $(a_1, b_1) \prec_{A\times B} (a_2, b_2)$ et $(a_2, b_2) \prec_{A\times B} (a_3, b_3)$. Si $a_1 \neq a_3$, par transitivité puisque $a_1 \preceq_A a_2 \preceq_A a_3$, $a_1 \prec_A a_3$ et donc $(a_1, b_1) \prec_{A\times B} (a_3, b_3)$. Sinon $a_1 = a_2 = a_3$, et par définition de $\prec_{A\times B}$, $b_1 \prec_B b_2$ et $b_2 \prec_b b_3$. Par transitivité de $\prec_B$, $b_1 \prec_B b_3$ et donc $(a_1, b_1) \prec_{A\times B} (a_3, b_3)$.

Exercice 2.4 Nous devons produire un variant bien-fondé pour la boucle. Nous remarquons que i diminue de 1 à chaque itération, et doit rester positif à cause de la condition $i > 1$. Nous pouvons donc poser comme variant V avec $V(s) = s(i) \in \mathbb{N}$, fonction à valeur dans l'ensemble bien fondé $(\mathbb{N}, \leq)$. Nous vérifions la décroissance : si

$$\langle s\rangle t[i] \leftrightarrow t[i-1]; i \leftarrow i - 1\langle s'\rangle,$$

alors $s'(i) = s(i) - 1 < s(i)$, donc $V(s') \leq V(s)$.

Exercice 3.1

(1) Les diviseurs de 60 sont $\{1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60\}$;

(2) nous pouvons aussi l'écrire $\{d \in \mathbb{N} \ : \ d \mid 60\}$ ou $\{d \in \mathbb{N} \ : \ \exists k \in \mathbb{N}, k \times d = 60\}$;

(3) les nombres premiers sont

$$\{p \in \mathbb{N} \ : \ \forall d \in \mathbb{N}, (d \mid p) \implies (d = 1 \vee d = p)\}$$

avec $(d \mid p) \equiv (\exists k \in \mathbb{N}, k \times d = p)$;

(4) les vecteurs réels de dimension 3 sont $\mathbb{R}^3$ ou encore $\mathbb{R} \times \mathbb{R} \times \mathbb{R}$.

Exercice 3.2

X_1 contient les axiomes, qui sont $\{0, 1\}$;

X_2 contient en plus les valeurs obtenues des axiomes par application d'une seule règle. La première fonction s'applique à 1 avec $x = \varepsilon$, et fournit 10. La deuxième fonction s'applique à 1 aussi avec $x = \varepsilon$, et fournit 11. Ainsi $X_1 = \{0, 1, 10, 11\}$;

X_3 contient en plus les valeurs obtenues à partir de X_1, par application d'une seule règle. 1 ne donnera pas de nouvelle valeur. Les deux règles s'appliquent par contre sur 10 en prenant $x = 0$ pour fournir 100 et 101, et sur 11 en prenant $x = 11$, pour fournir 110 et 111, donc $X_2 = x_1 \cup \{100, 101, 110, 111\}$;

X_4 contient encore en plus les valeurs obtenues depuis X_2, de la même manière. Ainsi 100 donne 1000 et 1001, 101 donne 1010 et 1011, 110 donne 1100 et 1101, et 111 donne 1110 et 1111. Au final

$$X_3 = X_2 \cup \{1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111\}.$$

Exercice 3.3 Chaque axiome et chaque fonction de la définition inductive donne une règle. Les axiomes n'ont pas de prémisse, et simplement la valeur en conclusion de la règle.

$$\frac{}{0} \qquad \frac{}{1}$$

Les fonctions donnent lieu à des règles dont la prémisse (partie haute) est l'argument et dont la conclusion (partie basse) est l'image de l'argument par la fonction.

$$\frac{1x}{1x0} \qquad \frac{1x}{1x1}$$

Exercice 3.4 Pour montrer qu'une définition inductive est ambiguë, il suffit de trouver une valeur ayant deux dérivations différentes. Ici nous pouvons dériver 10 de plusieurs façons.

