Spécification et validation du logiciel

Jean Goubault-Larrecq

February 12, 1997

Résumé

Le but de ce cours est de présenter quelques-unes des méthodes utilisées aujourd'hui pour spécifier formellement des projets informatiques, et vérifier automatiquement ou semi-automatiquement leur mise en œuvre. Nous étudierons en particulier le langage et la méthode Z, utilisés surtout pour décrire et vérifier formellement des logiciels critiques. Les concepts de base sont la notion de spécification, la notion de raffinement d'une spécification en une autre, et la notion de preuve de correction d'un raffinement.

Table des matières

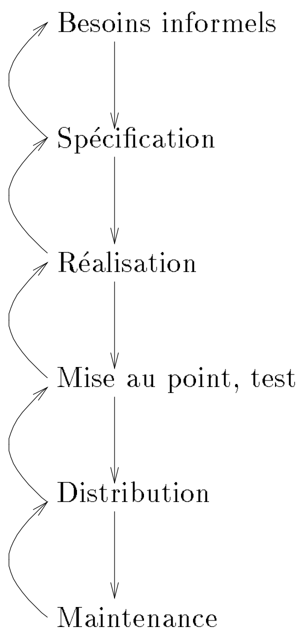

FIG. 1: Cycle de vie

1 Introduction

Il existe de nombreuses façons de développer un projet logiciel. La plus simple, mais aussi la plus artisanale, est celle qui consiste à programmer directement, en espérant que le code produit fera ce que l'on souhaite. Comme en général le programme ne fonctionne pas comme il est souhaité du premier coup, il est nécessaire de le mettre au point, en lui faisant passer un certain nombre de tests, et en corrigeant — en déboguant — le programme.

La mise au point de programmes est sans doute l'une des activités les plus coûteuses, et aussi l'une de celles dont le coût est le plus sous-estimé. Bien qu'un programmeur puisse écrire plusieurs centaines de lignes de programme par jour en moyenne, certains experts ont estimé que la productivité réelle, en tenant compte des journées passées à la mise au point ou à d'autres activités annexes, serait de 5 à 20 lignes de code par jour. Sachant que le nombre de jours travaillés par an tourne autour de 200, ceci revient à dire qu'un programme d'un million de lignes (un chiffre relativement typique des projets industriels) demande au minimum 250 hommes ans, et jusqu'à 1000 hommes ans, en supposant que le projet ne dérape pas. Le temps passé à réellement écrire les lignes de programme qui feront partie du produit final est dans le meilleur des cas de 20%, mais le plus souvent est inférieur à 5%. Le reste du temps est passé non seulement à la mise au point des programmes, mais aussi et surtout en discussions entre membres de l'équipe de développement pour savoir quelles sont les fonctionnalités à fournir, et quelles fonctionnalités existantes peuvent être utilisées.

Ces estimations sont, malgré les apparences, encore optimistes. Car la vie de logiciel ne s'arrête pas à sa date de première distribution: il sera encore nécessaire de l'adapter aux besoins des clients, de l'améliorer, en un mot d'effectuer sa maintenance. On résume en général la vie d'un logiciel par son cycle de vie (cf. Figure 1). Idéalement, la vie d'un projet commence par l'exposé des besoins auxquels il est censé répondre, puis par l'établissement d'un cahier des charges, c'est-à-dire d'une spécification des fonctionnalités du logiciel. Cette spécification doit ensuite guider la réalisation du programme, puis la mise au point (le programme fait-il ce qu'on attend de lui?), et la maintenance (que fait le programme? comment l'améliorer?). En général, cependant, il est nécessaire d'effectuer des retours en arrière. Un cas typique et très coûteux en général est celui de l'*erreur de conception*, où l'on s'aperçoit lors de la phase de test, ou pire après distribution du produit, que l'algorithme qui a été codé ne répond pas à la spécification. Il est alors nécessaire de recoder un nouvel algorithme. En fait, le coût du retour à une phase antérieure est d'autant plus énorme que l'erreur est constatée le plus tard, et affecte une décision prise le plus tôt.

Pour diminuer au maximum les coûts associés à la mise au point, à la maintenance et surtout aux retours à des phases antérieures de conception, il est donc nécessaire de faire de la spécification un document de référence, consultable par tous les développeurs, et qui décrive avec le moins d'ambiguïté possible ce que l'on attend du logiciel.

Si l'on souhaite une sécurité absolue — dans le cas de logiciels de commande d'aiguillages, par exemple, où toute erreur peut se solder par des morts d'homme —, il est alors nécessaire, premièrement de décrire cette spécification dans un langage formel, c'est-à-dire dont l'interprétation, contrairement au langage naturel, ne souffre aucune interprétation subjective. Et secondement, de disposer de méthodes et d'outils pour vérifier que les programmes réalisés effectuent réellement ce qui est décrit par la spécification. Les mathématiques fournissent le meilleur cadre possible pour décrire des spécifications (sous forme de propositions logiques), et pour vérifier qu'un programme réalise une spécification (vérifier que la relation définie par le programme entre ses états d'entrée et de sortie vérifie la proposition mathématique qu'est la spécification). C'est pourquoi les diverses méthodes de spécification et de vérification, dites *méthodes formelles*, sont fondées sur des formalismes mathématiques.

En général, l'objectif n'est pas d'atteindre l'absence totale de défauts dans le programme. En effet, l'activité de spécification et de vérification est elle-même extrêmement coûteuse. Si l'on prend les méthodes formelles à la lettre, le programme sera absolument sûr — en tous cas, beaucoup plus sûr que si seuls des tests ont été effectués —, mais le coût pourra être aussi élevé, sinon plus élevé qu'un développement classique. Dans ce cas, l'établissement de spécifications formelles peut quand même aider, non à s'assurer de l'absence d'erreurs, mais à augmenter la confiance qu'un utilisateur peut placer dans le logiciel. En fait, le simple fait d'écrire une spécification formelle d'un logiciel permet usuellement d'éclaircir les idées des concepteurs, et ainsi d'éviter des erreurs de conception ultérieures qui seraient à des points laissés dans l'ombre.

L'établissement d'une spécification peut aussi être utile à d'autres activités. Citons notamment la génération automatique, ou semi-automatique de tests: en effet, il n'est pas rare que les tests de logiciels oublient certaines conditions d'utilisation aux limites auxquelles les concepteurs n'ont pas pensé, alors qu'il est possible d'extraire ces conditions de la spécification automatiquement.

Le plan du cours est le suivant. En section 2, nous présentons les principes de base des langages de spécification formels, et illustrons ces principes par l'exemple du langage Z [Spi88, Spi89]. Nous montrerons en section 3 comment utiliser une telle méthode pour écrire des spécifications, et les raffiner petit à petit jusqu'à aboutir à un programme exécutable. Pour vérifier chaque étape de raffinement, il faudra que l'on prouve qu'elle est correcte: ceci demande à savoir exactement ce qu'est une preuve formelle (une démonstration), ce que nous verrons en section 4. Nous concluons ce cours en section 5. Les annexes ne font pas partie du cours, mais serviront de référence pour définir notre version simplifiée de Z: syntaxe en section A, sémantique en section B, règles de preuve et axiomes de Z en section C.

<u>En résumé:</u>

- Les coûts d'une erreur sont d'autant plus élevés qu'elle a été commise tôt et détectée tard.
- Une *spécification* est un document de référence établi le plus tôt possible et servant de référence pour toute la vie d'un logiciel.
- Une spécification *formelle*, fondée sur un langage mathématique sans ambiguïté, laisse la place à un minimum d'erreurs possibles.

▸ EXERCICE 1.1

Une société d'assurances développe une base de données des souscripteurs de son assurance automobile. Le groupe de développement s'est réuni pour écrire une spécification, qui est un document en français. Les annexes de ce document décrivent les formats des enregistrements, en C++, langage choisi pour le développement. (Il est entendu que les spécificateurs ont écrit les fichiers d'en-tête [.h] d'abord, pour servir de référence lors de l'écriture du code proprement dit.) Un de ces fichiers, `client.h`, décrit un enregistrement client comme suit (nous avons simplifié intentionnellement):

```
struct MAPG_client {
  char mapgc_num[24]; // numero de contrat
  char mapgc_nom[80]; // nom du client
  char mapgc_adresse[200]; // son adresse principale
  // Pour l'Europe seulement:
  char mapgc_ape[6]; // code APE de l'entreprise
  MAPG_info *mapgc_info;
  // liste des autres personnes autorisees a conduire le vehicule
  };
```

Le groupe de développement se met ensuite à coder. La partie de saisie des informations clients revient au développeur A, et B écrit une partie d'un système expert exploitant les informations de la base de données clients. Un an plus tard, le logiciel est installé dans tous les bureaux européens de la compagnie, et fonctionne bien. Deux ans plus tard, il est décidé de le déployer dans le reste du monde. Le jour du déploiement international, de nombreux plantages du système expert sont signalés, stoppant l'activité contentieux pour deux jours, le temps de revenir à l'ancien système installé. Après un mois d'analyses, il s'avère que A avait compris qu'il ne fallait remplir le champ `mapgc_info` que pour les sociétaires vivant en Europe, alors que B pensait qu'il était toujours rempli, et que le commentaire `pour l'Europe seulement` ne s'appliquait qu'au champ `mapgc_ape` (le code APE est effectivement une codification purement européenne): pour les sociétaires non européens, le champ `mapgc_info` était un pointeur arbitraire, et sa manipulation par le système expert provoquait des plantages.

D'après vous, d'où vient le problème? Comment aurait-on pu l'éviter?

▸ EXERCICE 1.2

Une compagnie d'ascenseurs utilise la spécification suivante pour ses installations de n ascenseurs desservant m étages (cf. figure 2):

1. Tout ascenseur a un bouton par étage. Ils s'allument lorsqu'on les presse, et forcent l'ascenseur à aller à l'étage correspondant. Lorsque l'ascenseur s'arrête à l'étage, le bouton s'éteint.

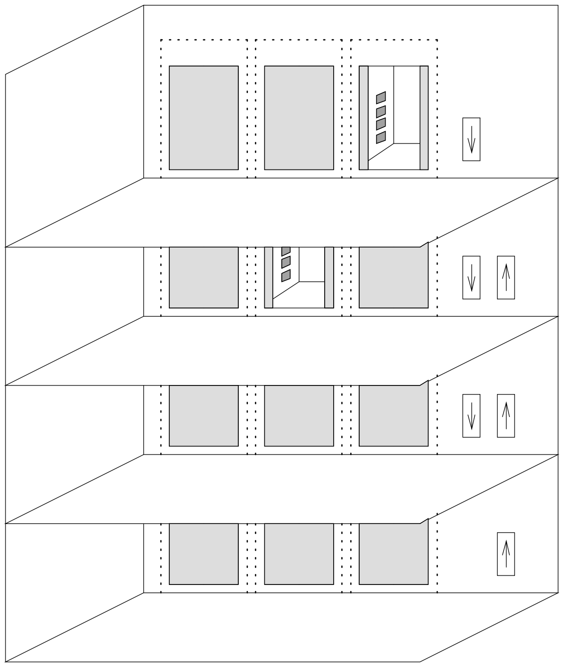

FIG. 2: Un système d'ascenseurs

2. Chaque étage a deux boutons, ↑ (sauf pour le dernier étage, demande à monter) et ↓ (sauf rez-de-chaussée, demande à descendre). Ils s'allument lorsqu'on les presse. Les boutons s'éteignent si un ascenseur s'arrête à l'étage qui voyage dans la direction désirée ou qui n'a aucun bouton allumé. Dans ce dernier cas, si les deux boutons de l'étage sont allumés, un seul devrait s'éteindre. L'algorithme utilisé doit alors minimiser le temps d'attente pour les requêtes de montée et de descente.

3. Si un ascenseur n'a aucun bouton allumé, il doit rester immobile, portes fermées et attendre qu'on presse un bouton.

4. Toutes les requêtes de montée (↑) et de descente (↓) doivent être honorées en temps fini, et équitablement entre étages.

5. Toutes les requêtes d'étage (appuis de boutons d'ascenseurs) doivent être honorées en temps fini, séquentiellement dans l'ordre de déplacement.

Dans le dernier modèle de la compagnie, l'algorithme implémenté dans les ascenseurs est une boucle dans laquelle sont réalisés les différents points ci-dessus. En particulier, le point 3 est réalisé comme suit: si aucun bouton de l'ascenseur n'est allumé, l'ascenseur est arrêté (s'il ne l'est pas déjà), ses portes se ferment (si elles étaient ouvertes), et l'algorithme revient au début de la boucle pour attendre. Une maquette est réalisée, et testée: initialement, tous les ascenseurs sont vides, portes fermées et au rez-de-chaussée. Que se passe-t-il? Comment corriger le problème?

▸ EXERCICE 1.3

Considérons le problème de l'exercice 1.2 résolu. Maintenant, les ascenseurs se déplacent aux étages demandés, mais leurs portes ne s'ouvrent pas. L'équipe de production rejette la faute sur le développeur de l'algorithme, mais ce dernier estime qu'il n'a fait qu'obéir à la spécification. Qu'en est-il?

Et si, à l'inverse, les portes s'ouvraient pendant le déplacement d'un ascenseur, qu'en serait-il?

▸**EXERCICE 1.4**
Pouvez-vous encore voir d'autres problèmes liés à la spécification informelle de l'exercice 1.2?

2 Langages et méthodes de spécification formelle

Dans cette section, nous introduisons d'abord, en section 2.1, les fondements des méthodes formelles: logique, sémantique, preuves. Nous présentons ensuite en section 2.2 un des langages de spécification formelle les plus utilisés aujourd'hui, le langage Z de Jean-Raymond Abrial et Mike Spivey. Cette section a un caractère largement mathématique, et constitue une introduction aux langages des méthodes formelles. Nous reviendrons à la pratique de la spécification et de la vérification des logiciels quand nous aurons acquis les bases, en section 3.

2.1 Logique, langage et preuves

Qu'est-ce qu'une spécification? Si l'on débarrasse la notion de toute considération superflue, une spécification d'un produit est un *ensemble de propriétés* que l'on désire que le produit ait.

Comment établir une spécification qui soit la moins ambiguë possible? Les textes en langage naturel (français, anglais, etc.) se prêtent trop souvent à différentes interprétations contradictoires: voir par exemple les exercices 1.1 ou 1.2. Il est donc nécessaire d'utiliser un langage dont les constructions syntaxiques et la *sémantique* (la fonction qui a chaque construction associe son sens) soient précisées de façon claire et soient sans ambiguïté. C'est, par exemple et dans une certaine mesure, ce que font depuis longtemps les juristes en matière de droit.

Cependant, si tout texte de droit donne toujours lieu à interprétation, il est hors de question de laisser du flou dans la spécification d'un logiciel. Qui dit flou dit variation dans l'interprétation du texte, quiproquo, et finalement erreur de conception. Il existe pourtant un langage qui n'est pas sujet à ce genre de problème, à savoir le langage des mathématiques, la *logique*.

Un langage de spécification *formel* est un langage fondé sur la logique mathématique. Les expressions d'un tel langage sont des *formules mathématiques* servant à décrire les propriétés d'un produit, c'est-à-dire sa spécification. La *sémantique* d'un tel langage est une fonction qui à toute formula associe une *valeur de vérité*, vrai ($\top$) ou faux ($\bot$).

Fondamentalement, un programme π vérifie donc une spécification σ si et seulement si pour toute propriété P dans σ (qui est un ensemble de propriétés), $P(\pi)$ est une proposition vraie. Et pour vérifier une proposition mathématique F, la logique fournit une méthode: la *preuve*.

Retenons donc qu'un langage formel est un langage de formules dont la syntaxe est fixée à l'avance (par une grammaire, typiquement), dont la sémantique est non ambiguë (définie mathématiquement) et permet de fixer le *sens* de chaque proposition, et dont un système de preuves permet d'établir rigoureusement la validité (ou non) des propositions.

Illustrons ces concepts, et présentons une des logiques les plus utilisées: la logique classique du premier ordre, ou *calcul des prédicats*. Définissons-en d'abord la syntaxe:

Definition 2.1 (Formules) *Soit $\mathcal{V}$ un ensemble dénombrable infini de* variables d'individu *x, y, z, ..., $\mathcal{F}$ un ensemble non vide de* symboles de fonction *f, g, h, ..., et $\mathcal{P}$ un ensemble non vide de* symboles de prédicats *P, Q, R, ...*

On associe à chaque symbole de fonction ou de prédicat un entier positif ou nul appelé son arité*. Un symbole d'arité 0 est appelé une* constante*, un symbole d'arité 1 (resp. 2, 3) est dit* unaire *(resp. binaire, ternaire).*

L'ensemble des termes *t, u, v, ... est le plus petit ensemble contenant $\mathcal{V}$ et tel que pour tout symbole de fonction f d'arité n, pour tous termes t_1, ..., t_n, alors l'expressions $f(t_1, \ldots, t_n)$ est un terme.*

Les formules atomiques *A, B, C, ... sont les expressions de la forme $P(t_1, \ldots, t_n)$, où P est un symbole de prédicat, n est son arité, et où t_1, ..., t_n sont des termes.*

L'ensemble des formules *F, G, H, ... est le plus petit ensemble contenant les formules atomiques, et tel que $F_1 \wedge F_2$ (lu "F_1 et F_2"), $F_1 \vee F_2$ ("lu F_1 ou F_2"), $\neg F_1$ ("non F_1"), $F_1 \Rightarrow F_2$ ("F_1 implique F_2"), $F_1 \Leftrightarrow F_2$ ("F_1 équivalent à F_2"), $\forall x \bullet F$ ("pour tout x, F"), $\exists x \bullet F$ ("il existe x tel que F") sont des formules dès que F_1 et F_2 sont des formules.*

Cette grosse définition s'explique de la façon suivante. Les termes seront des expressions du langage servant à désigner des objets (des entiers, des chaînes de caractères, des arbres, des programmes, ...). On les forme à l'aide des opérateurs de base que sont les symboles de fonction. Par exemple, $+(1(), 2())$ est censé représenter la somme de la constante 1 et de la constante 2 (vus comme opérateurs zéroaires). On utilisera en général une syntaxe plus agréable à l'œil, pour écrire ce terme $1 + 2$, par exemple. (La syntaxe du sous-ensemble de Z que nous utilisons est donnée en section A.)

Les formules atomiques sont les propriétés de base que l'on peut exprimer sur les objets d'étude. Par exemple, le prédicat d'égalité $=$ (binaire) sert à former des *équations*, c'est-à-dire des formules atomiques de la forme $=(t_1, t_2)$ (que l'on notera plus simplement $t_1 = t_2$). Ces propriétés de base sont combinées à l'aide des *connecteurs logiques* $\wedge$ (conjonction), $\vee$ (disjonction), $\neg$ (négation), $\Rightarrow$ (implication), $\Leftrightarrow$ (équivalence) et des *quantificateurs* $\forall$ (quantificateur universel) et $\exists$ (quantificateur existentiel). Les quantificateurs sont la raison première de la présence de variables dans les termes. (On notera au passage que dans $\forall x \bullet F$ et $\exists x \bullet F$, x n'a de sens que dans F, et est indéfinie à l'extérieur de F: on dit que la portée de x est limitée à F. Cf. l'exercice 2.1 plus bas.)

Les termes dénotent des *valeurs* (des objets), et les formules dénotent des *valeurs de vérité*, vrai ou faux. Ceci est l'affaire de la *sémantique* du langage, qui est définie formellement comme suit. Si ρ est une fonction, on note $\rho \oplus \{x \mapsto y\}$ la fonction qui à x associe y, et à tout x' différent de x associe $\rho(x')$ (il s'agit de "ρ sauf qu'à x est associé y").

Nous utilisons la *convention de Strachey*: si e est une expression du langage (un terme, une formule), nous noterons $[\![e]\!]$ l'expression e en tant qu'objet manipulable (représenté sous forme de chaîne de caractères, par exemple). On dira aussi que $[\![e]\!]$ est le *nom* de l'expression e. Ceci permet de ne pas faire de confusion regrettable entre expressions (syntaxe) et valeurs (sémantique): $[\![\mathbb{N}]\!]$ est juste une chaîne de caractères, alors que $\mathbb{N}$ est l'ensemble des entiers naturels, par exemple.

Definition 2.2 (Sémantique) *Une* interprétation *I est un triplet $(D, (I_f)_{f \in \mathcal{F}}, (I_P)_{P \in \mathcal{P}})$, où D est un ensemble non vide de valeurs, appelé le* domaine *de l'interprétation, pour tout $f \in \mathcal{F}$ d'arité n, I_f est une fonction de D^n dans D, et pour tout $P \in \mathcal{P}$ d'arité n, I_P est une fonction de D^n dans $\{\top, \bot\}$.*

Un environnement *ρ est une fonction qui à tout nom de variable $[\![x]\!]$ associe un élément de D.*

La valeur *$\mathcal{E}[\![t]\!]I\rho$ d'un terme t (resp. la valeur $\mathcal{E}[\![F]\!]I\rho$ d'une formule F) dans l'interprétation I est un élément de D (resp. $\{\top, \bot\}$) défini par récurrence structurelle sur t (resp. F) par:*

- $\mathcal{E}[\![x]\!]I\rho = \rho([\![x]\!])$,
- $\mathcal{E}[\![f(t_1,\ldots,t_n)]\!]I\rho = I_f(\mathcal{E}[\![t_1]\!]I\rho,\ldots,\mathcal{E}[\![t_n]\!]I\rho)$,
- $\mathcal{E}[\![P(t_1,\ldots,t_n)]\!]I\rho = I_P(\mathcal{E}[\![t_1]\!]I\rho,\ldots,\mathcal{E}[\![t_n]\!]I\rho)$,
- $\mathcal{E}[\![F_1 \wedge F_2]\!]I\rho = \begin{cases} \top & \textit{si } \mathcal{E}[\![F_1]\!]I\rho = \top \textit{ et } \mathcal{E}[\![F_2]\!]I\rho = \top \\ \bot & \textit{sinon} \end{cases}$,
- $\mathcal{E}[\![F_1 \vee F_2]\!]I\rho = \begin{cases} \top & \textit{si } \mathcal{E}[\![F_1]\!]I\rho = \top \textit{ ou } \mathcal{E}[\![F_2]\!]I\rho = \top \\ \bot & \textit{sinon} \end{cases}$,
- $\mathcal{E}[\![\neg\, F_1]\!]I\rho = \begin{cases} \top & \textit{si } \mathcal{E}[\![F_1]\!]I\rho = \bot \\ \bot & \textit{sinon} \end{cases}$,
- $\mathcal{E}[\![F_1 \Rightarrow F_2]\!]I\rho = \begin{cases} \top & \textit{si } \mathcal{E}[\![F_1]\!]I\rho = \bot \textit{ ou } \mathcal{E}[\![F_2]\!]I\rho = \top \\ \bot & \textit{si } \mathcal{E}[\![F_1]\!]I\rho = \top \textit{ et } \mathcal{E}[\![F_2]\!]I\rho = \bot \end{cases}$,
- $\mathcal{E}[\![F_1 \Leftrightarrow F_2]\!]I\rho = \begin{cases} \top & \textit{si } \mathcal{E}[\![F_1]\!]I\rho = \mathcal{E}[\![F_2]\!]I\rho \\ \bot & \textit{sinon} \end{cases}$,
- $\mathcal{E}[\![\forall\, x \bullet F_1]\!]I\rho = \begin{cases} \top & \textit{si } \mathcal{E}[\![F_1]\!]I(\rho \oplus \{[\![x]\!] \mapsto v\}) = \top \textit{ pour tout } v \in D \\ \bot & \textit{sinon} \end{cases}$,
- $\mathcal{E}[\![\exists\, x \bullet F_1]\!]I\rho = \begin{cases} \top & \textit{s'il existe } v \in D \textit{ tel que } \mathcal{E}[\![F_1]\!]I(\rho \oplus \{[\![x]\!] \mapsto v\}) = \top \\ \bot & \textit{sinon} \end{cases}$.

Observer comment la sémantique définit la valeur d'une variable x: il s'agit simplement de la valeur que l'on a décidé par avance pour x dans l'environnement ρ, à savoir $\rho([\![x]\!])$. On observera aussi comment la sémantique des quantificateurs consiste à quantifier sur les valeurs v qu'il est possible d'attribuer à la variable quantifiée x.

▸ **EXERCICE 2.1**

Considérer la formule $(\exists\, x \bullet P(x)) \Rightarrow x = 1$. Quelle est la portée de la variable x? Quelle serait la lecture sémantique de cette formule? Discuter.

Pour établir la validité d'une formule logique, il est en général impossible de tester toutes les valeurs possibles. Par exemple, considérons la formule suivante:

$$\forall\, n \in \mathbb{N} \bullet n \neq 0 \vee \exists\, m \in \mathbb{N} \bullet n = m + 1$$

qui est une abréviation de:

$$\forall\, n \bullet \in (n, \mathbb{N}()) \Rightarrow (\neg\; = (n, 0()) \vee \exists\, m \bullet \in (m, \mathbb{N}()) \wedge = (n, +(m, 1())))$$

et considérons l'interprétation standard des symboles: à $\mathbb{N}$ on associe l'ensemble des entiers naturels, à $\in$ on associe la relation d'appartenance d'un élément à un ensemble, à $=$ la relation d'égalité, à 0 le zéro, à 1 le un et à $+$ l'addition des entiers. Dans cette interprétation, la formule est toujours vraie, mais il est exclu de le vérifier: cela reviendrait à énumérer tous les entiers pour n, et de trouver pour chaque un entier (valeur de m) vérifiant la formule.

À la place, on établit la validité de formules à l'aide d'un *système de preuve*. Il y a aussi un langage des preuves, et de même que pour les formules, ce langage doit lui aussi être formel pour éviter toute erreur de raisonnement. Voici par exemple une preuve en déduction naturelle de $(A \land B \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C)$:

$$
\begin{array}{lll}
(1) & A \land B \Rightarrow C & [\mathit{Hyp}] \\
(2) & A & [\mathit{Hyp}] \\
(3) & B & [\mathit{Hyp}] \\
(4) & A \land B & \text{par } [\land\ \mathit{Intro}](2,3) \\
(5) & C & \text{par } [\Rightarrow \mathit{Elim}](1,4) \\
(6) & B \Rightarrow C & \text{par } [\Rightarrow \mathit{Intro}](3,5) \\
(7) & A \Rightarrow B \Rightarrow C & \text{par } [\Rightarrow \mathit{Intro}](2,6) \\
(8) & (A \land B \Rightarrow C) \Rightarrow (A \Rightarrow B \Rightarrow C) & \text{par } [\Rightarrow \mathit{Intro}](1,7)
\end{array}
$$

Nous verrons en section 4 comment écrire des preuves formelles.

En résumé:

- Un langage formel, la *logique mathématique*, permet d'éliminer toute ambiguïté.
- Un langage de spécification formelle est un habillage plus confortable de la logique.
- Une logique, un langage formel, est un trio formé d'une *syntaxe*, d'une *sémantique* et d'un *système de preuves*.

2.2 Le langage Z

Les langages et méthodes de spécification et de vérification de logiciel sont nombreux. Souvent, de tels méthodes et langages ont été inventés pour répondre aux besoins particuliers de tel ou tel domaine, comme les logiques de Floyd-Hoare et la logique dynamique pour faire des preuves de programmes [Cou90, KT90], les logiques temporelles destinées à la vérification de circuits électroniques [Eme90]. D'autres méthodes (Merise, SADT, ...) ont un spectre beaucoup plus large, et permettent de spécifier non seulement toutes sortes de logiciels et même des projets non logiciels, mais ne sont que des méthodes semi-formelles: aucune sémantique précise n'est fournie pour les contenus des boîtes SADT par exemple. Il existe finalement des langages et méthodes à large spectre et *formelles*, parmi lesquelles Z, B, Coq, Nuprl, Isabelle, Larch, VDM, Raise, Unity, ... Nous avons choisi Z, qui a l'avantage de présenter clairement la plupart des concepts fondamentaux. On consultera [Spi89] pour une présentation plus complète du langage Z.

La syntaxe de Z est celle du calcul des prédicats et de la théorie des ensembles, enrobée de *sucre syntaxique* (une expression due à Donald Perlis). C'est-à-dire que le cœur de Z est le calcul des prédicats, mais comme ce dernier est très malcommode à manipuler, Z l'enrichit de constructions additionnelles qui le rendent plus agréable. L'apport principal de la syntaxe Z est un moyen de structuration des propriétés logiques par le *calcul des schémas*: toute spécification Z est organisée comme une collection de briques, les *schémas*, qui peuvent être combinées entre elles par des opérations prédéfinies.

Voici un exemple simple de schéma:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Premier}} \\
\hline
premiers : \mathbb{P}\,\mathbb{N} \\
\hline
\forall\, n : \mathbb{N} \bullet n \in premiers \Leftrightarrow \\
\qquad (n \neq 0 \land \\
\qquad \forall\, m : \mathbb{N} \bullet n \bmod m = 0 \Rightarrow m = 1 \lor m = n) \\
\hline
\end{array}
$$

Un schéma est une boîte de texte. Cette boîte porte un nom (ici, *Premier*), et consiste en deux parties: une partie de déclarations d'identificateurs appelée *signature* du schéma, (au-dessus de la ligne médiane: ici, nous déclarons un identificateur *premiers* de type $\mathbb{P}\,\mathbb{N}$ — ceci est identique à la notion de signature présente dans d'autres langages de programmation, notamment Standard ML), et une seconde partie appelée la *contrainte*, qui est une liste de formules décrivant la spécification des identificateurs déclarés. (Cette liste doit être vue comme le et logique de chaque ligne. Si la contrainte est omise, elle est comprise comme toujours vraie.)

La signification, autrement dit la *sémantique* d'un schéma est une collection d'environnements ρ, où chaque environnement lie les identificateurs déclarés dans le schéma à une valeur obéissant à la spécification. Illustrons ceci sur l'exemple *Premier* avant d'en donner une définition plus formelle. La sémantique de *Premier* est une collection d'environnements ρ de la forme $\{[\![premiers]\!] \mapsto v\}$, où v est une valeur appartenant à l'ensemble des parties de l'ensemble des entiers naturels, dénoté par $\mathbb{P}\,\mathbb{N}$. Cette valeur v doit de surcroît être telle que la spécification est vérifiée (la formule sous la barre médiane), qui exprime que les éléments de v sont les entiers non nuls et divisibles uniquement par 1 et par eux-mêmes.

Formellement, un schéma:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Nom}} \\
\hline
a_1 : T_1 \\
\ldots \\
a_m : T_m \\
\hline
F_1 \\
\ldots \\
F_n \\
\hline
\end{array}
$$

dénote la collection de tous les environnements ρ de la forme $\{[\![a_1]\!] \mapsto v_1, \ldots, [\![a_m]\!] \mapsto v_m\}$, où:

- pour tout i, $1 \leq i \leq m$, $\mathcal{E}[\![T_i]\!] I_Z \rho_i$ est défini et est un ensemble A_i tel que $v_i \in A_i$, où $\rho_i = \{[\![a_1]\!] \mapsto v_1, \ldots, [\![a_{i-1}]\!] \mapsto v_{i-1}\}$,
- et pour tout j, $1 \leq j \leq n$, $\mathcal{E}[\![F_j]\!] I_Z \rho = \top$.

où I_Z est le modèle standard des constructions du langage Z (cf. annexe B).

Dans l'exemple de *Premier*, les constructions Z utilisées sont $\mathbb{N}$ (dénotant l'ensemble des entiers naturels), $\mathbb{P}$ (la fonction qui à tout ensemble en argument associe son ensemble de parties), $\in$ (le prédicat d'appartenance d'un élément à un ensemble), $\neq$ (le prédicat d'inégalité), mod (la fonction de reste par division euclidienne), $=$ (le prédicat d'égalité) et 1 (la constante 1). Alors la sémantique du schéma *Premiers* est la collection des environnements ρ de la forme $\{[\![premiers]\!] \mapsto v\}$ tels que $\mathcal{E}[\![\forall\, n : \mathbb{N} \bullet n \in premiers \Leftrightarrow (n \neq 0 \land \forall\, m : \mathbb{N} \bullet n \bmod m = 0 \Rightarrow m = 1 \lor m = n)]\!] I_Z \{[\![premiers]\!] \mapsto v\} = \top$, c'est-à-dire la signification annoncée.

On notera qu'il y a exactement un environnement ρ vérifiant le schéma *Premier*: celui qui à $[\![premiers]\!]$ associe l'ensemble des nombres premiers. En général, il est possible qu'aucun environnement ne satisfasse un schéma (on dit que le schéma est *incohérent*), ou que plusieurs environnements satisfassent un schéma (on dit alors qu'il est *sous-déterminé*).

On notera aussi que le schéma *Premier*, s'il spécifie ce qu'est la notion d'entier premier (le *quoi*), ne précise par contre pas comment calculer si un nombre est premier ou pas (le *comment*): il est important de se souvenir qu'un langage de spécification permet de décrire le quoi indépendamment du comment. C'est ce qui fait la souplesse des langages de spécification comme Z, comparés aux langages de programmation, qui ne peuvent décrire que des algorithmes (le comment). Cependant, cette richesse d'expression se paie par le fait qu'il n'y a aucun moyen clair d'*exécuter* une spécification comme on le ferait d'un programme.

▸ **EXERCICE 2.2**

Quelle est la sémantique du schéma suivant?

Exercice 2.2

$$f : \mathbb{N} \to \mathbb{N}$$
$$g : \mathbb{N} \to \mathbb{N}$$

$$\forall\, n : \mathbb{N} \bullet n = f(n) * g(n)$$

Cette spécification est-elle cohérente? sous-déterminée?

▸ **EXERCICE 2.3**

Quelle est la sémantique du schéma suivant?

Exercice 2.3

$$n? : \mathbb{N}$$
$$p! : \mathbb{N}$$
$$q! : \mathbb{N}$$

$$n? = p! * q!$$

Comparer avec l'exercice 2.2.

▸ **EXERCICE 2.4**

On suppose que *NOM* et *DATE* sont deux types de données non spécifiés, c'est-à-dire que $\mathcal{E}[\![NOM]\!]I_Z\{\}$ et $\mathcal{E}[\![DATA]\!]I_Z\{\}$ sont des ensembles a priori quelconques. (On verra comment déclarer des types non spécifiés en section 2.6.) On souhaite décrire un agenda contenant pour chaque nom de personne (de type *NOM*) sa date d'anniversaire (de type *DATE*):

Anniversaires

$$connus : \mathbb{P}\, NOM$$
$$anniversaire : NOM \nrightarrow DATE$$

$$connus = \mathrm{dom}\; anniversaire$$

En consultant la section B.1, décrire la sémantique de ce schéma.

En résumé:

- Le langage Z est le calcul des prédicats plus la théorie des ensembles, enrobé de sucre syntaxique. L'élément de structuration d'une spécification Z est le *schéma*.
- Tout schéma consiste en une *signature* déclarant des identificateurs et une liste de *contraintes*.
- La sémantique d'un schéma est un environnement associant à chaque identificateur une valeur respectant la signature et obéissant aux contraintes.

2.3 Spécifier des comportements dynamiques en Z

De ce que nous avons présenté jusqu'ici, il peut sembler que les propriétés que l'on peut exprimer en Z sont extrêmement *statiques* (ne changeant pas au cours du temps), alors que le propre d'un programme est de faire évoluer l'état d'un calcul au cours du temps. En particulier, on aimerait vouloir spécifier des *procédures informatiques* qui prennent des arguments, retournent des résultats et modifient l'état global du système informatique considéré (c'est-à-dire modifient des cases mémoire, allouent ou libèrent de la mémoire, écrivent dans des fichiers, etc.)

Considérons en premier lieu la spécification d'une procédure qui n'a pas à modifier l'état global. Par exemple, spécifions la procédure *Carre*, qui prend un entier n en entrée et retourne son carré n^2. Pour accomplir ceci, il suffit de spécifier la collection de tous les couples (n, n^2), pour tout entier n. On écrira alors par exemple:

Carre

$n? : \mathbb{N}$
$res! : \mathbb{N}$

$res! = n? * n?$

On a utilisé ici une convention propre à Z: si $n?$ et $res!$ sont des identificateurs ordinaires, le fait de les décorer d'un point d'interrogation (?) ou d'exclamation (!) permet d'afficher notre intention de les voir traiter comme des variables d'entrée (?) ou de sortie (!), mais ne signifie rien de spécial pour Z. Ainsi, bien que le schéma *Carre* ne fasse que déclarer trois identificateurs liés par une contrainte, leurs noms fournissent l'aide à la lecture suivante: pour chaque entier $n?$ en entrée, on désire retourner un entier $res!$ tel que $res! = n? * n?$.

On notera bien que dans $res! = n? * n?$, $=$ est le prédicat d'égalité, pas une affectation à une variable! L'exercice 2.5 illustre ce point, par exemple.

▸ **EXERCICE 2.5**
Quelle est l'intention affichée quant à l'utilisation des identificateurs dans l'exercice 2.3?

De même, une autre convention permet de lire certaines décorations d'identificateurs comme indiquant un changement d'état. Le schéma:

Decompte

$n, n' : \mathbb{N}$

$n' + 1 = n$

qui spécifie exactement tous les environnements dans lesquels n vaut un de plus que n', le symbole prime ('$'$') qui décore n' est une aide à la lecture disant: n doit être compris comme l'état avant exécution de la procédure *Decompte*, et n' son état après exécution.

▸ **EXERCICE 2.6**
Dans le schéma *Decompte*, qu'est-il spécifié comme comportement de la procédure lorsque la valeur de départ n vaut 0? (Aide: ne cherchez pas à utiliser votre intuition, appliquez simplement les définitions sémantiques.) Discuter.

▸ **EXERCICE 2.7**
Écrire un schéma *LCG* dont l'intention est de spécifier une procédure qui modifie la valeur de l'entier *seed* à $(23227 * \mathit{seed} + 9881) \bmod 65535$, et retourne sa nouvelle valeur dans $\mathit{res}!$.

2.4 Inclusion de schémas

L'intérêt principal de la notation des schémas est qu'on peut les voir comme des modules élémentaires de spécifications. Nous présentons maintenant les opérateurs qui permettent de les combiner entre eux.

Prenons l'exemple suivant: on a écrit le schéma *Premier* de la section 2.2, et l'on souhaite spécifier une procédure *FacteurPremier* prenant un entier $n?$ en entrée et retournant un entier $p!$ premier qui divise $n?$. On serait tenté d'écrire:

$$
\begin{array}{|l}
\textit{FacteurPremier}_1 \\
\hline
n?, p! : \mathbb{N} \\
\hline
p! \in \mathit{premiers} \land n? \bmod p! = 0 \\
\hline
\end{array}
$$

mais ce n'est pas correct. Regardons en effet quelle pourrait être la sémantique du schéma *FacteurPremier*$_1$: il s'agit par définition de l'ensemble de tous les environnements ρ de la forme $\{[\![n?]\!] \mapsto n, [\![p!]\!] \mapsto p\}$ tels que $n \in \mathbb{N}$, $p \in \mathbb{N}$ et tels que $\mathcal{E}[\![p! \in \mathit{premiers} \land n? \bmod p! = 0]\!] I_Z \rho$ soit vrai. Mais l'environnement ρ ne donne aucune valeur à l'identificateur *premiers*, donc la valeur de la contrainte n'est jamais définie!