$$\frac{\dfrac{0}{5}}{10} \qquad \frac{\dfrac{0}{7}}{10} \qquad \frac{\dfrac{0}{3}}{10}$$

Exercice 4.1 Nous commençons par repérer les formules élémentaires ou formules atomiques, c'est-à-dire les formules ne pouvant pas être décomposées en assemblage de formules plus petites. Ici ce sont les littéraux vrai et faux, et les variables x, y, z. Ces formules atomiques sont les axiomes de la définition inductive. Nous pouvons introduire l'ensemble X des variables disponibles, par exemple $X = \{x, y, z\}$ ou $X = \Sigma^\star \setminus \{\varepsilon\}$ avec $\Sigma = \{a, b, \ldots, z\}$ si nous voulons un ensemble plus grand, et poser comme axiome que tout élément de X est une formule. Cela nous donne :

$$\frac{}{\text{vrai}} \qquad \frac{}{\text{faux}} \qquad \frac{}{x}\ x \in X$$

Cela correspond à dire qu'il y a $|X| + 2$ axiomes, le 3e correspondant à une famille d'un axiome par élément de X.

Il faut ensuite repérer les constructions permettant de construire des formules à partir de formules plus simples. Ici ce sont les opérateurs $\wedge$, $\vee$, $\implies$, $\neg$ et les quantificateurs $\exists$ et $\forall$. Les opérateurs binaires $\wedge$, $\vee$, $\implies$ nécessitent deux prémisses, puisqu'ils se construisent avec deux sous-formules.

$$\frac{\phi \qquad \phi}{\phi \wedge \psi} \qquad \frac{\phi \qquad \phi}{\phi \vee \psi} \qquad \frac{\phi \qquad \phi}{\phi \implies \psi}$$

L'opérateur $\neg$ est unaire, la négation se construit sur une seule formule, la règle n'a donc qu'une prémisse.

$$\frac{\phi}{\neg\phi}$$

Enfin les quantificateurs sont aussi unaires, mais dépendent aussi du choix d'une variable. Ce choix est mentionné dans l'étiquette à droite de la règle, ce qui revient à dire qu'il y a une infinité de règle (si X est infini) : une pour chaque variable disponible.

$$\frac{\phi}{\exists x.\phi}\ x \in X \qquad \frac{\phi}{\forall x.\phi}\ x \in X$$

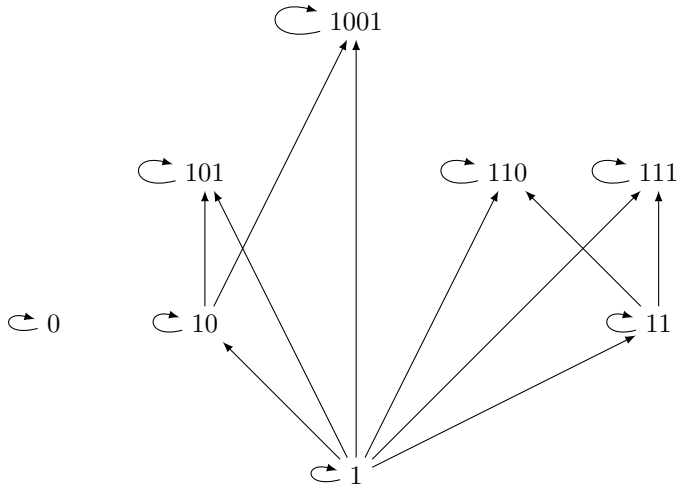
Exercice 5.1 L'ordre $\preceq$ induit par une définition inductive non-ambiguë est : $x \preceq y$ si x apparaît dans l'arbre de dérivation de y. Nous pouvons commencer par donner les arbres de dérivations des valeurs proposées.