La solution est d'inclure le schéma définissant *premiers* dans le schéma précédent. On obtient le schéma correct suivant:

$$
\begin{array}{|l}
\textit{FacteurPremier}_2 \\
\hline
\mathit{premiers} : \mathbb{P}\,\mathbb{N} \\
n?, p! : \mathbb{N} \\
\hline
\forall n : \mathbb{N} \bullet n \in \mathit{premiers} \Leftrightarrow \\
\qquad (n \neq 0 \land \\
\qquad \forall m : \mathbb{N} \bullet n \bmod m = 0 \Rightarrow m = 1 \lor m = n) \\
p! \in \mathit{premiers} \land n? \bmod p! = 0 \\
\hline
\end{array}
$$

Mais c'est extrêmement lourd: c'est pourquoi Z permet d'écrire ce même schéma (au nom près) comme suit:

$$
\begin{array}{|l}
\textit{FacteurPremier} \\
\hline
\mathit{Premier} \\
n?, p! : \mathbb{N} \\
\hline
p! \in \mathit{premiers} \land n? \bmod p! = 0 \\
\hline
\end{array}
$$

La notation *Premier* est une *abréviation syntaxique*, une *macro*. L'expansion d'une macro de la forme *Premier* dans *FacteurPremier* se fait en remplaçant la ligne *Premier* par la signature de *Premier*, et en ajoutant la contrainte de *Premier* à la contrainte de *FacteurPremier*. (On vérifiera que cette expansion redonne bien le schéma $\mathit{FacteurPremier}_2$.) Les règles formelles d'expansion sont données en section A.1.

Reprenons maintenant l'exemple du schéma *Decompte*. Il est plus judicieux de le découper en deux parties, l'une spécifiant sur quel état il opère (une variable entière), l'autre spécifiant le changement d'état opéré par *Decompte*. Voici déjà une description de l'état, qui est un compteur:

Compteur

$n : \mathbb{N}$

(On a omis la partie contrainte, ce qui par convention revient à écrire *true* comme contrainte, et *true* est une constante logique valant toujours vrai.) On est alors tenté d'écrire le schéma suivant pour *Decompte*:

$\mathit{Decompte}_1$

Compteur

$n' + 1 = n$

mais ceci est incorrect! En effet, si n est déclaré par *Compteur*, n' ne l'est pas, donc $n' + 1 = n$ ne peut avoir aucune valeur sémantique bien définie.

On résout ce problème par la convention suivante: si S est un schéma, alors S' est le même schéma, mais dans lequel tous les identificateurs déclarées sont primées. Par exemple, $\mathit{Compteur}'$ est automatiquement le schéma suivant:

$\mathit{Compteur}'$

$n' : \mathbb{N}$

et l'on peut écrire *Decompte* comme suit:

$\mathit{Decompte}_2$

Compteur

$\mathit{Compteur}'$

$n' + 1 = n$

(De même, $S?$ ajouterait un ? à tous les identificateurs déclarés de S, et $S!$ ajouterait un !.)

Ceci reste encore lourd (surtout dans cet exemple-ci, où l'état *Compteur* est de taille modeste). C'est pourquoi en Z on convient que pour tout schéma S, ΔS est un schéma dont la signature est l'union de celles de S et de S', et donc la contrainte est la conjonction de celles de S et de S'. On peut alors écrire:

Decompte

$\Delta \mathit{Compteur}$

$n' + 1 = n$

ce qui se lit plus rapidement: *Decompte* est un schéma qui fait évoluer un état *Compteur* (contenant n), de sorte que la valeur de n après, plus 1, égale la valeur de n avant.

En reprenant la définition du schéma *Anniversaires* de l'exercice 2.4, et en s'aidant de la sémantique (section B.1), on se convaincra que le schéma:

$$
\begin{array}{|l}
\hline
\textit{Ajoute} \\
\Delta \textit{Anniversaires} \\
nom? : NOM \\
date? : DATE \\
\hline
nom? \notin connus \\
anniversaire' = anniversaire \cup \{(nom?, date?)\} \\
\hline
\end{array}
$$

ajoute une entrée à l'agenda des anniversaires, associant la date de naissance *date*? au nom *nom*? (une fonction Z, comme *anniversaire*, est un ensemble de couples (antécédent,image)). Le comportement de la procédure est indéfini si le nom est déjà dans l'agenda (contrainte $nom? \notin connus$). En fait, ce schéma est l'abréviation de:

$$
\begin{array}{|l}
\hline
\textit{Ajoute} \\
connus : \mathbb{P}\, NOM \\
anniversaire : NOM \rightarrow\!\!\!\!\!+ DATE \\
connus' : \mathbb{P}\, NOM \\
anniversaire' : NOM \rightarrow\!\!\!\!\!+ DATE \\
nom? : NOM \\
date? : DATE \\
\hline
connus = \mathrm{dom}\ anniversaire \\
connus' = \mathrm{dom}\ anniversaire' \\
nom? \notin connus \\
anniversaire' = anniversaire \cup \{(nom?, date?)\} \\
\hline
\end{array}
$$

On réalise sur ce petit exemple à quel point expanser les constructions de schémas présentées ici est indésirable.

Nous introduisons maintenant la dernière convention de cette section. On souhaite maintenant spécifier une procédure *Trouve* prenant un nom *nom*? en argument, et retournant la date de naissance associée dans l'agenda. On est tenté d'écrire:

$$
\begin{array}{|l}
\hline
Trouve_1 \\
\textit{Anniversaires} \\
nom? : NOM \\
date! : DATE \\
\hline
nom? \in connus \\
date! = anniversaire(nom?) \\
\hline
\end{array}
$$

où on inclut *Anniversaires* et non Δ*Anniversaires*, puisque l'agenda n'est pas modifié ... mais ce n'est pas ce que ce schéma dit!

En effet, il ne dit tout simplement rien sur l'état après de l'agenda: dire que l'agenda n'a pas changé, c'est dire que *connus* et *anniversaire* sont inchangés entre l'entrée dans la procédure et la sortie de la procédure, autrement dit $connus' = connus$ et $anniversaire' = anniversaire$. Mais dans ce schéma, $connus'$ et $anniversaire'$ sont *indéfinis*, et non contraints à être égaux à $connus$ et à $anniversaire$ respectivement. Il faudrait donc écrire ceci, en réalité:

$$
\begin{array}{|l}
\hline
\textit{Trouve}_2 \\
\Delta Anniversaires \\
nom? : NOM \\
date! : DATE \\
\hline
connus' = connus \\
anniversaire' = anniversaire \\
nom? \in connus \\
date! = anniversaire(nom?) \\
\hline
\end{array}
$$

Cette notation est lourde, d'autant plus qu'elle inclut l'information paradoxale $\Delta Anniversaires$ (il y a changement de l'"état $Anniversaires$) et $connus' = connus$, $anniversaire' = anniversaire$ (et ce changement consiste à ne rien changer). Z fournit pour cela la notation plus élégante $\Xi Anniversaires$:

$$
\begin{array}{|l}
\hline
\textit{Trouve} \\
\Xi Anniversaires \\
nom? : NOM \\
date! : DATE \\
\hline
nom? \in connus \\
date! = anniversaire(nom?) \\
\hline
\end{array}
$$

En général, si S est un schéma, ΞS est le schéma ΔS, avec en plus les contraintes $x_1 = x_1'$, ..., $x_k = x_k'$, lorsque x_1, ..., x_k sont les identificateurs déclarés par la signature de S. (Un moyen mnémotechnique est que Ξ est une lettre grecque majuscule comme Δ, mais ressemble aussi au symbole $=$ d'égalité.)

Il peut sembler stupide d'avoir à spécifier que l'état ne change pas, alors que c'est le comportement par défaut dans les langages de programmation séquentiels: si on écrit $x := 3$ en Pascal, on s'attend à ce que x prenne la valeur 3, mais que y et z restent inchangés. Mais ceci est en fait intéressant, en particulier dans le cas de spécifications de systèmes parallèles. Si deux processus P_1 et P_2 opèrent respectivement sur des états S_1 et S_2, on s'attend à ce que toute opération effectuée par P_1 par exemple modifie S_1, mais ne touche pas à S_2: ce qui ne veut pas dire que S_2 ne change pas, puisqu'en parallèle P_2 peut très bien le modifier! Que P_1 ne touche pas à S_2 est donc spécifié naturellement par l'absence de contraintes imposées par P_1 sur S_2.

▸ **EXERCICE 2.8**

Dans *Ajoute*, que peut-on dire de $connus'$?

▸ **EXERCICE 2.9**

Que décrit le schéma suivant?

Exercice 2.9

$\Xi Anniversaires$
$aujourdhui? : DATE$
$cartes! : \mathbb{P}\, NOM$

$cartes! = \{\ n : connus \mid anniversaire(n) = aujourdhui? \bullet n\ \}$

($\{x : T \mid P(x) \bullet e(x)\}$ dénote l'ensemble des $e(x)$, lorsque x est un élément de T tel que $P(x)$ soit vrai: consulter la section B.1 pour une définition plus précise.)

En résumé:

- mentionner un nom de schéma S dans la signature d'un autre schéma revient à l'inclure.
- S' est S, mais avec tous ses identificateurs primés.
- ΔS exprime un changement de l'état S, ΞS exprime que l'état S ne change pas. (Sinon, il peut arriver n'importe quoi à l'état S.)

2.5 Le calcul des schémas

Les opérations $'$, Δ et Ξ de la section 2.4 sont complétés par des opérateurs prenant un ou plusieurs schémas en argument, et produisant un nouveau schéma. Il s'agit encore une fois de mécanismes d'abréviation (des "macros").

Si S_1 et S_2 sont des schémas, on note $S_1 \wedge S_2$ le schéma dont la signature est l'union des signatures de S_1 et S_2, et donc la contrainte est la conjonction des contraintes de S_1 et de S_2. Par exemple, ΔS n'est rien d'autre que $S \wedge S'$.

De même, $S_1 \vee S_2$ est le schéma dont la signature est l'union des signatures de S_1 et de S_2, et dont la contrainte est la disjonction des contraintes de S_1 et S_2. Une utilisation courante de la disjonction de schéma est la représentation d'un *choix non-déterministe*. Par exemple, considérons le schéma *Decompte* (cf. section 2.4) et le schéma suivant:

CompteurStable

$\Xi Compteur$

alors *Decompte* $\vee$ *CompteurStable* spécifie une opération qui peut soit faire décroître le compteur soit le laisser inchangé.

On peut aussi donner un nom au schéma obtenu, par exemple:

$$DecompteOuStable \mathrel{\widehat{=}} Decompte \vee CompteurStable$$

▸ **EXERCICE 2.10**
Dans le schéma *DecompteOuStable*, qu'est-il spécifié lorsque l'entrée n vaut 0? Comparer avec l'exercice 2.6.

▸ **EXERCICE 2.11**
On souhaite modifier le schéma *Trouve* (cf. section 2.4). En effet, *Trouve* doit toujours renvoyer dans *date*! l'anniversaire de la personne de nom *nom*?. Mais l'on souhaite pouvoir implémenter la base de données des anniversaires sur un système de machines distribuées, et les

réseaux n'étant pas fiables, on souhaite prendre en compte la possibilité de l'échec de la requête de date d'anniversaire. Écrire un schéma *TrouveOuEchec* qui exécute *Trouve* et retourne *trouve*! valant 1, ou bien retourne *trouve*! valant 0. La phrase précédente est-elle ambiguë? (Indication: examiner la sémantique du schéma résultant dans le cas d'un échec.) Discuter.

▸ **EXERCICE 2.12**

Quelle est la sémantique du schéma $Decompte \lor Ajoute$? Ceci a-t-il une signification intuitive? Discuter.

Similairement, $S_1 \Rightarrow S_2$ a pour contrainte l'implication de celle de S_1 et de celle de S_2, et $S_1 \Leftrightarrow S_2$ a pour contrainte leur équivalence logique. De même, si S est un schéma, alors $\neg\, S$ est le schéma de même signature dont la contrainte est la négation logique de celle de S. Ces opérations sont beaucoup moins courantes que les précédentes.

On peut généraliser: si S est un schéma déclarant x de type T et $x_1 : T_1, \ldots, x_k : T_k$ et de contrainte F, alors $\exists\, x : T \bullet S$ est le schéma dont la signature est celle de S moins la déclaration de x (soit $x_1 : T_1, \ldots, x_k : T_k$) et dont la contrainte est $\exists\, x : T \bullet F$. (Similairement pour $\forall$.) Ceci permet de *cacher* des variables de schémas (comme les modules dans les langages de programmation permettent de cacher des identificateurs à d'autres modules). Par exemple, reprenons le schéma *Carre* de la section 2.3. Le schéma suivant:

$$EstUnCarre_1 \mathrel{\widehat{=}} \exists\, n? : \mathbb{N} \bullet Carre$$

aurait tout aussi bien pu être défini comme suit:

```
┌─ EstUnCarre₁ ─────────────────────────────
│ res! : ℕ
├──────────────
│ ∃ n? : ℕ • n? = res!
└───────────────────────────────────────────
```

et déclare donc uniquement *res*! comme identificateur, alors que *Carre* déclarait à la fois $n?$ et $res!$.

La manipulation consistant à cacher un (ou plusieurs) identificateurs est tellement courante que Z dispose d'une construction spéciale pour cela: si S est un schéma, et $x_1, \ldots, x_k$ sont des identificateurs de la signature de S, alors:

$$S \setminus (x_1, \ldots, x_k)$$

est le schéma dont la signature est celle de S moins les déclarations de $x_1, \ldots, x_k$, contraint par le fait que la contrainte de S est vérifiée, pour des valeurs gardées cachées de $x_1, \ldots, x_k$. Par exemple, on aurait pu écrire:

$$EstUnCarre_1 \mathrel{\widehat{=}} Carre \setminus (n?)$$

autrement dit, *res*! est un carré si et seulement si il est le carré d'un entier ($n?$) que l'on n'expose pas à l'extérieur du schéma.

Un problème avec $EstUnCarre_1$ est que, selon la convention de décoration des identificateurs, il ne spécifie pas une procédure testant si *res*! est un carré (comme le nom le suggère), mais une procédure retournant de façon non déterministe un carré. On aimerait disposer d'un schéma *EstUnCarre* prenant un entier $m?$ en argument, et effectuant ce test. On pourrait faire ceci par:

$$\begin{array}{|l}\hline \textit{EstUnCarre}_2 \\ m? : \mathbb{N} \\ \hline \exists\, n? : \mathbb{N} \bullet m? = n? * n? \\ \hline \end{array}$$

mais on aimerait aussi pouvoir réutiliser le schéma *Carre*. Les quantificateurs de schémas permettent d'effectuer un renommage des variables, grâce à l'astuce suivante. Considérons le schéma:

$$\begin{array}{|l}\hline \textit{RenommeResM} \\ m? : \mathbb{N} \\ res! : \mathbb{N} \\ \hline m? = res! \\ \hline \end{array}$$

alors le schéma:

$$\textit{EstUnCarre} \mathrel{\widehat=} (\textit{Carre} \wedge \textit{RenommeResM}) \setminus (res!, n?)$$

est par définition identique à:

$$\begin{array}{|l}\hline \textit{EstUnCarre} \\ m? : \mathbb{N} \\ \hline \exists\, res! : \mathbb{N};\ n? : \mathbb{N} \bullet res! = n? * n? \wedge m? = res! \\ \hline \end{array}$$

et la contrainte est logiquement équivalente à $\exists\, n? : \mathbb{N} \bullet m? = n? * n?$, soit ce qui était recherché.

▸ **EXERCICE 2.13**
À l'aide de *Carre*, écrire un schéma *Racine* extrayant d'un entier $m?$ sa racine carrée $racine!$, lorsque $m?$ est un carré.

En résumé:

- Le calcul des schémas combine des schémas pour en produire de nouveaux, à l'aide de $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$, $\exists$, $\forall$.
- $\setminus$ (une instance de $\exists$) permet de cacher certaines composantes de schémas.
- $\setminus$ permet aussi de renommer des identificateurs.

2.6 Opérations de base de Z

Au fil des exemples et des exercices, nous avons déjà introduit une bonne partie des types de données et des opérations de base que Z fournit. Les autres sont détaillés en section B. Nous en donnons ici un rapide aperçu.

Les objets fondamentaux en Z sont les *ensembles*. (Z est l'initiale de Zermelo, mathématicien allemand du début du vingtième siècle, à qui l'on doit une bonne partie de la formalisation standard de la théorie des ensembles.) Z fournit les opérateurs ensemblistes usuels: ensemble vide $\varnothing$, relation d'appartenance $\in$, union $\cup$, intersection $\cap$, différence ensembliste $\setminus$, etc.

Les types en Z sont représentés par des ensembles: être du type T, c'est appartenir à l'ensemble T. Ainsi, plutôt que de dire: x est de type $\mathbb{N}$, on écrira $x \in \mathbb{N}$. Les quantifications de la forme

$\forall x : T \mid P(x) \bullet Q(x)$ (par exemple) signifiant "pour tout x de type T tel que $P(x)$, $Q(x)$" se comprennent alors comme "pour tout x, si $x \in T$ et $P(x)$, alors $Q(x)$". De même, $\exists x : T \mid P(x) \bullet Q(x)$, qui signifie "il existe x de type T tel que $P(x)$, $Q(x)$" se comprend comme "il existe x tel que $x \in T$ et $P(x)$ et $Q(x)$".

Z incorpore aussi la notion de *fonction mathématique*: la fonction f, qui à x associe $f(x)$ (aussi noté $f\ x$) n'est rien d'autre que l'ensemble de tous les couples de la forme $(x, f\ x)$. On note $A \nrightarrow B$ l'ensemble de toutes les fonctions de A vers B, et $A \to B$ l'ensemble de toutes les fonctions *totales* de A vers B (c'est-à-dire définies sur tout élément de A). Le *domaine* $\mathrm{dom} f$ d'une fonction f est l'ensemble des x auxquels sont associés une image (une fonction totale de A vers B est donc telle que $\mathrm{dom} f = A$), et l'*image* $\mathrm{ran} f$ est l'ensemble des y qui sont image d'éléments du domaine.

Une fonction n'est pas une *procédure*: appliquer une fonction à un argument ne peut en aucun cas provoquer d'effets de bord, ou bien retourner deux résultats différents si elle est appelée deux fois avec le même argument.

Toute fonction est unaire, comme dans le langage ML. Pour coder une fonction à n arguments, on peut utiliser une fonction *currifiée*, c'est-à-dire une fonction de type $A_1 \nrightarrow A_2 \nrightarrow \ldots \nrightarrow A_n \nrightarrow A$ (qui appliquée à son premier argument renvoie la fonction des $n-1$ arguments restants): alors f appliquée à $e_1, \ldots, e_n$ est simplement $f\ e_1\ \ldots\ e_n$. On peut aussi utiliser une fonction qui à tout *n-uplet* d'arguments (vu comme un argument unique) associe le résultat: la fonction est alors de type $A_1 \times A_2 \times \ldots \times A_n \nrightarrow A$ et f appliquée à $e_1, \ldots, e_n$ est simplement $f(e_1, \ldots, e_n)$.

Toute fonction est, plus généralement, un cas particulier de *relation*, c'est-à-dire d'ensemble de couples: une fonction f est une relation qui à tout élément x associe au plus une image y. Une fonction f peut être appliquée à un argument e dans son domaine, par la notation $f\ e$. L'opération $\lhd$ permet d'obtenir la *restriction* d'une fonction à un ensemble: $A \lhd f$ est la fonction qui à tout x dans $\mathrm{dom} f \cap A$ associe $f(x)$. Symétriquement, l'*anti-restriction* $A \ntriangleleft f$ est la fonction qui à tout x dans $\mathrm{dom} f \setminus A$ associe $f(x)$. On peut aussi composer deux fonctions: $f \circ g$ est la fonction qui à tout x associe $f(g(x))$, lorsque x est dans le domaine de g et $g(x)$ est dans le domaine de f. On peut alors définir f^n, lorsque n est un entier non nul comme la composition de f avec elle-même n fois ($f^1 = f$, $f^2 = f \circ f$, etc.)

Ces notions (sauf l'application) se généralisent aux relations générales, que l'on peut voir comme des "multi-fonctions" qui à un élément peuvent associer plusieurs éléments: si R est une relation, R envoie x sur tous les y tels que $(x, y) \in R$ — par définition. On peut voir aussi une relation R comme un graphe dont les nœuds sont les x et les y, et telle que $(x, y) \in R$ si et seulement s'il y a un arc de x vers y. Si R et R' sont deux relations, $R \circ R'$ envoie x sur tous les z tels que R' envoie x sur y et R envoie y sur z, pour un certain y (en termes de graphe, cela consiste à mettre les deux graphes R' et R bout à bout). On définit alors la clôture transitive R^+ de toute relation R comme (voyant les relations comme des graphes) celle qui à tout x associe tous les y accessibles depuis x en suivant les arcs donnés pas R.

Les *suites* (appelées listes en ML et d'autres langages de programmation) sont toutes les suites finies $v_1, \ldots, v_n$ de valeurs, pour n'importe quel $n \in \mathbb{N}$. Elles sont représentées comme des fonctions: la suite ci-dessus n'est autre que la fonction qui à 1 associe v_1, à 2 associe v_2, ..., à n associe v_n. On la note alors $\langle v_1, \ldots, v_n \rangle$. On peut récupérer le i-ième élément d'une suite v en écrivant $v\ i$. On peut surtout concaténer deux suites v_1 et v_2 par l'opérateur $\frown$: $v_1 \frown v_2$ est la suite qui commence par les éléments de v_1 dans l'ordre, suivis de ceux de v_2 dans l'ordre.

On peut aussi spécifier de nouveaux types. Pour définir de nouveaux types, il suffit de les définir comme des ensembles. Voici par exemple le type *surjection* des surjections de A vers B défini dans le schéma suivant (on remarquera la notation $Surj[A, B]$ d'un schéma $Surj$ paramétré par deux paramètres A et B):

$$
\begin{array}{|l}
\hline
\textit{Surj}[A, B] \\
\textit{surjection} : \mathbb{P}(A \rightarrow B) \\
\hline
\textit{surjection} = \{ f : A \rightarrow B \mid (\forall y : B \bullet \exists x : A \bullet f\ x = y) \bullet f \} \\
\hline
\end{array}
$$

Si l'on souhaite définir un schéma déclarant, par exemple, une surjection *zigzag* de $\mathbb{N}$ vers $\mathbb{N} \times \mathbb{N}$, on peut le faire de la façon suivante:

$$
\begin{array}{|l}
\hline
\textit{ZigZag} \\
\textit{Surj}[\mathbb{N}, \mathbb{N} \times \mathbb{N}] \\
\textit{zigzag} : \textit{surjection} \\
\hline
\end{array}
$$

Finalement, il est parfois utile (comme dans l'exercice 2.4, page 12) de spécifier des types sans les définir. Dans l'exercice 2.4, on avait besoin de deux types non spécifiés *NOM* et *DATE*. On peut le faire en écrivant le paragraphe $[NOM, DATE]$ énumérant les noms des nouveaux types. La spécification complète de l'exercice 2.4 est donc:

$$[NOM, DATE]$$

$$
\begin{array}{|l}
\hline
\textit{Anniversaires} \\
\textit{connus} : \mathbb{P}\ \textit{NOM} \\
\textit{anniversaire} : \textit{NOM} \rightarrow\!\!\!\!\!+\ \textit{DATE} \\
\hline
\textit{connus} = \mathrm{dom}\ \textit{anniversaire} \\
\hline
\end{array}
$$

Les exercices suivants ont pour but d'assimiler les constructions essentielles. On se référera aux sections A et B pour la syntaxe et la sémantique des constructions de base de Z.

▸ **EXERCICE 2.14**

En utilisant le schéma *Surj*, définir un schéma *DeuxSurjections* déclarant deux identificateurs *zigzag* et *zagzig*, respectivement surjections de $\mathbb{N}$ vers $\mathbb{N} \times \mathbb{N}$, et de $\mathbb{N} \times \mathbb{N}$ vers $\mathbb{N}$. (Vérifiez sur la sémantique que vous avez bien écrit ce que vous souhaitiez écrire.)

▸ **EXERCICE 2.15**

Une *mémoire d'ordinateur* n'est rien d'autre qu'une fonction qui à toute adresse (élément d'un type *Adresse* non spécifié) associe une donnée (élément d'un type *Donnee* non spécifié). Écrire un schéma *Store* déclarant une mémoire *store*.

▸ **EXERCICE 2.16**

En utilisant l'état *Store* de l'exercice 2.15, spécifier une opération *Lire* qui lit une case mémoire d'adresse $adr?$ et retourne son contenu $data!$.

▸ **EXERCICE 2.17**

En utilisant l'état *Store* de l'exercice 2.15, spécifier une opération *Ecrire* qui écrit une donnée $data?$ dans une case mémoire d'adresse $adr?$. (Il y a plusieurs styles, mais une façon directe d'écrire *Ecrire* est d'utiliser l'opérateur $\oplus$.)

▸EXERCICE 2.18
En prenant pour point de départ les exercices 2.15 à 2.17, où l'on instanciera *Adresse* en utilisant le schéma:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Adresses}} \\
\hline
\textit{lomem}, \textit{himem} : \mathbb{N} \\
\textit{Adresse} : \mathbb{P}\,\mathbb{N} \\
\hline
\textit{lomem} \leq \textit{himem} \\
\textit{Adresse} = \textit{lomem} \mathrel{..} \textit{himem} \\
\hline
\end{array}$$

écrire une spécification abstraite de deux procédures *malloc* (d'argument $taille? : \mathbb{N}$, retournant une adresse mémoire $alloc! : \mathbb{N}$ de début d'un bloc de mémoire occupant *taille*? adresses consécutives, si c'est possible), et *free* (libérant un bloc précédemment alloué par *malloc*, d'adresse $free? : \mathbb{N}$).

▸EXERCICE 2.19
Un *graphe orienté* est un couple (V, E) d'un ensemble V dit de *sommets*, et d'un ensemble E d'*arcs*, tels que chaque arc e a un sommet de départ $source(e)$ et un sommet d'arrivée $cible(e)$. Spécifier la notion de graphe orienté au moyen d'un schéma *GrapheOriente*.

▸EXERCICE 2.20
Utilisant l'exercice 2.19, spécifier la notion de successeur d'un sommet dans un graphe orienté. On écrira un schéma *Successeur* définissant une relation *successeur*. (Un sommet est successeur d'un autre s'il est cible d'un arc de source l'autre.)

▸EXERCICE 2.21
Utilisant les exercices 2.19 et 2.20, écrire un schéma *DAG* spécifiant les graphes orientés acycliques. (Indication: utiliser l'opérateur de fermeture transitive.)

3 Développements formels de programmes

Une spécification n'est qu'un point de départ: il s'agit de développer un programme obéissant à cette spécification, et surtout de *vérifier* que le programme obéit bien à la spécification. Le procédé de vérification consistera principalement en une *preuve* — et nous verrons en section 4 ce que cela signifie — mais peut aussi consister en des tests fonctionnels ou de performances, en des évaluations par des groupes d'utilisateurs, etc. (Ces dernières méthodes se faisant préférablement en amont, sur un prototype.)

3.1 Traduire les besoins en spécification

La première tâche à accomplir est de traduire les besoins que l'on a en des spécifications. L'exercice 2.11, page 18, montre que la rigueur du langage de spécification formelle (ici, Z) nous force déjà à éliminer une ambiguïté qui n'apparaissait pas clairement dans le texte en français: dans le cas ou l'on dit “retourner *trouve*! valant 0”, a-t-on simplement le choix, ou bien ne doit-on retourner 0 qu'en cas d'échec de la requête?

Un cas plus réaliste est celui de l'exercice 1.1, page 5. Nous avons vu qu'un placement de commentaire malheureux pouvait induire des ambiguïtés dans un texte pourtant apparemment non ambigu (un déclaration de structures en C). Regardons comment nous aurions fait pour écrire cette

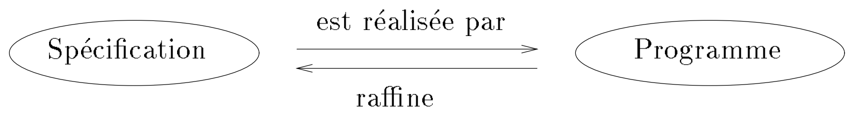

Fig. 3: Réalisation d'une spécification par un programme

spécification en Z, et supposons que nous avons déjà défini un type *MAPG_info* dans un schéma *Personne* décrivant ce qu'est une personne, ainsi que d'autres notions que nous ne précisons pas. Au moment de spécifier le champ `mapgc_info`, nous écrivons un schéma *MapGCInfo* comme suit:

$$\begin{array}{|l}
\textit{MapGCInfo} \\
\hline
\textit{Personne} \\
\textit{mapgc_info} : \mathbb{P}\, \textit{MAPG_info} \\
\hline
?
\end{array}$$

où le point d'interrogation est l'endroit où on va devoir écrire une contrainte. Cette dernière est une formule mathématique, qui sera:

$$\textit{mapgc\_info} = \{\ p : \textit{MAPG\_info} \mid \textit{autorise\_a\_conduire}(p) \land p \neq \textit{client} \bullet p\ \}$$

s'il s'agit de l'ensemble des personnes autres que la personne courante (que l'on suppose décrite par *client* de type *MAPG_client*) autorisées à conduire le véhicule, et:

$$\begin{array}{l}
\textit{en_europe}(\textit{client}) \Rightarrow \\
\quad \textit{mapgc_info} = \{\ p : \textit{MAPG_info} \mid \textit{autorise_a_conduire}(p) \land p \neq \textit{client} \bullet p\ \}
\end{array}$$

dans l'autre interprétation.

On notera que spécifier formellement le champ `mapgc_info` nous oblige à répondre à d'autres questions. Par exemple, la spécification informelle demande la "liste des autres personnes" que le client: mais qu'entend-on par "autres"? La réponse envisagée est que p est autre si et seulement si $p \neq \textit{client}$. Mais p est de type *MAPG_info*, et *client* est de type *MAPG_client*, et donc $p \neq \textit{client}$ n'a pas grande signification. Il faut donc amender la spécification, par exemple inclure une information de type *MAPG_info* dans le schéma *Personne*, ou bien ne comparer que les noms (mais que faire des homonymes?), etc.

On notera aussi que nous avions spécifié informellement "la liste" des autres personnes. Mais une liste (une suite) est une structure ordonnée: dans quel ordre est-elle présentée, et que représenterait le fait qu'une même personne apparaisse plusieurs fois? Il est probable que ces dernières informations n'aient tout simplement aucune signification: c'est pourquoi *mapgc_info* a été spécifié comme un *ensemble*, et non une liste, de personnes.

▸**EXERCICE 3.1**
Écrire une spécification pour l'ensemble d'ascenseurs de l'exercice 1.2. (Attention, le problème est de longue haleine.)

3.2 Raffinements

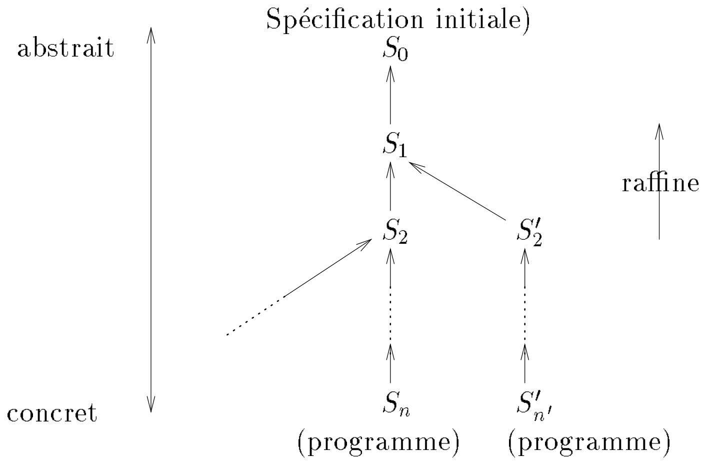

FIG. 4: Raffinements successifs

Il reste à organiser la relation entre spécification et programme, c'est-à-dire ce qui fait que l'opération nommée "réalisation" en figure 1 est correcte. Naïvement, le tableau de la situation est celui de la figure 3: on a écrit une spécification, et l'on souhaite écrire un programme et être sûr que le programme obéit à la spécification. Mais il y a souvent un saut gigantesque entre les deux: la spécification de l'agenda d'anniversaires (*Anniversaires*, *Ajoute*, *Trouve*) est très courte, mais sa réalisation utilisera peut-être des tables de hachage pour coder l'agenda, plus diverses optimisations, et demandera probablement mille ou deux mille lignes de code (sans les commentaires). Essayer de prouver directement que ce dernier programme vérifie la spécification est déjà quasi-irréalisable. Que dire alors de gros projets logiciels?

L'idée est de décomposer le passage de la spécification au programme en plusieurs sous-étapes: partant d'une spécification *abstraite* S_0, on va écrire des spécifications qui spécifient de plus en plus le comment et non plus seulement le quoi, c'est-à-dire des spécifications de plus en plus concrètes. Le raffinement sera progressif, facilitant ainsi la tâche de preuve de correction du raffinement.

Ainsi, sur la figure 4, S_0 est *raffinée* en S_1, qui est elle-même raffinée en S_2, ..., en S_n. Une spécification totalement concrète S_n sera un programme — nous verrons en quel sens en section 3.5.

Par exemple, dans le cas de l'agenda d'anniversaires (cf. page 12), on pourra réaliser S_1 en codant la fonction $anniversaire : NOM \nrightarrow DATE$ sous forme d'une suite de couples $(nom, date)$, avec $nom \in NOM$, $date \in DATE$. Dans S_1, on spécifiera *Trouve* (cf. page 17) comme demandant que $nom?$ soit un des noms de la liste, et $date!$ soit la date associée, mais sans dire encore par quel algorithme la trouver. On raffinera ensuite dans S_2 l'opération *Trouve* en demandant à ce que *Trouve* fasse une recherche en parcourant la liste de gauche à droite (depuis la tête jusqu'à la fin) par exemple, et ainsi de suite.

On pourrait aussi décider d'utiliser une structure de données plus efficace mais plus complexe pour représenter *anniversaire*, comme une table de hachage. Nul besoin alors de jeter les raffinements précédentes: il suffit de réécrire une spécification S'_1, qui raffine S_0 en codant *anniversaires* sous forme de table de hachage, et de raffiner S'_1 en S'_2, et ainsi de suite, comme le montre la figure 4. On voit que la méthode de raffinements successifs permet aussi de *gérer les versions* successives de choix algorithmiques et de structures de données.

Il y a deux grandes façons d'opérer un raffinement: raffiner les structures de données (changer la représentation de *anniversaires* sous forme de liste ou de table, par exemple), et raffiner les opérations effectuées (préciser dans quel ordre parcourir la liste par exemple).

Nous examinons dans les deux sections suivantes ces deux techniques, ainsi que les *obligations de preuve* qu'elles engendrent, c'est-à-dire les formules qu'il va falloir prouver pour être assuré que le raffinement est correct.

En résumé:

- Passer d'une spécification à un programme se fait par raffinements successifs, suffisamment peu importants pour être prouvés facilement.
- Deux façons de raffiner une spécification: raffiner les opérations et raffiner les données.

3.3 Raffinement d'opérations

Considérons deux schémas S_1 et S_2, spécifiant une opération. Dire que S_2 est un raffinement d'opérations de S_1, c'est dire que S_2 fait la même chose que S_1, mais de façon plus concrète. En particulier, si S_2 est suffisamment concret pour être compris comme un programme (nous verrons comment en section 3.5), cela revient à dire que S_2 *implémente* la spécification S_1.

Par exemple, considérons la spécification Get_1 suivante:

$Elements_1$

$l : \mathrm{seq}\,\mathbb{N}$

Get_1

$\Xi Elements_1$
$elt! : \mathbb{N}$

$elt! \in \mathrm{ran}\, l$

qui spécifie une procédure devant retourner un élément $elt!$ d'une liste l de l'état (qui ne change pas). Intuitivement, le schéma Get_2:

Get_2

$\Xi Elements_1$
$elt! : \mathbb{N}$

$l = \langle\rangle \Rightarrow elt! = 0$
$l \neq \langle\rangle \Rightarrow elt! = l(1)$

raffine Get_1: on a précisé une façon de retourner un élément de la liste l (en tout cas lorsque c'est possible).

Cette intuition demande à être formalisée, et la propriété d'être lié par une étape de raffinement demande a être définie mathématiquement.

Tout d'abord, chaque fois que Get_1 est applicable (à savoir chaque fois que la liste l n'est pas la suite vide $\langle\rangle$), Get_2 est applicable elle aussi. En fait, Get_2 est applicable même lorsque Get_1 ne l'est pas, et retourne 0 si l est la suite vide. En général, pour qu'une spécification S_2 raffine S_1, il faut que son domaine d'applicabilité soit au moins aussi grand que celui de S_1, et ce sera la première condition de raffinement.

Le domaine d'applicabilité d'un schéma S construit sur un état *Etat*, avec variable d'entrées $x_1? : T_1, \ldots, x_m? : T_m$ et de sortie $y_1! : T'_1, \ldots, y_n! : T'_n$ est décrit par une formule portant sur $x_1?$, $\ldots$, $x_m?$ et les variables d'état avant, qu'on appelle la *précondition* pre S de S, et qui est défini comme suit:

$$\mathrm{pre}\ S \mathrel{\widehat=} \exists\, Etat';\ y_1! : T'_1;\ \ldots;\ y_n! : T'_n \bullet S$$

La première condition de raffinement est donc:

$$\text{(D)}\quad \forall\, Etat;\ x_1? : T_1;\ \ldots;\ x_m? : T_m \bullet \mathrm{pre}\ S_1 \Rightarrow \mathrm{pre}\ S_2$$

Dans l'exemple Get_1/Get_2, cette condition s'écrit:

$$\begin{array}{l}
\forall\, l : \mathrm{seq}\,\mathbb{N} \bullet \\
\quad (\exists\, l' : \mathrm{seq}\,\mathbb{N};\ elt! : \mathbb{N} \bullet l' = l \land elt! \in \mathrm{ran}\, l) \Rightarrow \\
\quad\quad (\exists\, l' : \mathrm{seq}\,\mathbb{N};\ elt! : \mathbb{N} \bullet \\
\quad\quad\quad l = l' \land \\
\quad\quad\quad (l = \langle\rangle \Rightarrow elt! = 0) \land \\
\quad\quad\quad (l \neq \langle\rangle \Rightarrow elt! = l(1)))
\end{array}$$

(Noter les conditions $l = l'$, qui proviennent des expansions du schéma $\Xi Elements_1$.) On se convaincra que cette propriété est valide, intuitivement. Bien sûr l'intuition n'est pas une raison suffisante pour être sûr que la propriété est valide, et il nous faudra la *prouver*: nous verrons comment en section 4.

La deuxième condition que nous devons imposer est que le schéma concret S_2 doit produire des résultats ($Etat'$, $y_1!$, $\ldots$, $y_n!$) qui obéissent aux conditions imposées par le schéma abstrait S_1. Ceci se traduit mathématiquement par le fait que si S_2 est vérifié, alors S_1 est vérifié, c'est-à-dire le schéma $S_2 \Rightarrow S_1$. Plus précisément, on ne souhaite pas que S_1 soit vérifié chaque fois que S_2 est vérifié, mais seulement dans les cas où la spécification abstraite est applicable: en-dehors du domaine d'applicabilité de la spécification abstraite, on a le droit de faire ce que l'on veut. La condition est donc:

$$\begin{array}{l}
\text{(R)}\quad \forall\, Etat;\ Etat';\ x_1? : T_1;\ \ldots;\ x_m? : T_m;\ y_1! : T'_1;\ \ldots;\ y_n! : T'_n \bullet \\
\qquad\qquad \mathrm{pre}\ S_1 \land S_2 \Rightarrow S_1
\end{array}$$

Par exemple, dans le cas de Get_1/Get_2, cette condition est:

$$\begin{array}{l}
\forall\, l, l' : \mathrm{seq}\,\mathbb{N};\ elt! : \mathbb{N} \bullet \\
\quad (\exists\, l' : \mathrm{seq}\,\mathbb{N};\ elt! : \mathbb{N} \bullet l' = l \land elt! \in \mathrm{ran}\, l) \land \\
\quad (l' = l \land \\
\quad\ (l = \langle\rangle \Rightarrow elt! = 0) \land \\
\quad\ (l \neq \langle\rangle \Rightarrow elt! = l(1))) \\
\quad\quad \Rightarrow (l' = l \land elt! \in \mathrm{ran}\, l)
\end{array}$$

dont on se convaincra, encore une fois, qu'elle est valide.