$$\frac{}{0} \qquad \frac{\dfrac{\dfrac{}{1}}{10}}{101} \qquad \frac{\dfrac{\dfrac{\dfrac{}{1}}{10}}{100}}{1001} \qquad \frac{\dfrac{\dfrac{}{1}}{11}}{110} \qquad \frac{\dfrac{\dfrac{}{1}}{11}}{111}$$

Nous obtenons donc :

$$1 \preceq 10 \preceq 101$$
$$1 \preceq 10 \preceq 100 \preceq 1001$$
$$1 \preceq 11 \preceq 110$$
$$1 \preceq 11 \preceq 111$$

que nous pouvons représenter par un graphe orienté :



Exercice 5.2 Nous souhaitons écrire une définition inductive de la fonction qui double un entier. Définir une fonction inductivement requière de poser une valeur pour chaque règle de la définition inductive. La première règle dans la définition inductive des entiers naturels est celle posant 0 comme axiome :

$$\frac{}{0}$$

Nous savons que le double de 0 est 0, donc nous posons simplement

$$\frac{}{\text{double}(0) : 0}$$

La deuxième règle est celle du successeur :

$$\frac{n}{S(n)}$$

Inductivement nous pouvons définir le double de $S(n)$ en fonction du double de n. Nous savons que $2(n+1) = 2n+2$, donc le double de $S(n)$ est 2 plus le double de n. Nous en déduisons la règle suivante.

$$\frac{\text{double}(n) : p}{\text{double}(S(n)) : S(S(p))}$$

qui se lit : si p est le double de n alors $S(S(p))$ est le double de $S(n)$.

Exercice 5.3 Nous procédons de la même manière, en donnant pour chaque règle de la déifnition inductive de la représentation binaire des entiers, une règle pour déterminer sa valeur. La première règle est l'axiome 0, dont nous savons que la valeur est 0.

$$\frac{}{0} \qquad \frac{}{\text{valeur}(0) : 0}$$

La deuxième règle est l'axiome 1, dont nous savons qu'il représente la valeur 1.

$$\frac{}{1} \qquad \frac{}{\text{valeur}(1) : 1}$$

La troisième règle permet d'ajouter un 0 en fin d'écriture binaire d'un nombre pourvu qu'elle commence par un 1. Comme préciser dans l'énoncé, cela a pour effet de doubler la valeur du nombre représenté.

$$\frac{1x}{1x0} \qquad \frac{\text{valeur}(1x) : n}{\text{valeur}(1x0) : 2n}$$

La dernière règle permet d'ajouter un 1 en fin d'écriture binaire d'un nombre pourvu qu'elle commence par un 1. Cela a pour effet de doubler la valeur du nombre représenté et de lui ajouter un.

$$\frac{1x}{1x1} \qquad \frac{\text{valeur}(1x) : n}{\text{valeur}(1x1) : 2n+1}$$

Exercice 5.4 Les mots se construisent inductivement à partir du mot vide en ajoutant des lettres, par exemple à la fin. Cela nous donne cette définition pour l'alphabet $\{a, b, c\}$.

$$\frac{}{\varepsilon} \qquad \frac{w}{wa} \qquad \frac{w}{wb} \qquad \frac{w}{wc}$$

Nous pouvons ensuite donner une définition inductive de la fonction de substitution, en détailant la fonction pour chacune des quatre règle s.

$$\frac{}{f(\varepsilon) : \varepsilon} \qquad \frac{f(w) : w'}{f(wa) : w'b} \qquad \frac{f(w) : w'}{f(wb) : w'a} \qquad \frac{f(w) : w'}{f(wc) : w'c}$$

Dans chacun des trois cas d'induction, w' représente l'image de w par la fonction de substitution f, et il nous reste à appliquer la substitution sur la dernière lettre du mot.