<u>En résumé:</u> Un raffinement d'opérations est valide si:

- (D) le domaine d'applicabilité de la spécification concrète contient celui de la spécification abstraite,
- (R) si la spécification abstraite est applicable, alors toute résultat retourné par la spécification concrète vérifie la spécification abstraite.

3.4 Raffinement de données

L'autre façon de raffiner une spécification, c'est de changer les représentations de données utilisées: il s'agit du *raffinement de données*. L'exemple typique est celui du remplacement des structures de données ensemblistes par des structures de données plus concrètes: suites (listes), tableaux, tables de hachage, arbres, etc.

Par exemple, la spécification Get_1 de la section 3.3 provient de la spécification plus abstraite suivante:

$$\begin{array}{|l}
\hline
\textit{Elements}_0 \\
s : \mathbb{P}\,\mathbb{N} \\
\hline
\# s \in \mathbb{N} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Get}_0 \\
\Xi \textit{Elements}_0 \\
elt! : \mathbb{N} \\
\hline
elt! \in s \\
\hline
\end{array}$$

où s est spécifié comme étant non pas une liste, mais un ensemble fini (ce qu'exprime la condition $\# s \in \mathbb{N}$, cf. section B.1). Get_1 est intuitivement plus concrète que Get_1, car $l?$ y est implémentée par une structure de donnée informatique, la liste. Get_0 est aussi intuitivement plus abstraite que Get_1, car peu importe l'ordre dans lequel les éléments de l sont rangés dans Get_1, ce que capture bien le fait que $l?$ soit un ensemble dans Get_0.

Mais en premier, ce qui fait que l'on peut parler de raffinement, et que nous n'avons pas encore explicité, c'est le fait qu'il y a une correspondance entre les s abstraits et les l concrets. Cette correspondance est donnée par le schéma:

$$\begin{array}{|l}
\hline
\textit{Get}_{01} \\
\textit{Elements}_0 \\
\textit{Elements}_1 \\
\hline
s = \operatorname{ran} l \\
\hline
\end{array}$$

Ce dernier vérifie la condition suivante d'*adéquation*:

$$(A) \quad \forall\, \textit{Elements}_0 \bullet \exists\, \textit{Elements}_1 \bullet \textit{Get}_{01}$$

qui exprime que tout état abstrait est représenté par au moins un état concret, autrement dit, nous avons bien une représentation concrète de *tous* les états abstraits.

Les conditions de raffinement entre Get_0 et Get_1 sont alors, premièrement la généralisation de la condition (D), selon laquelle si Get_0 est applicable sur son état de départ s, alors Get_1 est applicable sur tous les états d'entrée l correspondants possibles, c'est-à-dire:

$$\forall\, s : \mathbb{P}\,\mathbb{N};\ l : \mathrm{seq}\,\mathbb{N} \bullet \mathrm{pre}\ Get_0 \wedge Get_{01} \Rightarrow \mathrm{pre}\ Get_1$$

et deuxièmement la généralisation de la condition (R), selon laquelle si Get_0 est applicable et que l'on applique Get_1, alors la sortie de Get_1 obéit à la spécification abstraite Get_0, c'est-à-dire $\forall\, s : \mathbb{P}\,\mathbb{N};\ l, l' : \mathrm{seq}\,\mathbb{N};\ elt! : \mathbb{N} \bullet \mathrm{pre}\ Get_0 \wedge Get_{01} \wedge Get_1 \Rightarrow Get_0$ ...ou presque! Ceci n'est pas juste: le résultat de Get_1 (l', $elt!$) ne peut pas obéir à Get_0, puisque ce dernier spécifie s' et $elt!$, non l'. Il faut donc exprimer que l'état après de Get_1 *correspond à un état concret après* qui vérifie Get_0, ce qui s'exprime ainsi:

$$\forall\, s : \mathbb{P}\,\mathbb{N};\ l, l' : \mathrm{seq}\,\mathbb{N};\ elt! : \mathbb{N} \bullet \mathrm{pre}\ Get_0 \wedge Get_{01} \wedge Get_1 \Rightarrow (\exists\, s' : \mathbb{P}\,\mathbb{N} \bullet Get'_{01} \wedge Get_0)$$

On notera que l'on a écrit $\exists\, s' : \mathbb{P}\,\mathbb{N} \bullet Get'_{01} \wedge Get_0$ et non $\forall\, s' : \mathbb{P}\,\mathbb{N} \bullet Get_{01'} \Rightarrow Get_0$, car il suffit que l'état concret après représente *un certain état abstrait* qui obéisse à la spécification abstraite Get_0, mais pas nécessairement tous.

En général, étant donnés un schéma $Etat_0$ d'états abstraits, un schéma $Etat_1$ d'états concrets, des entrées $\vec{x}? : \vec{T}$, des sorties $\vec{y}? : \vec{T}'$, (nous abrégeons une liste $x_1 : T_1, \ldots, x_m : T_m$ par $\vec{x} : \vec{T}$, pour plus de commodité) et un *schéma d'abstraction* $Abstr$ portant sur ces derniers (le schéma Get_{01}, ci-dessus), on dit qu'un schéma S_1 concret *raffine* un schéma S_0 abstrait via l'abstraction $Abstr$ si et seulement si:

- (A) $\forall\, Etat_0 \bullet \exists\, Etat_1 \bullet Abstr$,
- (D) $\forall\, Etat_0;\ Etat_1;\ \vec{x}? : \vec{T} \bullet \mathrm{pre}\ S_0 \wedge Abstr \Rightarrow \mathrm{pre}\ S_1$, et
- (R) $\forall\, Etat_0;\ Etat_1;\ Etat'_1;\ \vec{x}? : \vec{T};\ \vec{y}! : \vec{T}' \bullet \mathrm{pre}\ S_0 \wedge Abstr \wedge S_1 \Rightarrow (\exists\, Etat'_0 \bullet Abstr' \wedge S_0)$.

Ces formules doivent être vérifiées rigoureusement, et le moyen de le faire est l'écriture d'une *preuve*. Pour insister sur ce fait, on appelle ces formules des *obligations de preuve*. Les preuves sont le sujet de la section 4.

▸ EXERCICE 3.2

Dans l'exemple Get_0/Get_1, nous avons triché dans la présentation des obligations de preuve de domaine et de raffinement, dans le but d'en rendre la présentation plus simple. Utiliser les définitions formelles ci-dessus de (A), (D) et (R) pour écrire les obligations de preuve réelles. Sont-elles équivalentes aux obligations de preuve simplifiées du texte pour Get_0/Get_1? Sont-elles valides?

▸ EXERCICE 3.3

Écrire une spécification $Anniversaires_1$, $Trouve_1$, $Ajoute_1$ censée raffiner *Anniversaires*, *Trouve*, *Ajoute* en représentant l'agenda d'anniversaires par une suite de couples, comme indiqué dans le texte. Y a-t-il des difficultés imprévues? Comment corrigeriez-vous le problème?

▸ EXERCICE 3.4

Choisir un algorithme d'allocation mémoire (par liste de blocs libres, avec taille enregistrée en début de bloc, par exemple), et raffiner la spécification abstraite de l'exercice 2.18. Écrire les obligations de preuve de raffinement: quels sont les problèmes? Ne pas hésiter à revenir sur la spécification abstraite pour la reformuler.

▸ **EXERCICE 3.5**
On cherche à raffiner les spécifications abstraites des exercices 2.15 à 2.17, pour prendre en compte une mémoire virtuelle. C'est-à-dire que l'on a d'un côté une *mémoire réelle*, qui est une fonction $mem : REALADDR \nrightarrow DATA$, et de l'autre côté un *disque*, qui est une fonction $disk : DISKADDR \nrightarrow DATA$. Le système ne fonctionne que si la mémoire est une vue cohérente du disque, c'est-à-dire qu'il existe aussi une fonction $map : \mathrm{dom}\, mem \rightarrow DISKADDR$ qui à toute adresse mémoire utilisée associe l'adresse disque dont elle est le reflet.

Écrire les obligations de preuve de chaque étape de raffinement, en essayant qu'elles soient valides. Pourrait-on relâcher la condition de cohérence entre la mémoire et le disque? (Indication: au prix de complications algorithmiques, les systèmes de caches modernes n'imposent pas de conditions de cohérence aussi fortes. Imaginer comment c'est possible.)

3.5 Implémentation

À l'extrémité concrète d'un raffinement, on trouve une spécification suffisamment détaillée pour être exécutable. Cette spécification est écrite dans un sous-ensemble de Z qui n'utilise que certains connecteurs de schémas, reconnaissables comme étant exécutables. Nous en décrivons maintenant quelques-uns, inspirés des langages impératifs comme Pascal, dans le calcul des schémas de Z.

Soit *Etat* un schéma dénotant un état, typiquement l'ensemble des variables manipulées par le programme avec leurs types: convenons que la signature de *Etat* est $x_1 : T_1, \ldots, x_n : T_n$. Nous considérons des schémas dont la signature sera toujours $\Delta Etat$.

La première sorte de schéma que l'on peut écrire est une *affectation*. Si $x : T$ est un identificateur déclaré dans *Etat*, le schéma $(x := e)_{Etat}$ (affectation de x à l'expression e, dans une machine à état *Etat*) est décrit comme suit:

$x \leftarrow e$
$\Delta Etat$
$x' = e$

$$(x := e)_{Etat} \mathrel{\widehat{=}} (x \leftarrow e) \wedge \Xi(Etat \setminus (x))$$

c'est-à-dire que $(x := e)_{Etat}$ modifie x de sorte qu'il vaille e $(x \leftarrow e)$, et que toutes les autres variables de *Etat* sont inchangées. On notera que l'indice *Etat* est indispensable, et intervient de façon décisive dans la définition de l'affectation. (Dans un système parallèle par exemple, ceci permet de séparer les espaces de variables des différentes processus.)

Une autre opération simple que l'on peut écrire est la *composition séquentielle* de deux opérations. Si S_1 et S_2 sont deux schémas de signature $\Delta Etat$, alors $S_1 \fatsemi_{Etat} S_2$ ("exécuter S_1, puis S_2") est défini comme suit:

$$\begin{aligned} S_1 \fatsemi_{Etat} S_2 \mathrel{\widehat{=}} \exists\, y_1 : T_1; \ldots; y_n : T_n \bullet & \\ & (\exists\, Etat' \bullet [S_1 \mid x_1' = y_1 \wedge \ldots x_n' = y_n]) \wedge \\ & (\exists\, Etat \bullet [S_2 \mid x_1 = y_1 \wedge \ldots x_n = y_n]) \end{aligned}$$

autrement dit, il existe des valeurs intermédiaires $y_1, \ldots, y_n$ qui sont égales aux valeurs des variables d'état juste après S_1, et aux valeurs des variables d'état juste avant S_2.

Il est en fait possible de définir les tests (if/then/else) et les boucles while. Notons d'abord qu'une formule F dont les variables libres sont toutes déclarées dans *Etat* peut être utilisée pour représenter une *expression de test*. Ceci est accompli par la définition du schéma suivant:

$$
\begin{array}{|l}
\multicolumn{1}{l}{F?_{Etat}} \\
\hline
\Xi Etat \\
\hline
F \\
\hline
\end{array}
$$

Intuitivement, $F?_{Etat} \fatsemi_{Etat} S$ teste d'abord si F est vrai des variables d'état, sans les changer, puis si la condition F est vérifiée, S est "exécuté". Si la condition F n'est pas vérifiée, le schéma $F?_{Etat}$, ainsi que $F?_{Etat} \fatsemi_{Etat} S$ n'ont aucun modèle, c'est-à-dire que leur exécution ne peut rendre aucun état après: on considère que cela représente un échec (un plantage, ou la non-terminaison du programme, par exemple).

On peut maintenant coder le if/then/else comme suit (nous arrêtons de noter *Etat* en indice par souci de clarté, mais il est entendu que tous les opérateurs que nous introduisons dépendent de l'état et devraient être décorés par *Etat*):

$$\mathbf{if}\ F\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2 \mathrel{\widehat{=}} (F? \fatsemi S_1) \vee (\neg F? \fatsemi S_2)$$

Plus généralement, on peut coder le *choix entre commandes gardées* suivant, dû à E.W. Dijkstra:

$$[F_1 \to S_1 [\!] \ldots [\!] F_k \to S_k] \mathrel{\widehat{=}} (F_1? \fatsemi S_1) \vee \ldots \vee (F_k? \fatsemi S_k)$$

qui généralise la construction case/of de Pascal ou de ML (les formules F_i sont appelées les *gardes* des S_i). Noter qu'on pourrait définir **if** F **then** S_1 **else** S_2 par $[F \to S_1 [\!] \neg F \to S_2]$.

Pour réaliser les boucles, on va d'abord définir, pour chaque schéma S définissant une opération sur l'état *Etat*, un schéma S^*_{Etat} (ou plus simplement S^*) spécifiant un programme pouvant ne rien faire, ou exécuter S, ou exécuter $S \fatsemi S$, ou $S \fatsemi S \fatsemi S$, etc. Voici une définition possible:

$$
\begin{array}{|l}
\multicolumn{1}{l}{S^+_{Etat}} \\
\hline
\Delta Etat \\
\hline
((x_1, \ldots, x_n), (x'_1, \ldots, x'_n)) \in \{\ S \bullet ((x_1, \ldots, x_n), (x'_1, \ldots, x'_n))\ \}^+ \\
\hline
\end{array}
$$

$$S^*_{Etat} \mathrel{\widehat{=}} \Xi Etat \vee S^+_{Etat}$$

On peut alors définir:

$$\mathbf{while}\ F\ \mathbf{do}\ S\ \mathbf{end} \mathrel{\widehat{=}} (F? \fatsemi S)^* \fatsemi \neg F?$$

autrement dit, une boucle while exécute une suite $\underbrace{F? \fatsemi S \fatsemi \ldots \fatsemi F? \fatsemi S}_{n \text{ fois}} \fatsemi \neg F?$ où n est le nombre de tours dans la boucle, c'est-à-dire qu'elle exécute S tant que F est vrai, et sort lorsque F devient faux. On remarquera que la boucle while ne termine pas si et seulement si la spécification n'a aucun modèle.

On peut aussi généraliser la boucle while à la *boucle de commandes gardées*, dues à Dijkstra:

$$\mathbf{do}\ F_1 \to S_1 [\!] \ldots [\!] F_k \to S_k\ \mathbf{end} \mathrel{\widehat{=}} ((F_1? \fatsemi S_1) \vee \ldots \vee (F_k? \fatsemi S_k))^* \fatsemi \neg (F_1 \wedge \ldots \wedge F_k)?$$

qui exécute un S_i tel que F_i soit vrai, tant que l'une des gardes F_i est vraie, et sinon sort de la boucle. On notera que **while** F **do** S **end** = **do** $F \to S$ **end**.

On pourrait encore définir de nouveaux opérateurs de programmation sur les schémas pour traiter des appels de procédures, des allocations mémoire, du parallélisme (c'est ici que bien préciser l'état sur lequel on travaille devient crucial, comme dans la logique TLA de L. Lamport), etc. Nous ne le ferons pas ici, mais le lecteur peut s'amuser à cet intéressant exercice d'écriture d'une sémantique dénotationnelle pour son langage préféré.

<u>En résumé:</u>

- On peut voir les langages de programmation comme des sous-ensembles des langages de spécification.
- Implémenter une spécification, c'est la raffiner jusqu'à une spécification *pleinement concrète*, c'est-à-dire construite à partir des opérateurs de programmation et n'utilisant que des structures de données concrètes.

▸ **EXERCICE 3.6**

En se basant sur les exercices 2.19 à 2.21, écrire une spécification abstraite des graphes orientés acycliques ayant une unique racine (ou RDAG): on dit qu'un sommet n est la racine d'un graphe si et seulement si n n'est la successeur d'aucun sommet, mais tous les sommets sont accessibles depuis n.

Écrire une spécification abstraite d'une fonction qui numérote les sommets d'un RDAG de sorte qu'en partant d'un sommet n quelconque, tous les sommets accessibles depuis n portent des numéros strictement supérieurs.

Décrire un schéma décrivant l'algorithme de parcours d'un DAG en *profondeur d'abord*. (Informellement, il s'agit de partir de $i = 0$, et de maintenir un ensemble de sommets en attente, contenant au départ uniquement la racine: pour chaque sommet en attente qui n'a pas déjà été visité, on le retire de l'ensemble, on le numérote par i, on incrémente i, et ainsi de suite jusqu'à ce que l'ensemble soit vide. On utilisera l'intuition que la fermeture transitive peut coder des boucles pour exprimer cette récursion.)

Écrire les obligations de preuve: sont-elles valides?

4 Preuves

4.1 Systèmes de preuves

Les sections précédentes ont montré la nécessité d'établir la validité de propositions mathématiques pour montrer qu'un raffinement était correct. On établit la validité d'une proposition mathématique en la *prouvant*.

Mais qu'est-ce qu'une preuve? Les mathématiciens utilisent une notion informelle de preuve: un texte en langage naturel décrivant les étapes de raisonnement logique menant d'axiomes et de postulats à des théorèmes. (Et certaines étapes, jugées faciles, sont laissées implicites.) Mais si une spécification en langue naturelle peut être ambiguë, une preuve en langue naturelle peut l'être tout autant. Le problème est beaucoup plus sensible en validation de logiciel qu'en mathématiques: si les preuves sont usuellement faciles, elles sont en revanche très longues et répétitives, et une erreur a tôt fait de s'immiscer.

Il est donc nécessaire de disposer non seulement d'un langage de spécification formel, mais un aussi d'un *langage de preuves formelles* pour éviter toute erreur de raisonnement. (Certaines méthodes, comme Coq, Automath, Isabelle et les autres "logical frameworks", vont jusqu'à adopter un langage unique pour les spécifications et les preuves.) Un tel langage de preuves s'appelle un *système de preuve*.

Un système de preuve, étant formel, définit mathématiquement une preuve comme un objet mathématique, que l'on peut vérifier mécaniquement. (Trouver une telle preuve est une autre affaire, quoique ce soit possible dans certains fragments, comme les fragments propositionnels, c'est-à-dire sans quantificateurs.)

Un système de preuve consiste en un ensemble de *règles logiques*, permettant de tirer de nouvelles conclusions valides à partir de précédents énoncés valides, et un ensemble d'*axiomes*, ou postulats de base, dont la validité est supposée suffisamment évidente pour être admise. Un des systèmes de preuve les plus simples est la *déduction naturelle*, et nous fonderons notre système de preuve sur ce dernier.

4.2 La déduction naturelle

Supposons donc que nous avons à notre disposition une suite de formules $F_1, \ldots, F_n$, consistant en les axiomes de base d'une part, et les lemmes et théorèmes déjà démontrés à partir de ceux-là d'autre part. On peut imaginer que nous avons écrit chacune de ces formules sur une ligne, repérée par un numéro par exemple:

$$\begin{array}{lll} (1) & F_1 & [Axiome_1] \\ & \vdots & \\ (n) & F_n & [Axiome_n] \end{array}$$

et qu'une étape de démonstration consiste à rajouter une nouvelle ligne à la suite des précédentes. Cette ligne consistera en trois parties: un nouveau numéro de ligne d'abord, la nouvelle formule que nous avons déduite ensuite, et enfin la règle qui a été utilisée pour produire cette dernière formule (ce que nous appellerons la *justification* de cette étape de raisonnement). La justification d'un axiome est vide (un axiome est présupposé), mais il sera utile de référencer un axiome quand on voudra l'utiliser: c'est ce à quoi sert le nom ($[Axiome_1]$, ..., $[Axiome_n]$ ici) de l'axiome. Les axiomes que nous utiliserons en Z sont donnés en section C.

Une des premières choses que l'on peut déduire d'un ensemble de formules est l'une de ces formules elles-même. (C'est trivial mais fondamental.) C'est-à-dire que nous pouvons toujours ajouter la ligne suivante:

$$(n+1) \quad F_i \quad \text{par } (i)$$

où F_i, $1 \leq i \leq n$, est l'une des formules précédentes. La justification consiste juste en le numéro de ligne où se trouve l'hypothèse F_i utilisée.

Cette règle seule n'accomplit pas grand-chose. Une des règles qui effectue une déduction non triviale est l'*élimination de l'implication* $[\Rightarrow Elim]$, aussi appelée *modus ponens*. Intuitivement, elle dit que si on sait déjà $G \Rightarrow H$ et G, alors on peut en déduire H. Nous représentons ceci par la règle:

$$\begin{array}{lll} & \vdots & \\ (i) & G \Rightarrow H & [\ldots] \\ & \vdots & \\ (j) & G & [\ldots] \\ & \vdots & \\ (n+1) & H & \text{par } [\Rightarrow Elim](i,j) \end{array}$$

où $1 \leq i, j \leq n$ (la situation symétrique où $j < i$ est aussi admise) et $(n+1)$ est la nouvelle ligne déduite. Noter que la conclusion H de la règle est obtenue à partir de $G \Rightarrow H$ en *éliminant* une implication, d'où le nom de la règle.

De façon symétrique, on utilise une règle d'*introduction de l'implication* [$\Rightarrow$ *Intro*], qui permet d'introduire des hypothèses auxiliaires. Pour prouver une formule de la forme $F \Rightarrow G$, il est souvent nécessaire de poser F comme hypothèse auxiliaire, pour prouver G. On en déduit alors $F \Rightarrow G$, et ceci tient toujours: on n'a plus besoin de supposer F pour que $F \Rightarrow G$ soit vrai. Il faut donc un mécanisme permettant de passer d'une preuve de G sous l'hypothèse F à une preuve de $F \Rightarrow G$ sans cette hypothèse — ce qu'on appelle *décharger* l'hypothèse auxiliaire F. L'introduction de l'implication effectue ceci, en permettant de passer de la preuve:

$$\begin{array}{lll} (1) & F_1 & [\ldots] \\ & \vdots & \\ (n) & F_n & [\ldots] \\ (n+1) & F & [\mathit{Hyp}] \\ & \vdots & \\ (n+m) & G & [\ldots] \end{array} \quad \text{à la preuve:} \quad \begin{array}{lll} (1) & F_1 & [\ldots] \\ & \vdots & \\ (n) & F_n & [\ldots] \\ (n+1)' & F \Rightarrow G & [?] \end{array}$$

où nous ne précisons pas (encore) la justification [?]. Cette notation est maladroite, parce qu'elle oblige à écrire la preuve de gauche, puis à la réécrire en celle de droite. On combine en fait les deux en une preuve unique, en entourant la partie de la première preuve qui permet de déduire G de l'hypothèse temporaire F (lignes $(n+1)$ à $(n+m)$) dans un cadre comme suit:

$$\begin{array}{|lll} \multicolumn{1}{l}{\vdots} & & \\ \hline (n+1) & F & [\mathit{Hyp}] \\ & \vdots & \\ (n+m) & G & [\ldots] \\ \hline (n+m+1) & F \Rightarrow G & \text{par } [\Rightarrow \mathit{Intro}](n+1, n+m) \end{array}$$

Le cadre a pour effet de masquer la sous-preuve $(n+1) - (n+m)$, et de ne laisser visible que la conclusion (la ligne $(n+m+1)$) de la sous-preuve masquée. (La boîte est opaque de l'extérieur. Mais remarquer que les lignes $(n+2)$, ..., $(n+m-1)$ peuvent utiliser les résultats des lignes (1), ..., (n): de l'intérieur de la boîte, on peut regarder vers l'extérieur.)

Ces règles ainsi que les autres règles de la logique propositionnelle (c'est-à-dire sans les quantificateurs) sont présentées en figure 5. Dans les règles [$\vee$ *Intro*$_1$], [$\vee$ *Intro*$_2$], [$\neg$ *Elim*] et [*false Elim*], G est une formule arbitraire, et nous convenons du fait que *false* est une constante logique valant le faux (sémantiquement, $\mathcal{E}[\![\mathit{false}]\!]I\rho = \bot$). La règle [*false Elim*] est aussi appelée *ex falsum quod libet* (du faux on peut tout déduire). La règle [$\vee$ *Elim*] permet de prouver une formule H sachant que $F \vee G$ est vrai, en montrant H sous l'hypothèse F et sous l'hypothèse G séparément: il s'agit du *raisonnement par cas*. Enfin [*Bivalence*] est aussi appelée loi du *tiers exclu* (*tertium non datur*), et énonce que toute proposition ne peut être que vraie ou fausse.

La figure 6 présente d'autres règles de logique propositionnelle, que l'on peut déduire des précédentes, quoiqu'assez péniblement. On notera que le principe de raisonnement par contradiction, c'est-à-dire le fait de démontrer F en posant $\neg F$ comme hypothèse, et en déduire *false*,

Recopie:
$$\begin{array}{lll} & \vdots & \\ (i) & F & [\ldots] \\ & \vdots & \\ (n+1) & F & \text{par } (i) \end{array}$$

Éliminations:	Introductions:
$\begin{array}{lll} & \vdots & \\ (i) & F \Rightarrow G & [\ldots] \\ & \vdots & \\ (j) & F & [\ldots] \\ & \vdots & \\ (n+1) & G & \text{par } [\Rightarrow \mathit{Elim}](i,j) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & F & [\mathit{Hyp}] \\ & \vdots & \\ (n) & G & [\ldots] \\ (n+1) & F \Rightarrow G & \text{par } [\Rightarrow \mathit{Intro}](i,n) \end{array}$
$\begin{array}{lll} & \vdots & \\ (i) & F \wedge G & [\ldots] \\ & \vdots & \\ (n+1) & F & \text{par } [\wedge\ \mathit{Elim}_1](i) \\ (\text{resp.} & G & \text{par } [\wedge\ \mathit{Elim}_2](i)) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & F & [\ldots] \\ & \vdots & \\ (j) & G & [\ldots] \\ & \vdots & \\ (n+1) & F \wedge G & \text{par } [\wedge\ \mathit{Intro}](i,j) \end{array}$
$\begin{array}{lll} & \vdots & \\ (i) & F \vee G & [\ldots] \\ & \vdots & \\ (j) & F \Rightarrow H & [\ldots] \\ & \vdots & \\ (k) & G \Rightarrow H & [\ldots] \\ & \vdots & \\ (n+1) & H & \text{par } [\vee\ \mathit{Elim}](i,j,k) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & F & [\ldots] \\ & \vdots & \\ (n+1) & F \vee G & \text{par } [\vee\ \mathit{Intro}_1](i) \\ (\text{resp.} & G \vee F & \text{par } [\vee\ \mathit{Intro}_2](i)) \end{array}$
$\begin{array}{lll} & \vdots & \\ (i) & \neg F & [\ldots] \\ & \vdots & \\ (j) & F & [\ldots] \\ & \vdots & \\ (n+1) & G & \text{par } [\neg\ \mathit{Elim}](i,j) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & F & [\mathit{Hyp}] \\ & \vdots & \\ (n) & \mathit{false} & [\ldots] \\ (n+1) & \neg F & \text{par } [\neg\ \mathit{Intro}](i) \end{array}$
$\begin{array}{lll} & \vdots & \\ (i) & \mathit{false} & [\ldots] \\ & \vdots & \\ (n+1) & G & \text{par } [\mathit{false}\ \mathit{Elim}](i) \end{array}$	$\begin{array}{lll} & \vdots & \\ (n+1) & F \vee \neg F & \text{par } [\mathit{Bivalence}] \end{array}$

FIG. 5: Règles de déduction naturelle en logique propositionnelle

$\vdots$ (i) $\neg\neg F$ $[\ldots]$ $\vdots$ $(n+1)$ F par $[\neg\neg\, \mathit{Elim}](i)$	$\vdots$ $(n+1)$ true par $[\mathit{true}\ \mathit{Intro}]$
$\vdots$ (i) $\neg(F \Rightarrow G)$ $[\ldots]$ $\vdots$ $(n+1)$ F par $[\neg \Rightarrow \mathit{Elim}_1](i)$ (resp. $\neg G$ par $[\neg \Rightarrow \mathit{Elim}_2](i)$)	$\vdots$ (i) F $[\ldots]$ $\vdots$ (j) $\neg G$ $[\ldots]$ $\vdots$ $(n+1)$ $\neg(F \Rightarrow G)$ par $[\neg \Rightarrow \mathit{Intro}](i,j)$
$\vdots$ (i) $\neg(F \vee G)$ $[\ldots]$ $\vdots$ $(n+1)$ $\neg F$ par $[\neg \vee \mathit{Elim}_1](i)$ (resp. $\neg G$ par $[\neg \vee \mathit{Elim}_2](i)$)	$\vdots$ (i) $\neg F$ $[\ldots]$ $\vdots$ (j) $\neg G$ $[\ldots]$ $\vdots$ $(n+1)$ $\neg(F \vee G)$ par $[\neg \vee \mathit{Intro}](i,j)$
$\vdots$ (i) $\neg(F \wedge G)$ $[\ldots]$ $\vdots$ (j) $(\neg F) \Rightarrow H$ $[\ldots]$ $\vdots$ (k) $(\neg G) \Rightarrow H$ $[\ldots]$ $\vdots$ $(n+1)$ H par $[\neg \wedge \mathit{Elim}](i,j,k)$	$\vdots$ (i) $\neg F$ $[\ldots]$ $\vdots$ $(n+1)$ $\neg(F \wedge G)$ par $[\neg \wedge \mathit{Intro}_1](i)$ (resp. $\neg(G \wedge F)$ par $[\neg \wedge \mathit{Intro}_2](i)$)

FIG. 6: Règles de déduction propositionnelle (suite)

s'écrit comme suit:

$$
\begin{array}{|lll}
(1) & \neg\, F & [\mathit{Hyp}] \\
 & \vdots & \text{(un certain nombre d'étapes de démonstration)} \\
(n) & \mathit{false} & \\
\hline
(n+1) & \neg\,\neg\, F & \text{par } [\neg\ \mathit{Intro}](1)
\end{array}
$$
$$(n+2) \quad F \quad \text{par } [\neg\,\neg\ \mathit{Elim}](n+1)$$

Voici maintenant une astuce pour trouver des preuves dans ce système: essayer d'appliquer les règles d'introduction en dernier, et les règles d'élimination en premier.

Quelle règle d'introduction utiliser comme dernière règle de preuve est donné par la forme de la formule à prouver. Par exemple, si l'on doit prouver une formule de la forme $F \wedge G$, il suffit de chercher une preuve de F et une de G, et de les combiner par la règle $[\wedge\ \mathit{Intro}]$. De même, si l'on doit prouver une formule de la forme $F \Rightarrow G$, il suffit de poser F en hypothèse et de démontrer G, et d'appliquer $[\Rightarrow \mathit{Intro}]$. Ou si l'on doit prouver une formule de la forme $\neg\,(F \Rightarrow G)$, il suffit de prouver F et $\neg\, G$, et d'appliquer $[\neg\ \Rightarrow \mathit{Intro}]$. Ceci décompose le but à prouver en un ou plusieurs *sous-buts* de preuve.

Une fois les buts suffisamment décomposés, on peut chercher à appliquer des règles d'élimination pour établir les sous-buts courants.

Par exemple, pour prouver $A \wedge B \Rightarrow A$, on va poser $A \wedge B$ comme hypothèse et prouver A, puis terminer par $[\Rightarrow \mathit{Intro}]$. Le sous-but à prouver est maintenant A, pour lequel aucune règle d'introduction ne pourra servir. On explore alors les hypothèses courantes (ici, $A \wedge B$), et on applique des règles d'élimination: $[\wedge\ \mathit{Elim}_1]$ fait l'affaire, et produit A, le résultat cherché.

Nous donnons les règles sur les quantificateurs en section C.2, que nous conseillons de lire avant de passer aux exercices 4.2 et suivants. En général, les règles et axiomes portant sur les constructions Z se trouvent en section C.

▸EXERCICE 4.1

Chercher des preuves des formules suivantes:

$$
\begin{array}{llll}
(i) & A \Rightarrow B \Rightarrow A & (ii) & (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C) \\
(iii) & \neg\,\neg\, A \Rightarrow A & (iv) & A \Rightarrow \neg\,\neg\, A \\
(v) & ((A \Rightarrow B) \Rightarrow A) \Rightarrow A & (vi) & ((A \vee B) \wedge \neg C) \Rightarrow B \\
(vii) & \neg A \wedge ((A \Rightarrow B) \Rightarrow A) & (viii) & ((A \wedge \neg B) \Rightarrow C) \Rightarrow B \\
(ix) & A \Rightarrow (\neg A \vee B) & (x) & (A \Rightarrow B) \vee (B \Rightarrow A) \\
(xi) & \multicolumn{3}{l}{((A \Rightarrow B) \wedge (C \Rightarrow D)) \Rightarrow ((A \wedge C) \Rightarrow (B \vee D))} \\
(xii) & \multicolumn{3}{l}{((A \Rightarrow B) \vee (C \Rightarrow D)) \Rightarrow ((A \wedge C) \Rightarrow (B \vee D))}
\end{array}
$$

Lesquelles de ces formules ne sont pas prouvables et pourquoi? (On remarquera que toute formule prouvable est valide, c'est-à-dire vraie dans toute interprétation.)

▸EXERCICE 4.2

Chercher des preuves de:

$$
\begin{array}{ll}
(i) & (\forall\, x : \mathbb{N} \bullet P(x)) \Rightarrow (\exists\, x : \mathbb{N} \bullet P(x)) \\
(ii) & (\exists\, y : A \bullet P(y) \vee Q(y)) \Rightarrow (\exists\, x : A \bullet P(x)) \vee (\exists\, x : A \bullet Q(x)) \\
(iii) & (\exists\, x : A \bullet P(x)) \vee (\exists\, x : A \bullet Q(x)) \Rightarrow (\exists\, y : A \bullet P(y) \vee Q(y))
\end{array}
$$

▸ **EXERCICE 4.3**
Chercher des preuves de:

$$
\begin{array}{ll}
(i) & \exists\, x : \mathbb{N} \bullet (P(x) \Rightarrow \forall\, y : \mathbb{N} \cdot P(y)) \\
(ii) & \forall\, e : \mathbb{P}\, E \mid e \neq \varnothing \bullet \exists\, x : E \bullet x \in e \\
(iii) & \forall\, n : \mathbb{N} \bullet n \bmod 2 = 0 \vee n \bmod 2 = 1
\end{array}
$$

(Pour (i), on utilisera le raisonnement par contradiction. Pour (ii), on utilisera l'axiome $[Ext]$ et le raisonnement par contradiction, notamment. Pour (iii), on utilisera l'un des axiomes de récurrence pour établir que pour tout entier k, $k \leq 1$ implique $k = 0 \vee k = 1$.)

▸ **EXERCICE 4.4**
Chercher une preuve de (on s'inspirera de l'exercice 4.3 (i)):

$$
(i) \quad \exists\, x : A \bullet (P(x) \Rightarrow \forall\, y : A \cdot P(y))
$$

▸ **EXERCICE 4.5**
Prouver les obligations de preuve de l'exercice 3.2.

▸ **EXERCICE 4.6**
Esquisser des preuves des obligations de preuve de l'exercice 3.4. (On n'écrira pas la preuve formelle, juste une preuve mathématique rigoureuse, énonçant quels axiomes sont utilisés, et à partir de laquelle une preuve formelle serait reconstructible sans effort d'invention.)

▸ **EXERCICE 4.7**
De même que dans l'exercice 4.6, esquisser des preuves des obligations de preuve de l'exercice 3.5.

▸ **EXERCICE 4.8**
De même que dans l'exercice 4.6, esquisser des preuves des obligations de preuve de l'exercice 3.6.

5 Conclusion

Ce cours a présenté, à travers une version simplifiée du langage Z, un certain nombre de principes présidant à la conception et à la vérification formelle du logiciel: expression d'une spécification par une formule logique (structurée en Z par le calcul des schémas), vérification par la preuve, et décomposition du passage de la spécification à la réalisation par raffinements successifs.

Il y a beaucoup d'autres langages et méthodes formels: Coq a une approche plus rigoriste encore que Z et est fondé sur la notion de preuve, alors que la méthode B privilégie la notion de raffinement. Le langage TLA (Temporal Logic of Actions) et quant à lui un langage minimal de spécification de systèmes parallèles de préférence, les langages synchrones (Esterel, par exemple) ne spécifient que des systèmes parallèles synchronisés par une horloge globale, et disposent d'outils automatiques de vérification, etc.

Comment choisir un outil de spécification ou de validation du logiciel? Le maître mot ici est la *confiance*: plus on apporte de rigueur à la conception et à la vérification d'un logiciel (et en général, de n'importe quel produit), plus grande est la confiance que l'on peut porter en ce produit, mais aussi plus le développement est long et coûteux. Les méthodes formelles sont indispensables aujourd'hui dans quelques domaines protégés (sécurité des transports, conception de circuits intégrés digitaux, contrats défense, etc.). Dans les autres, on atteint en général un niveau de confiance suffisant par la simple rédaction d'une spécification formelle ou semi-formelle, permettant de détecter des ambiguïtés qui passeraient inaperçues en langage naturel.

Dans tous les cas, il est une chose plus importante que l'outil utilisé: le bon sens, et la rigueur, l'esprit logique, qui est le fondement de ce cours et dont nous espérons avoir fourni quelques bases.

A Syntaxe de Z

La syntaxe de Z est donnée par la table suivante (cf. [Spi89]) en EBNF. Nous ne présentons que la partie de la syntaxe nécessaire pour suivre le cours. Nous mentionnons les priorités des opérateurs lorsque c'est nécessaire: les opérateurs de priorités les plus hautes sont alors ceux qui lient leurs arguments le plus fort, par exemple $x+y*z$ abrège $x+(y*z)$. Nous mentionnons aussi l'associativité des opérateurs pour lesquels cela est nécessaire: ainsi, $Sch_1 \fatsemi Sch_2 \fatsemi Sch_3$ est-il une abréviation de $(Sch_1 \fatsemi Sch_2) \fatsemi Sch_3$ car $\fatsemi$ est associatif à gauche (mentionné par la lettre L), mais $A \Rightarrow B \Rightarrow C$ abrège $A \Rightarrow (B \Rightarrow C)$ parce que $\Rightarrow$ est associatif à droite (lettre R).