Exercice 5.5 Rappelons d'abord les définitions. Les représentations en binaire sont données par

$$\frac{}{0} \qquad \frac{}{1} \qquad \frac{1x}{1x0} \qquad \frac{1x}{1x1}$$

et la valeur associée est définie comme

$$\frac{}{\text{Valeur}(0) : 0} \qquad \frac{}{\text{Valeur}(1) : 1} \qquad \frac{\text{Valeur}(1x) : n}{\text{Valeur}(1x0) : 2n} \qquad \frac{\text{Valeur}(1x) : n}{\text{Valeur}(1x1) : 2n+1}$$

Nous nous souvenons aussi que la représentation binaire $x_k x_{k-1} \dots x_0$ représente l'entier $\sum_{i=0}^{k} x_i 2^i$, chaque x_i étant un bit 0 ou 1. Nous vérifions par induction sur toute représentation binaire w que valeur(w) est bien la valeur représentée par w. Suivant la définition inductive des représentations binaires, nous avons 4 cas à traiter.

- ▷ Cas $w = 0$, par définition valeur$(0) = 0$, ce qui est correct.
- ▷ Cas $w = 1$, par définition valeur$(1) = 1$, ce qui est correct aussi.
- ▷ Cas $w = 1x0$, alors

  $$\text{valeur}(w) = \text{valeur}(1x0) = 2 \times \text{valeur}(1x)$$

  par définition de la fonction valeur. En appliquant l'hypothèse d'induction à $1x$, nous avons que

  $$\text{valeur}(1x) = \sum_{i=0}^{k} (1x)_i \times 2^i,$$

  où $(1x)_i$ représente le i^{e} bit du $1x$ en partant de la droite, et k est le nombre de bits de $1x$. Mais alors

  $$\begin{aligned}
  \sum_{i=0}^{k+1} (1x0)_i \times 2^i &= 0 \times 2^0 + \sum_{i=1}^{k+1} (1x0)_i \times 2^i && \text{(en isolant le cas } i = 0) \\
  &= \sum_{i=0}^{k} (1x0)_{i+1} \times 2^{i+1} && \text{(par changement de variable } i \to i+1) \\
  &= 2 \sum_{i=0}^{k} (1x)_i \times 2^i && \text{(car } 2^{i+1} = 2 \times 2^i) \\
  &= 2 \times \text{valeur}(1x) \\
  &= \text{valeur}(1x0).
  \end{aligned}$$

- ▷ Cas $w = 1x0$, le raisonnement est similaire. Nous avons

  $$\text{valeur}(w) = \text{valeur}(1x1) = 2 \times \text{valeur}(1x) + 1$$

par définition de la fonction valeur. En appliquant l'hypothèse d'induction à $1x$, nous avons que

$$\text{valeur}(1x) = \sum_{i=0}^{k} (1x)_i 2^i,$$

où $(1x)_i$ représente le i^{e} bit du $1x$ en partant de la droite, et k est le nombre de bits de $1x$. Mais alors

$$\begin{aligned}
\sum_{i=0}^{k+1} (1x1)_i 2^i &= 1 \times 2^0 + \sum_{i=1}^{k+1} (1x0)_i \times 2^i && \text{(en isolant le cas } i = 0) \\
&= 1 + \sum_{i=0}^{k} (1x0)_{i+1} \times 2^{i+1} && \text{(par changement de variable } i \to i+1) \\
&= 1 + 2 \sum_{i=0}^{k} (1x)_i \times 2^i && \text{(car } 2^{i+1} = 2 \times 2^i) \\
&= 1 + 2 \times \text{valeur}(1x) \\
&= \text{valeur}(1x1).
\end{aligned}$$

Exercice 5.6 Nous définissons l'incrément en donnant une règle pour chaque règle de la définition inductives des représentations binaires d'entiers.

$$\frac{}{\text{incr}(0) : 1} \qquad \frac{}{\text{incr}(1) : 10} \qquad \frac{\text{incr}(1x) : y}{\text{incr}(1x0) : 1x1} \qquad \frac{\text{incr}(1x) : y}{\text{incr}(1x1) : y0}$$

Nous vérifions par induction sur toute représentation binaire w que $\text{valeur}(\text{incr}(w)) = 1 + \text{valeur}(w)$.