Specification	::=	[*Paragraph* '\n']* *Paragraph*	
Paragraph	::=	'[' [*Ident* ',']* *Ident* ']'	
	|	*Schema-Box*	
	|	*Ident* [*Gen-Formals*] ' $\widehat{=}$ ' *Schema-Exp*	
Gen-Formals	::=	'[' [*Ident* ',']* *Ident* ']'	

Schema-Box ::=

```
┌── Ident [Gen-Formals] ─────────────────
│ Decl-Part
├─────────────
│ Axiom-Part
└────────────────────────────────────────
```

Decl-Part	::=	[*Basic-Decl Sep*]* *Basic-Decl*	
Axiom-Part	::=	[*Predicate Sep*]* *Predicate*	
Sep	::=	'; ' | '\n'+	
Schema-Exp	::=	*Quantor Schema-Text* ' $\bullet$ ' *Schema-Exp*	
	|	*Schema-Exp*-1	
Quantor	::=	' $\forall$ ' | ' $\exists$ '	
Schema-Exp-1	::=	'[' *Schema-Text* ']'	
	|	*Schema-Text*	
	|	*Schema-Ref*	
	|	'Δ' *Schema-Exp*-1 | 'Ξ' *Schema-Exp*-1 | '$\neg$ ' *Schema-Exp*-1	priorité 9
	|	*Schema-Exp*-1 *Schema-Connector* *Schema-Exp*-1	
	|	*Schema-Exp*-1 ' \ ' '(' [*Ident* ',']* *Ident* ')'	priorité 2
	|	'(' *Schema-Exp* ')'	
Schema-Connector	::=	' $\fatsemi$ '	L, priorité 1
	|	*Connector*	
Connector	::=	' $\Leftrightarrow$ '	L, priorité 4
	|	' $\Rightarrow$ '	R, priorité 5
	|	' $\vee$ '	L, priorité 6
	|	' $\wedge$ '	L, priorité 7
Schema-Text	::=	*Declaration* [' | ' *Predicate*]	
Schema-Ref	::=	*Ident Decoration* [*Gen-Actuals*]	
Gen-Actuals	::=	'[' [*Expression* ',']* *Expression* ']'	
Declaration	::=	[*Basic-Decl* '; ']* *Basic-Decl*	
Basic-Decl	::=	[*Ident* ',']* *Ident* ' : ' *Expression*	
	|	*Schema-Ref*	
Predicate	::=	*Quantor Schema-Text* ' $\bullet$ ' *Predicate*	
	|	*Predicate*-1	

$$
\begin{array}{lcll}
\mathit{Predicate}\text{-}1 & ::= & \mathit{Expression}\ \mathit{In\text{-}Rel}\ \mathit{Expression} & \\
 & | & \mathit{Schema\text{-}Ref} & \\
 & | & \text{‘}\mathit{true}\text{’} \mid \text{‘}\mathit{false}\text{’} & \\
 & | & \text{‘}\neg\text{’}\ \mathit{Predicate}\text{-}1 \text{ priorité } 9 & \\
 & | & \mathit{Predicate}\text{-}1\ \mathit{Connector}\ \mathit{Predicate}\text{-}1 & \\
 & | & \text{‘(’}\ \mathit{Predicate}\ \text{‘)’} & \\
\mathit{In\text{-}Rel} & ::= & \text{‘}=\text{’} \mid \text{‘}\neq\text{’} \mid \text{‘}\notin\text{’} \mid \text{‘}\subseteq\text{’} \mid \text{‘}\subset\text{’} \mid \text{‘}<\text{’} \mid \text{‘}>\text{’} \mid \text{‘}\leq\text{’} \mid \text{‘}\geq\text{’} & \\
\mathit{Expression} & ::= & \mathit{Expression}\ \mathit{In\text{-}Gen}\ \mathit{Expression} & \text{priorité } 1 \\
 & | & [\mathit{Expression}\text{-}1\ \text{‘}\times\text{’}]^*\ \mathit{Expression}\text{-}1 & \text{priorité } 2 \\
\mathit{In\text{-}Gen} & ::= & \text{‘}\leftrightarrow\text{’} \mid \text{‘}\nrightarrow\text{’} \mid \text{‘}\rightarrow\text{’} & \\
\mathit{Expression}\text{-}1 & ::= & \mathit{Expression}\text{-}1\ \mathit{In\text{-}Fun}\ \mathit{Expression}\text{-}1 & \\
 & | & \mathit{Pre\text{-}Gen}\ \mathit{Expression}\text{-}3 & \\
 & | & \mathit{Expression}\text{-}3^+ & \\
 & | & \mathit{Expression}\text{-}3^{\mathit{Expression}} & \\
 & | & \mathit{Expression}\text{-}2 & \\
\mathit{In\text{-}Fun} & ::= & \text{‘}\,..\,\text{’} & \text{priorité } 2 \\
 & | & \text{‘}+\text{’} \mid \text{‘}-\text{’} \mid \text{‘}\cup\text{’} \mid \text{‘}\setminus\text{’} \mid \text{‘}\frown\text{’} & \text{priorité } 3 \\
 & | & \text{‘}*\text{’} \mid \text{‘div’} \mid \text{‘mod’} \mid \text{‘}\cap\text{’} & \text{priorité } 4 \\
 & | & \text{‘}\oplus\text{’} & \text{priorité } 5 \\
 & | & \text{‘}\lhd\text{’} \mid \text{‘}\unlhd\text{’} & \text{priorité } 6 \\
\mathit{Pre\text{-}Gen} & ::= & \text{‘}\mathbb{P}\text{’} \mid \text{‘seq’} & \\
\mathit{Expression}\text{-}2 & ::= & \mathit{Expression}\text{-}2\ \mathit{Expression}\text{-}3 & \\
 & | & \mathit{Expression}\text{-}3 & \\
\mathit{Expression}\text{-}3 & ::= & \mathit{Ident}\ [\mathit{Gen\text{-}Actuals}] & \\
 & | & \mathit{Number} & \\
 & | & \mathit{Schema\text{-}Ref} & \\
 & | & \mathit{Set\text{-}Expression} & \\
 & | & \text{‘}\langle\text{’}\ [[\mathit{Expression}\ \text{‘,’}]^*\ \mathit{Expression}]\ \text{‘}\rangle\text{’} & \\
 & | & \text{‘(’}\ [\mathit{Expression}\ \text{‘,’}]^*\ \mathit{Expression}\ \text{‘)’} & \\
 & | & \mathit{Expression}\text{-}3\ \text{‘.’}\ \mathit{Ident} & \\
 & | & \text{‘(’}\ \mathit{Expression}\ \text{‘)’} & \\
\mathit{Set\text{-}Exp} & ::= & \text{‘\{’}\ [[\mathit{Expression}\ \text{‘,’}]^*\ \mathit{Expression}]\ \text{‘\}’} & \\
 & | & \text{‘\{’}\ \mathit{Schema\text{-}Text}\ \text{‘}\bullet\text{’}\ \mathit{Expression}\ \text{‘\}’} & \\
\mathit{Ident} & ::= & \mathit{Word}\ \mathit{Decoration} & \\
\mathit{Number} & ::= & [\text{‘}\mathtt{1}\text{’} - \text{‘}\mathtt{9}\text{’}][\text{‘}\mathtt{0}\text{’} - \text{‘}\mathtt{9}\text{’}]^* & \\
\mathit{Word} & ::= & [\text{‘}A\text{’} - \text{‘}Z\text{’}\text{‘}a\text{’} - \text{‘}z\text{’}]^+ & \\
\mathit{Decoration} & ::= & \mathit{Stroke}^* & \\
\mathit{Stroke} & ::= & \text{‘}{\prime}\text{’} \mid \text{‘!’} \mid \text{‘?’} \mid \text{‘}_0\text{’} \mid \text{‘}_1\text{’} \mid \text{‘}_2\text{’} \mid \text{‘}_3\text{’} \mid \text{‘}_4\text{’} \mid \text{‘}_5\text{’} \mid \text{‘}_6\text{’} \mid \text{‘}_7\text{’} \mid \text{‘}_8\text{’} \mid \text{‘}_9\text{’} &
\end{array}
$$

A.1 Règles d'expansion de schémas

Il s'agit des règles permettant d'expanser des abréviations syntaxiques (les "macros" du langage Z):

- Une boîte de schéma (*Schema-Box*):

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Nom}} \\
\hline
decl_1 \\
\vdots \\
decl_m \\
\hline
P_1 \\
\vdots \\
P_n \\
\hline
\end{array}$$

est exactement la même chose que la déclaration:

$$Nom \mathrel{\widehat=} [decl_1;\ \ldots;\ decl_m \mid P_1 \wedge \ldots \wedge P_n]$$

Il ne s'agit que de deux syntaxes différentes, la première étant conçue pour être plus lisible. (De même si le nom *Nom* est décoré d'une liste de paramètres formels, i.e. de classe *Gen-Formals*. De plus, si $n = 0$, alors on convient que $P_1 \wedge \ldots \wedge P_n$ désigne la constante *true*.)

- Expansion des déclarations:

  $$x_1, \ldots, x_n : E$$

  est une abréviation de:

  $$x_1 : E;\ \ldots;\ x_n : E$$

- Expansion des quantificateurs universels:

  $$\forall\, decls \mid P \bullet Q$$

  est une abréviation de:

  $$\forall\, decls \bullet P \Rightarrow Q$$

- Expansion des quantificateurs existentiels:

  $$\exists\, decls \mid P \bullet Q$$

  est une abréviation de:

  $$\exists\, decls \bullet P \wedge Q$$

- Expansion des références de schéma (*Schema-Ref*): si S est défini comme un schéma $[decls \mid P]$, alors toute texte de schéma (*Schema-Text*) plus bas dans la spécification de la forme:

  $$decls_1;\ S;\ decls_2 \mid P_1$$

  est une abréviation de :

  $$decls_1;\ decls;\ decls_2 \mid P \wedge P_1$$

- Décoration de schémas: si S est défini comme un schéma $[x_1 : T_1; \ldots; x_k : T_k \mid P]$, alors $S' = [x_1' : T_1; \ldots; x_k' : T_k \mid P[x_1'/x_1, \ldots, x_k'/x_k]$.

  (De même pour $S?$, $S!$, etc.)

- Notation ΔS: si S est défini comme un schéma $[x_1 : T_1; \ldots; x_k : T_k \mid P]$, alors ΔS est le schéma $[x_1 : T_1; \ldots; x_k : T_k; x_1' : T_1; \ldots; x_k' : T_k \mid P \wedge P[x_1'/x_1, \ldots, x_k'/x_k]]$.

- Notation ΞS: si S est défini comme un schéma $[x_1 : T_1; \ldots; x_k : T_k \mid P]$, alors ΞS est le schéma $[x_1 : T_1; \ldots; x_k : T_k; x_1' : T_1; \ldots; x_k' : T_k \mid P \wedge P[x_1'/x_1, \ldots, x_k'/x_k] \wedge x_1 = x_1' \wedge \ldots \wedge x_k = x_k']$.

- Notations $S_1 \wedge S_2$, $S_1 \vee S_2$, $S_1 \Rightarrow S_2$, $S_1 \Leftrightarrow S_2$. Si S_1 est défini comme un schéma $[x_1 : T_1; \ldots; x_k : T_k \mid P_1]$ et S_2 comme $[x_1' : T_1'; \ldots; x_{k'}' : T_{k'}' \mid P_2]$, et *op* est $\wedge$, $\vee$, $\Rightarrow$ ou $\Leftrightarrow$, alors S_1 *op* S_2 est $[x_1 : T_1; \ldots; x_k : T_k; x_1' : T_1'; \ldots; x_{k'}' : T_{k'}' \mid P_1 \ op\ P_2]$.

- Notation $\neg\, S$: si S est défini comme un schéma $[x_1 : T_1; \ldots; x_k : T_k \mid P]$, alors $\neg\, S$ est le schéma $[x_1 : T_1; \ldots; x_k : T_k \mid \neg\, P]$.

- Quantifications de schémas: si S est défini comme un schéma $[x_1 : T_1; \ldots; x_k : T_k \mid P]$, *sch* est un texte de schéma $x_{i_1} : T_{i_1}, \ldots, x_{i_m} : T_{i_m}$, avec $1 \leq i_1 < \ldots < i_m \leq k$, alors pour tout quantificateur Q (soit $\exists$ soit $\forall$), $Q\ sch \bullet S$ est le schéma $[decls \mid Q\ sch \bullet P]$, où *decls* est la liste des déclarations $x_i : T_i$, $1 \leq i \leq k$, où x_i n'est pas un des x_{i_j}, $1 \leq j \leq m$.

- Notation $S \setminus (x_1, \ldots, x_k)$: si S est défini comme un schéma déclarant x_1 de type T_1, …, x_k de type T_k (entre autres), alors $S \setminus (x_1, \ldots, x_k)$ est $\exists\, x_1 : T_1; \ldots; x_k : T_k \bullet S$.

B Sémantique des constructions de base de Z

La sémantique de Z présuppose l'existence d'un domaine d'interprétation D suffisamment grand pour contenir toutes les valeurs nécessaires. Ce D existe, mais peu importe ce qu'il est réellement: nous ne le précisons donc pas. Nous donnons les fonctions sémantiques I_f et I_P d'interprétation des symboles de fonctions f et de prédicats P, ainsi que les extensions nécessaires à la sémantique.

Les opérateurs de base ont pour sémantique:

- $=$: $I_=(v, v')$ est vrai ($\top$) si et seulement si v et v' sont égaux.
- $\neq$: $I_=(v, v')$ est vrai ($\top$) si et seulement si v et v' sont différents.

B.1 Théorie des ensembles

Les symboles de prédicat sont:

- $\in$: $I_\in(v, v')$ est vrai ($\top$) si et seulement si v est un élément de v', si ce dernier est un ensemble. (Sinon, $I_\in(v, v')$ est un booléen non spécifié.)
- $\notin$: $I_\notin(v, v')$ est la négation logique de $I_\in(v, v')$.

Les symboles de fonction sont:

- $\cup$: $I_\cup(v, v')$ est l'union de v et de v', si ces derniers sont des ensembles.
- $\cap$: $I_\cap(v, v')$ est l'intersection de v et de v', si ces derniers sont des ensembles.

- $\setminus$: $I_\setminus(v, v')$ est la différence entre v et v' (l'ensemble de tous les éléments de v qui ne sont pas dans v'), si v et v' sont des ensembles.
- $\{\_, \ldots, \_\}$ (ensembles donnés par énumération: une énumération de n éléments est donnée par un symbole de fonction n-aire): $I_{\{\_,\ldots,\_\}}(v_1, \ldots, v_n)$ est l'ensemble qui contient exactement les éléments $v_1, \ldots, v_n$.
- $\_ \times \ldots \times \_$ (produit cartésien n-aire): $I_{\_\times\ldots\times\_}(v_1, \ldots, v_n)$ est l'ensemble des n-uplets de la forme $(v'_1, \ldots, v'_n)$, lorsque $v'_1, \ldots, v'_n$ parcourent $v_1, \ldots, v_n$ respectivement, à condition que ces derniers soient des ensembles.
- $\mathbb{P}$: $I_{\mathbb{P}}(v)$ est l'ensemble de toutes les parties de v, lorsque ce dernier est un ensemble.
- $\bigcup$ (union distribuée): $I_{\bigcup}(v)$ est l'union de tous les éléments de v, à condition que v soit un ensemble d'ensembles. (En particulier, $\bigcup\{e_1, \ldots, e_n\} = e_1 \cup \ldots \cup e_n$ est toujours vrai.)
- $\bigcap$ (intersection distribuée): $I_{\bigcap}(v)$ est l'intersection de tous les éléments de v, à condition que v soit un ensemble *non vide* d'ensembles. (En particulier, $\bigcap\{e_1, \ldots, e_n\} = e_1 \cap \ldots \cap e_n$ est toujours vrai, pour $n \neq 0$.)
- $\leftrightarrow$ (espace des relations): $I_\leftrightarrow(v, v')$ est l'ensemble de toutes les relations de v vers v', lorsque ces derniers sont des ensembles. (Une relation de v vers v' est juste un ensemble de couples (x, y), avec x élément de v et y élément de v'.)
- $\nrightarrow$ (espace des fonctions): $I_\nrightarrow(v, v')$ est l'ensemble de toutes les fonctions de v vers v', lorsque ces derniers sont des ensembles. (Une fonction de v vers v' est une relation de v vers v' qui à tout élément de v associe au plus un élément de v'.)
- $\rightarrow$ (espace des fonctions totales): $I_\rightarrow(v, v')$ est l'ensemble de toutes les fonctions totales de v vers v', lorsque ces derniers sont des ensembles. (Une fonction est totale de v vers v' si et seulement si à tout élément de v elle associe un élément de v'.)
- L'application $e\ e'$ de e (une fonction) à e' (son argument): $\mathcal{E}[\![e\ e']\!]I_Z\rho$ est par définition $I_{app}(\mathcal{E}[\![e]\!]I_Z\rho, \mathcal{E}[\![e']\!]I_Z\rho)$, où $I_{app}(v, v')$ est défini comme $v(v')$, lorsque v est une fonction et v' est dans son domaine. Il est indéfini sinon.
- dom (domaine d'une relation): $I_{\mathrm{dom}}(v)$ est le domaine de v, lorsque v est une relation.
- ran (image d'une relation): $I_{\mathrm{ran}}(v)$ est l'image de v, lorsque v est une relation.
- $\circ$ (composition): $I_\circ(v, v')$ est la composition de v et de v', si ces derniers sont des relations. Autrement dit, $x(v \circ v')z$ si et seulement s'il existe y tel que $x\ v'\ y$ et $y\ v\ z$. (Nous notons ici $x\ R\ y$ au lieu de $(x, y) \in R$).)
- $\lhd$ (restriction): $I_\lhd(v, v')$ est la restriction de la relation v' à l'ensemble v, c'est-à-dire qu'elle est la relation R telle que $x\ R\ y$ si et seulement si $x\ v'\ y$ et x appartient à l'ensemble v.
- $⩤$ (anti-restriction): $I_{⩤}(v, v')$ est la restriction de la relation v' au complémentaire de v dans v', c'est-à-dire la relation R telle que $x\ R\ y$ si et seulement si $x\ v'\ y$ et x n'appartient pas à l'ensemble v.
- $\oplus$ (surcharge fonctionnelle): $I_\oplus(v, v')$ est la relation qui à tout x dans $\mathrm{dom}\, v'$ associe les y tels que $x\ v'\ y$, et à tout x hors de $\mathrm{dom}\, v'$ associe les y tels que $x\ v\ y$.

- $\_^{-}$ (itération): $I_{\_^{-}}(v, v')$ n'est défini que lorsque v est une relation d'un ensemble E vers lui-même, et lorsque v' est un entier. Alors $I_{\_^{-}}(v, v')$ est $\underbrace{v \circ \ldots \circ v}_{v' \text{ fois}}$. Nous laissons cette opération non spécifiée si $v' = 0$.
- $\_^{+}$ (fermeture transitive): $I_{\_^{+}}(v)$ est la relation R telle que $x\ R\ y$ si et seulement s'il existe un nombre fini non nul n tel que $x\ R^n\ y$. (Autrement dit, qu'il existe $n+1$ éléments x_0, x_1, $\ldots$, x_n tels que $x_0 = x$, $x_0\ v\ x_1$, $x_1\ v\ x_2$, $\ldots$, $x_{n-1}\ v\ x_n$ et $x_n = y$. Ceci n'est défini que lorsque v est une relation d'un ensemble E vers lui-même.)
- .. (construction d'intervalles): $I_{..}(v, v')$ est l'intervalle de tous les entiers supérieurs ou égaux à v et inférieurs ou égaux à v'.
- # (cardinal): $I_{\#}(v)$ est le nombre d'éléments dans v, si v est un ensemble fini. Sinon, $I_{\#}(v)$ est n'importe quoi sauf un entier. (Ainsi, on peut caractériser les ensembles finis comme ceux dont le cardinal est entier.)

Les notations d'ensembles par compréhension sont une extension à la fonction d'interprétation: $\mathcal{E}[\![\{\ x_1 : E_1;\ \ldots;\ x_n : E_n \mid P \bullet e\ \}]\!] I_Z \rho$ est par définition l'ensemble de tous les éléments de la forme $\mathcal{E}[\![e]\!] I_Z(\rho \oplus \{[\![x_1]\!] \mapsto v_1, \ldots, [\![x_n]\!] \mapsto v_n\})$, lorsque v_1, $\ldots$, v_n parcourent les ensembles $\mathcal{E}[\![E_1]\!] I_Z\rho$, $\ldots$, $\mathcal{E}[\![E_n]\!] I_Z\rho$ respectivement de sorte que $\mathcal{E}[\![P]\!] I_Z(\rho \oplus \{[\![x_1]\!] \mapsto v_1, \ldots, [\![x_n]\!] \mapsto v_n\})$ soit vrai (égal à $\top$). Informellement, $\{\ x_1 : E_1;\ \ldots;\ x_n : E_n \mid P \bullet e\ \}$ est "l'ensemble des e tels que $x_1 : E_1$, $\ldots$, $x_n : E_n$ et P".

B.2 Arithmétique

Les symboles de prédicat sont:

- $\leq$: $I_{\leq}(v, v')$ est vrai ($\top$) si et seulement v est inférieur ou égal à v', lorsque ces derniers sont des entiers.
- $<$: $I_{<}(v, v')$ est vrai ($\top$) si et seulement v est strictement inférieur à v', lorsque ces derniers sont des entiers.
- $\geq$: $I_{\geq}(v, v')$ est vrai ($\top$) si et seulement v est supérieur ou égal à v', lorsque ces derniers sont des entiers.
- $>$: $I_{>}(v, v')$ est vrai ($\top$) si et seulement v est strictement supérieur à v', lorsque ces derniers sont des entiers.

On a une infinité dénombrable de symboles de constantes, $[\![0]\!]$, $[\![1]\!]$, $\ldots$ Pour tout entier naturel i, $I_{[\![i]\!]}$ est la constante entière i elle-même.