▷ Cas $w = 0$, alors

$$\begin{aligned}
\text{valeur}(\text{incr}(w)) &= \text{valeur}(1) && \text{(par définition de incr)} \\
&= 1 && \text{(par définition de valeur)} \\
&= 1 + 0 && \text{(par les lois de l'arithmétique)} \\
&= 1 + \text{valeur}(0) && \text{(par définition de valeur)} \\
&= 1 + \text{valeur}(w)
\end{aligned}$$

▷ Cas $w = 1$, alors

$$\begin{aligned}
\text{valeur}(\text{incr}(w)) &= \text{valeur}(10) && \text{(par définition de incr)} \\
&= 2 \times \text{valeur}(1) && \text{(par définition de valeur)} \\
&= 2 && \text{(par définition de valeur)} \\
&= 1 + 1 && \text{(par les lois de l'arithmétique)} \\
&= 1 + \text{valeur}(1) && \text{(par définition de valeur)} \\
&= 1 + \text{valeur}(w)
\end{aligned}$$

▷ Cas $w = 1x0$, alors

$$\begin{aligned}
\text{valeur}(\text{incr}(w)) &= \text{valeur}(\text{incr}(1x0)) \\
&= \text{valeur}(1x1) && \text{(par définition de incr)} \\
&= 1 + 2 \times \text{valeur}(1x) && \text{(par définition de valeur)} \\
&= 1 + \text{valeur}(1x0) && \text{(par définition de valeur)} \\
&= 1 + \text{valeur}(w)
\end{aligned}$$

▷ Cas $w = 1x0$, alors posons $y = \text{incr}(1x)$, nous avons

$$\begin{aligned}
\text{valeur}(\text{incr}(w)) &= \text{valeur}(\text{incr}(1x1)) \\
&= \text{valeur}(y0) && \text{(par définition de incr)} \\
&= 2 \times \text{valeur}(y) && \text{(par définition de valeur)} \\
&= 2 \times \text{valeur}(\text{incr}(1x)) && \text{(par définition de } y\text{)} \\
&= 2 \times (1 + \text{valeur}(1x)) && \text{(par hypothèse d'induction appliquée à } 1x\text{)} \\
&= 1 + (1 + 2 \times \text{valeur}(1x)) && \text{(par les lois de l'arithmétique)} \\
&= 1 + \text{valeur}(1x1) && \text{(par définition de valeur)} \\
&= 1 + \text{valeur}(w)
\end{aligned}$$

Exercice 6.1 Nous appliquons la définition de la sémantique d'une expression.

1. $$\dfrac{\dfrac{[\![n]\!]_{[n\mapsto 3, m\mapsto 5]} : 3 \qquad [\![3]\!]_{[n\mapsto 3, m\mapsto 5]} : 3}{[\![n+3]\!]_{[n\mapsto 3, m\mapsto 5]} : 6} \qquad [\![m]\!]_{[n\mapsto 3, m\mapsto 5]} : 5}{[\![(n+3)\times m]\!]_{[n\mapsto 3, m\mapsto 5]} : 30}$$

2. $$\dfrac{\dfrac{[\![n]\!]_{[n\mapsto 2, m\mapsto -3]} : 2 \qquad [\![m]\!]_{[n\mapsto 2, m\mapsto -3]} : -3}{[\![n+m]\!]_{[n\mapsto 2, m\mapsto -3]} : -1} \qquad \dfrac{[\![n]\!]_{[n\mapsto 2, m\mapsto 5]} : 2 \qquad [\![m]\!]_{[n\mapsto 2, m\mapsto -3]} : -3}{[\![n-m]\!]_{[n\mapsto 2, m\mapsto -3]} : 5}}{[\![(n+m)\times(n-m)]\!]_{[n\mapsto 2, m\mapsto -3]} : -5}$$

3. Notons $e = [m \mapsto 25, n \mapsto 16]$, alors
$$\dfrac{\dfrac{\dfrac{[\![n]\!]_e : 16 \qquad [\![2]\!]_e : 2}{[\![n/2]\!]_e : 8} \qquad \dfrac{[\![m]\!]_e : 25 \qquad [\![5]\!]_e : 5}{[\![m/5]\!]_e : 5}}{[\![(n/2)-(m/5)]\!]_e : 3} \qquad [\![3]\!]_e : 3}{[\![((n/2)-(m/5))/3]\!]_e : 1}$$

4. $$\dfrac{\dfrac{[\![m]\!]_{[n\mapsto 8, m\mapsto 1]} : 1 \qquad [\![n]\!]_{[n\mapsto 8, m\mapsto 1]} : 8}{[\![m-n]\!]_{[n\mapsto 8, m\mapsto 1]} : -7} \qquad \dfrac{[\![m]\!]_{[n\mapsto 8, m\mapsto 1]} : 1 \qquad [\![n]\!]_{[n\mapsto 8, m\mapsto 1]} : 8}{[\![m+n]\!]_{[n\mapsto 8, m\mapsto 1]} : 9}}{[\![(m-n)+(m+n)]\!]_{[n\mapsto 8, m\mapsto 1]} : 2}$$

Exercice 6.2 Notons e_0 l'état initial $[m \mapsto 5, n \mapsto 3]$, et calculons l'état final du programme p :

1: $s \leftarrow 0$
2: **tant que** $n \neq 0$ **faire**
3: $\quad n \leftarrow n - 1$
4: $\quad s \leftarrow s + m$

en appliquant les règles de la sémantique du langage. Notons aussi $q = n \leftarrow n - 1; s \leftarrow s + m$ le bloc d'instructions de la boucle.

$$\dfrac{\langle e_0\rangle s \leftarrow 0 \langle e_1\rangle \qquad \dfrac{\pi_1 \qquad \dfrac{\pi_2 \qquad \dfrac{\pi_3 \qquad \overline{\langle e_7\rangle \textbf{tant que } n \geq 0 \textbf{ faire } q \langle e_7\rangle}}{\langle e_5\rangle \textbf{tant que } n \geq 0 \textbf{ faire } q \langle e_7\rangle}}{\langle e_3\rangle \textbf{tant que } n \geq 0 \textbf{ faire } q \langle e_7\rangle}}{\langle e_1\rangle \textbf{tant que } n \geq 0 \textbf{ faire } q \langle e_7\rangle}}{\langle e_0\rangle p \langle e_7\rangle}$$

où

$$\pi_1 = \frac{\overline{\langle e_1\rangle n \leftarrow n-1\langle e_2\rangle} \quad \overline{\langle e_2\rangle s \leftarrow s+m\langle e_3\rangle}}{\langle e_1\rangle n \leftarrow n-1; s \leftarrow s+m\langle e_3\rangle} \qquad \pi_2 = \frac{\overline{\langle e_3\rangle n \leftarrow n-1\langle e_4\rangle} \quad \overline{\langle e_4\rangle s \leftarrow s+m\langle e_5\rangle}}{\langle e_3\rangle n \leftarrow n-1; s \leftarrow s+m\langle e_5\rangle}$$

$$\pi_3 = \frac{\overline{\langle e_5\rangle n \leftarrow n-1\langle e_6\rangle} \quad \overline{\langle e_6\rangle s \leftarrow s+m\langle e_7\rangle}}{\langle e_5\rangle n \leftarrow n-1; s \leftarrow s+m\langle e_7\rangle}$$

avec

$$\begin{aligned}
e_1 &= e_0[0 \leftarrow s] = [m \mapsto 5, n \mapsto 3, s \mapsto 0]\\
e_2 &= e_1[[\![n-1]\!]_{e_1} \leftarrow n] = e_1[2 \leftarrow n] = [m \mapsto 5, n \mapsto 2, s \mapsto 0]\\
e_3 &= e_2[[\![s+m]\!]_{e_2} \leftarrow s] = e_2[5 \leftarrow s] = [m \mapsto 5, n \mapsto 2, s \mapsto 5]\\
e_4 &= e_3[[\![n-1]\!]_{e_3} \leftarrow n] = e_3[2 \leftarrow n] = [m \mapsto 5, n \mapsto 1, s \mapsto 5]\\
e_5 &= e_4[[\![s+m]\!]_{e_4} \leftarrow s] = e_4[5 \leftarrow s] = [m \mapsto 5, n \mapsto 1, s \mapsto 10]\\
e_6 &= e_5[[\![n-1]\!]_{e_5} \leftarrow n] = e_5[2 \leftarrow n] = [m \mapsto 5, n \mapsto 0, s \mapsto 10]\\
e_7 &= e_6[[\![s+m]\!]_{e_6} \leftarrow s] = e_6[5 \leftarrow s] = [m \mapsto 5, n \mapsto 0, s \mapsto 15].
\end{aligned}$$

L'état final obtenu est alors e_7.

Exercice 6.3 Nous donnons une règle pour chaque branche de la conditionnelle. La première règle définit le comportement du programme lorsque la condition est vérifiée.

$$\frac{\langle e\rangle p\langle e'\rangle}{\langle e\rangle \textbf{si } \phi \textbf{ alors } p \textbf{ sinon } q\langle e'\rangle}\ [\![\phi]\!]_e = \text{vrai}$$

Dans ce cas l'état est modifié par le bloc de la branche **alors**. Sinon, il est modifié par le bloc de la branche **sinon** :

$$\frac{\langle e\rangle q\langle e'\rangle}{\langle e\rangle \textbf{si } \phi \textbf{ alors } p \textbf{ sinon } q\langle e'\rangle}\ [\![\phi]\!]_e = \text{faux}$$

Dans le cas d'une conditionnelle sans branche **sinon**, les deux règles deviennent :

$$\frac{\langle e\rangle p\langle e'\rangle}{\langle e\rangle \textbf{si } \phi \textbf{ alors } p\langle e'\rangle}\ [\![\phi]\!]_e = \text{vrai} \qquad \text{et} \qquad \frac{}{\langle e\rangle \textbf{si } \phi \textbf{ alors } p\langle e2\rangle}\ [\![\phi]\!]_e = \text{faux}$$

Exercice 7.1 Nous voulons toujours écrire $n = qm + r$ avec $0 \leq r < |m|$ mais cette fois q peut être négatif. Pour cela nous augmentons n s'il est négatif, et nous le diminuons s'il est supérieur ou égal à m. Pour augmenter n il faut ajouter m si m est positif, le soustraire sinon, et inversement pour diminuer n.

Entrée: n, un entier relatif, m, un entier relatif non nul.
Sortie: le résultat de la division entière de n par m

1: **soit** *quotient* $= 0$
2: **tant que** $(m > 0 \wedge n < 0) \vee (m < 0 \wedge n > -m)$ **faire**
3: $\quad$ *quotient* $\leftarrow$ *quotient* $- 1$
4: $\quad$ $n \leftarrow n + m$
5: **tant que** $(m > 0 \wedge n \geq m) \vee (m < 0 \wedge n < 0)$ **faire**
6: $\quad$ *quotient* $\leftarrow$ *quotient* $+ 1$
7: $\quad$ $n \leftarrow n - m$
8: **renvoyer** *quotient*

Posons $V : \mathbb{Z} \to (\mathbb{N}, \leq)$ avec

$$V(n) = \begin{cases} 0 & \text{si } 0 \leq n < |m|,\\ -n & \text{si } n < 0,\\ n - |m| + 1 & \text{si } n \geq |m|. \end{cases}$$

V est un variant dans un ordre bien fondé pour chacune des deux boucles, montrant que l'algorithme termine.

amU Faculté des sciences Aix Marseille Université