Les autres symboles de fonction sont:

- $+$ (addition): $I_{+}(v, v')$ est la somme de v et de v', si ces derniers sont des entiers.
- $-$ (soustraction): $I_{-}(v, v')$ est la différence entre v et v'. Elle est indéfinie si v ou v' n'est pas un entier, ou si v est strictement inférieur à v'.
- $*$ (multiplication): $I_{*}(v, v')$ est le produit de v et v', si ces derniers sont des entiers.
- div (quotient): $I_{\mathrm{div}}(v, v')$ est le quotient de la division euclidienne de v par v'.
- mod (reste): $I_{\mathrm{mod}}(v, v')$ est le reste de la division euclidienne de v par v'.

B.3 Suites

Une suite est ce qu'on appelle une liste en ML ou en Lisp. Il s'agit d'une suite finie $v_1, \ldots, v_n$ de valeurs. Une telle suite finie n'est rien d'autre qu'une fonction (partielle) qui à tout entier i entre 1 et n associe v_i. En particulier, si s est une expression dénotant une suite et i une expression dénotant un entier, alors $s\ i$ est une expression qui dénote le i-ième élément de la suite.

Les symboles de fonctions sont:

- seq (espace des suites): si v est un ensemble de valeurs, alors I_{seq} est l'ensemble de toutes les suites d'éléments de v.

- $\langle \_, \ldots, \_ \rangle$ (suite donnée par énumération, n-aire): $I_{\langle \_, \ldots, \_ \rangle}(v_1, \ldots, v_n)$ est la suite de n éléments, le premier étant v_1, le deuxième v_2, ..., le n-ième v_n. En particulier, $\langle\rangle$ est la suite vide et $\langle v \rangle$ est la suite qui ne contient que l'élément v.

- $\frown$ (concaténation): $I_{\frown}(v, v')$ est la concaténation des suites v et v'. C'est-à-dire que si v est la suite $v_1, \ldots, v_m$ et v' est la suite $v'_1, \ldots, v'_n$, alors $I_{\frown}(v, v')$ est la suite $v_1, \ldots, v_m, v'_1, \ldots, v'_n$.

- *tail* (queue de liste): $I_{tail}(v)$ est la suite de tous les éléments de v sauf le premier. Ceci n'est défini que si v est une suite (différente de $\langle\rangle$). Si v est vide, $I_{tail}(v)$ est par convention encore la suite vide.

On notera que l'on peut obtenir la tête d'une liste (son premier élément) en l'appliquant à 1. Aussi, si v est une suite, $I_{\#}(v)$ est non seulement le nombre d'associations (i, v_i) dans la suite, mais aussi la *longueur* de la suite v. On notera aussi que ran est la fonction qui retourne l'ensemble des éléments présents dans la suite donnée en argument.

C Règles de preuve et axiomes de Z

C.1 Définitions

- Règle [*Def*] (expansion des définitions). Si on a:

  $$Nom \mathrel{\widehat=} [decl_1;\ \ldots;\ decl_m \mid Q]$$

  alors, tout texte de schéma (*Schema-Text*) strictement plus bas dans la spécification et de la forme:

  $$d_1;\ \ldots;\ d_{i-1};\ Nom;\ d_{i+1};\ \ldots;\ d_k \mid P$$

  est remplaçable par:

  $$d_1;\ \ldots;\ d_{i-1};\ decl_1;\ \ldots;\ decl_m;\ d_{i+1};\ \ldots;\ d_k \mid Q \wedge P \quad \text{par } [Def](Nom)$$

C.2 Logique

On dit qu'une variable x est *liée* dans une expression e si et seulement si elle est dans la portée d'un $\forall\, x$, d'un $\exists\, x$, ou d'un $\{x : \ldots\}$. La variable x est *libre* sinon. (Une variable libre x dans e est un paramètre visible de l'expression e, et l'on peut remplacer toutes les occurrences libres de x par une nouvelle expression.)

$\begin{array}{lll} & \vdots & \\ (i) & F \Leftrightarrow G & [\ldots] \\ & \vdots & \\ (n+1) & F \Rightarrow G & \text{par } [\Leftrightarrow Elim_1](i) \\ \text{(resp.} & G \Rightarrow F & \text{par } [\Leftrightarrow Elim_2](i)) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & F \Rightarrow G & [\ldots] \\ & \vdots & \\ (j) & G \Rightarrow F & [\ldots] \\ & \vdots & \\ (n+1) & F \Leftrightarrow G & \text{par } [\Leftrightarrow Intro](i,j) \end{array}$
$\begin{array}{lll} & \vdots & \\ (i) & \forall S \bullet P & [\ldots] \\ & \vdots & \\ (j_1) & e_1 \in E_1 & [\ldots] \\ \vdots & \vdots & \\ (j_k) & e_k \in E_k & [\ldots] \\ & \vdots & \\ (n+1) & P[e_1/x_1, & \text{par } [\forall\, Elim](i, j_1, \\ & \quad \ldots, e_k/x_k] & \qquad \ldots, j_k) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & x_1 \in E_1 & [Hyp] \\ \vdots & \vdots & \vdots \\ (i+k-1) & x_k \in E_k & [Hyp] \\ & \vdots & \\ (n) & P & [\ldots] \\ (n+1) & \forall S \bullet P & \text{par } [\forall\, Intro](i, \ldots, \\ & & \qquad i+k-1, n) \end{array}$ (*)
$\begin{array}{lll} & \vdots & \\ (i) & \exists S \bullet Q & [\ldots] \\ & \vdots & \\ (j) & \forall S \bullet (Q \Rightarrow P) & [\ldots] \\ & \vdots & \\ (n) & P & \text{par } [\exists\, Elim](i,j) \end{array}$ (*)	$\begin{array}{lll} & \vdots & \\ (j_1) & e_1 \in E_1 & [\ldots] \\ \vdots & \vdots & \\ (j_k) & e_k \in E_k & [\ldots] \\ & \vdots & \\ (i) & P[e_1/x_1, \ldots, e_k/x_k] & [\ldots] \\ & \vdots & \\ (n+1) & \exists S \bullet P & \text{par } [\exists\, Intro](j_1, \\ & & \qquad \ldots, j_k, i) \end{array}$
$\begin{array}{lll} & \vdots & \\ (i) & \neg\, (\forall S \bullet P) & [\ldots] \\ & \vdots & \\ (n+1) & \exists S \bullet \neg\, P & \text{par } [\neg\, \forall\, Elim](i) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & \exists S \bullet \neg\, P & [\ldots] \\ & \vdots & \\ (n+1) & \neg\, (\forall S \bullet P) & \text{par } [\neg\, \forall\, Elim](i) \end{array}$
$\begin{array}{lll} & \vdots & \\ (i) & \neg\, (\exists S \bullet P) & [\ldots] \\ & \vdots & \\ (n+1) & \forall S \bullet \neg\, P & \text{par } [\neg\, \exists\, Elim](i) \end{array}$	$\begin{array}{lll} & \vdots & \\ (i) & \forall S \bullet \neg\, P & [\ldots] \\ & \vdots & \\ (n+1) & \neg\, (\exists S \bullet P) & \text{par } [\neg\, \exists\, Elim](i) \end{array}$

FIG. 7: Autres règles logiques

$\vdots$ $(n+1)$ $e = e$ par $[= \mathit{Refl}](e)$	$\vdots$ (i) $e = e'$ $[\ldots]$ $\vdots$ (j) $F[e/x, e'/x']$ $[\ldots]$ $\vdots$ $(n+1)$ $F[e'/x, e/x']$ par $[= \mathit{Subst}](j, i)$
$\vdots$ (i) $e \neq e'$ $[\ldots]$ $\vdots$ $(n+1)$ $\neg\,(e = e')$ par $[\neq \mathit{Elim}](i)$	$\vdots$ (i) $\neg\,(e = e')$ $[\ldots]$ $\vdots$ $(n+1)$ $e \neq e'$ par $[\neq \mathit{Intro}](i)$

FIG. 8: Règles d'égalité

On note $e[e_1/x_1, \ldots, e_n/x_n]$ le remplacement simultané dans l'expression e de toutes les occurrences libres de chaque x_i par e_i, $1 \leq i \leq n$. Ceci n'est correct que si aucune variable libre dans e_1, $\ldots$, e_n n'est liée dans e. (Sinon, par exemple, $(\forall\, x : \mathbb{N} \bullet P(y))[x/y]$ serait $\forall\, x : \mathbb{N} \bullet P(x)$, ce qui n'est pas le résultat escompté.)

On contourne la difficulté grâce à la règle d'*α-renommage*, qui permet de remplacer toute expression Z de la forme $Q\ x_1 : T_1, \ldots, x_k : T_k \mid P \bullet e$ (où Q peut être la compréhension d'ensembles, $\forall$ ou $\exists$) par:

$$Q\ x_1' : T_1, \ldots, x_k' : T_j \mid P[x_1'/x_1, \ldots, x_k'/x_k] \bullet e[x_1'/x_1, \ldots, x_k'/x_k] \quad \text{par } [\alpha]$$

où x_1', $\ldots$, x_k' sont k nouvelles variables, disjointes deux à deux. (Nouvelles signifiant n'apparaissant pas dans T_1, $\ldots$, T_k.) On appliquera cette règle sans la mentionner.

Les règles sur les quantificateurs sont données en figure 7, où dans chaque le texte de schéma S est $x_1 : E_1;\ \ldots;\ x_k : E_k$. Les conditions d'application de substitutions sont celles énoncées ci-dessus. Les règles marquées d'une astérisque $(*)$ ne peuvent être appliquées que lorsque x_1, $\ldots$, x_k sont des variables n'apparaissant pas libres avant l'étape (i). (Par exemple, pas du tout avant (i).)

Nous avons de plus les règles de la figure 8. La règle $[= \mathit{Subst}]$, où x et x' sont deux variables quelconques différentes, exprime que si $e = e'$, et si G est vrai, alors n'importe quelle formule obtenue à partir de G en remplaçant certaines occurrences de e par e' et certaines occurrences de e' par e est vraie aussi. (On identifie les occurrences de e et de e' dans G par l'astuce consistant à écrire que G s'écrit $F[e/x, e'/x']$ pour une certaine formule F.) On peut déduire que l'égalité est symétrique (si $e = e'$, alors $e' = e$) en posant pour F la formule $x = x'$, auquel cas $F[e/x, e'/x']$ est $e = e'$, et on déduit $F[e'/x, e/x']$, c'est-à-dire $e' = e$. On peut aussi déduire que l'égalité est transitive (si $e = e'$ et $e' = e''$, alors $e = e''$) en posant pour F la formule $x' = e''$, alors $F[e/x, e'/x']$ est $e' = e''$ qui est posée en hypothèse, et $F[e'/x, e/x']$ est $e = e''$, le résultat cherché.

C.3 Théorie des ensembles

Les axiomes sont donnés en figures 9, 10, 11, et 12.

L'axiome $[\mathit{Ext}]$ est appelé l'*axiome d'extensionnalité*, et exprime que deux ensembles sont égaux si et seulement s'ils ont les mêmes éléments.

$[\varnothing](e)$	$\neg\, e \in \varnothing$
$[\mathit{Ext}](e, e_1, e_2)$	$e_1 \in \mathbb{P}\, e \wedge e_2 \in \mathbb{P}\, e \Rightarrow$
	$\quad (\forall\, z : e \bullet z \in e_1 \Leftrightarrow z \in e_2) \Rightarrow e_1 = e_2$
$[\neg\, \varnothing](e, e_1)$	$e_1 \in e \Rightarrow e \neq \varnothing$
$[\notin](e_1, e_2)$	$e_1 \notin e_2 \Leftrightarrow \neg(e_1 \in e_2)$
$[\subseteq](e_1, e_2)$	$e_1 \subseteq e_2 \Leftrightarrow \forall\, z : e_1 \bullet z \in e_2$
$[\subset](e_1, e_2)$	$e_1 \subset e_2 \Leftrightarrow e_1 \subseteq e_2 \wedge e_1 \neq e_2$
$[\subseteq \mathit{Refl}](e)$	$e \subseteq e$
$[\subseteq \mathit{Sym}](e_1, e_2)$	$e_1 \subseteq e_2 \wedge e_2 \subseteq e_1 \Rightarrow e_1 = e_2$
$[\subseteq \mathit{Trans}](e_1, e_2, e_3)$	$e_1 \subseteq e_2 \wedge e_2 \subseteq e_2 \Rightarrow e_1 \subseteq e_3$
$[\varnothing \subseteq](e)$	$\varnothing \subseteq e$
$[\subset \mathit{Irrefl}](e)$	$\neg\, e \subset e$
$[\subset \mathit{Antisym}](e_1, e_2)$	$\neg\, (e_1 \subset e_2 \wedge e_2 \subset e_1)$
$[\subset \mathit{Trans}](e_1, e_2, e_3)$	$e_1 \subset e_2 \wedge e_2 \subset e_2 \Rightarrow e_1 \subset e_3$
$[\varnothing \subset](e)$	$\varnothing \subset e \Leftrightarrow e \neq \varnothing$
$[\mathit{Enum}._n](e, e_1, \ldots, e_n)$	$e \in \{e_1, \ldots, e_n\} \Leftrightarrow e = e_1 \vee \ldots \vee e = e_n$
$[\cup](e, e_1, e_2)$	$e \in e_1 \cup e_2 \Leftrightarrow e \in e_1 \vee e \in e_2$
$[\cup\varnothing](e)$	$e \cup \varnothing = e$
$[\varnothing\cup](e)$	$\varnothing \cup e = e$
$[\cup \mathit{Idemp}](e)$	$e \cup e = e$
$[\cup \mathit{Comm}](e_1, e_2)$	$e_1 \cup e_2 = e_2 \cup e_1$
$[\cup \mathit{Ass}](e_1, e_2, e_3)$	$(e_1 \cup e_2) \cup e_3 = e_1 \cup (e_2 \cup e_3)$
$[\cap](e, e_1, e_2)$	$e \in e_1 \cup e_2 \Leftrightarrow e \in e_1 \wedge e \in e_2$
$[\cap\varnothing](e)$	$e \cap \varnothing = \varnothing$
$[\varnothing\cap](e)$	$\varnothing \cap e = \varnothing$
$[\cap \mathit{Idemp}](e)$	$e \cap e = e$
$[\cap \mathit{Comm}](e_1, e_2)$	$e_1 \cap e_2 = e_2 \cap e_1$
$[\cap \mathit{Ass}](e_1, e_2, e_3)$	$(e_1 \cap e_2) \cap e_3 = e_1 \cap (e_2 \cap e_3)$
$[\cup \cap \mathit{DistD}](e_1, e_2, e_3)$	$e_1 \cup (e_2 \cap e_3) = (e_1 \cup e_2) \cap (e_1 \cup e_3)$
$[\cup \cap \mathit{DistG}](e_1, e_2, e_3)$	$(e_2 \cap e_3) \cup e_1 = (e_2 \cup e_1) \cap (e_3 \cup e_1)$
$[\cap \cup \mathit{DistD}](e_1, e_2, e_3)$	$e_1 \cap (e_2 \cup e_3) = (e_1 \cap e_2) \cup (e_1 \cap e_3)$
$[\cap \cup \mathit{DistG}](e_1, e_2, e_3)$	$(e_2 \cup e_3) \cap e_1 = (e_2 \cap e_1) \cup (e_3 \cap e_1)$
$[\setminus](e, e_1, e_2)$	$e \in e_1 \setminus e_2 \Leftrightarrow e \in e_1 \wedge \neg\, e \in e_2$
$[\setminus\varnothing](e)$	$e \setminus \varnothing = e$
$[\varnothing\setminus](e)$	$\varnothing \setminus e = \varnothing$
$[\cap \cup \setminus](e_1, e_2)$	$(e_1 \cap e_2) \cup (e_1 \setminus e_2) = e_1$
$[\mathit{Compl}](e_1, e_2)$	$(e_1 \setminus e_2) \cap e_2 = \varnothing$
$[\setminus\setminus_1](e_1, e_2, e_3)$	$e_1 \setminus (e_2 \setminus e_3) = (e_1 \setminus e_2) \cup (e_1 \cap e_3)$
$[\setminus\setminus_2](e_1, e_2, e_3)$	$(e_1 \setminus e_2) \setminus e_3 = e_1 \setminus (e_2 \cup e_3)$
$[\cup\setminus_1](e_1, e_2, e_3)$	$e_1 \cup (e_2 \setminus e_3) = (e_1 \cup e_2) \setminus (e_3 \setminus e_1)$
$[\cup\setminus_2](e_1, e_2, e_3)$	$(e_1 \cup e_2) \setminus e_3 = (e_1 \setminus e_3) \cup (e_2 \setminus e_3)$
$[\cap\setminus](e_1, e_2, e_3)$	$e_1 \cap (e_2 \setminus e_3) = (e_1 \cap e_2) \setminus e_3$
$[\setminus\cap](e_1, e_2, e_3)$	$e_1 \setminus (e_2 \cap e_3) = (e_1 \setminus e_2) \cup (e_1 \setminus e_3)$
$[\subseteq \cup_1](e_1, e_2)$	$e_1 \subseteq e_1 \cup e_2$
$[\subseteq \cup_2](e_1, e_2)$	$e_2 \subseteq e_1 \cup e_2$
$[\subseteq \cup_3](e_1, e_2, e_3)$	$e_1 \cup e_2 \subseteq e_3 \Leftrightarrow e_1 \subseteq e_3 \wedge e_2 \subseteq e_3$
$[\subseteq \cap_1](e_1, e_2)$	$e_1 \cap e_2 \subseteq e_1$
$[\subseteq \cap_2](e_1, e_2)$	$e_1 \cup e_2 \subseteq e_2$
$[\subseteq \cap_3](e_1, e_2, e_3)$	$e_3 \subseteq e_1 \cap e_2 \Leftrightarrow e_3 \subseteq e_1 \wedge e_3 \subseteq e_2$

FIG. 9: Axiomes ensemblistes (1)

$[\bigcup](e, e_1, e_2)$	$e_1 \subseteq \mathbb{P}\, e \Rightarrow (e_2 \in \bigcup e_1 \Leftrightarrow \exists\, x : \mathbb{P}\, e \bullet e_2 \in x \wedge x \in e_1)$
$[\bigcup \subseteq](e_1, e_2)$	$e_1 \subseteq \mathbb{P}\, e_2 \Rightarrow \bigcup e_1 \subseteq e_2$
$[\bigcup \varnothing]$	$\bigcup \varnothing = \varnothing$
$[\bigcup\{\_\}](e)$	$\bigcup\{e\} = e$
$[\bigcup \cup](e_1, e_2)$	$\bigcup(e_1 \cup e_2) = \bigcup e_1 \cup \bigcup e_2$
$[\bigcup \mathit{Mono}](e_1, e_2)$	$e_1 \subseteq e_2 \Rightarrow \bigcup e_1 \subseteq \bigcup e_2$
$[\subseteq \bigcup_1](e_1, e_2)$	$e_1 \in e_2 \Rightarrow e_1 \subseteq \bigcup e_2$
$[\subseteq \bigcup_2](e_1, e_2)$	$(\forall\, x : e_1 \bullet x \subseteq e_2) \Rightarrow \bigcup e_1 \subseteq e_2$
$[\bigcap](e, e_1, e_2)$	$e_1 \in \mathbb{P}\,\mathbb{P}\, e \wedge e_1 \neq \varnothing \Rightarrow (e_2 \in \bigcap e_1 \Leftrightarrow \forall\, x : \mathbb{P}\, e \bullet e_2 \in x \Rightarrow x \in e_1)$
$[\bigcap \subseteq](e_1, e_2)$	$e_1 \subseteq \mathbb{P}\, e_2 \wedge e_1 \neq \varnothing \Rightarrow \bigcap e_1 \subseteq e_2$
$[\bigcap\{\_\}](e)$	$\bigcap\{e\} = e$
$[\bigcap \cap](e_1, e_2)$	$\bigcap(e_1 \cap e_2) = \bigcap e_1 \cap \bigcap e_2$
$[\bigcap \mathit{Mono}](e_1, e_2)$	$e_1 \subseteq e_2 \Rightarrow \bigcap e_1 \subseteq \bigcap e_2$
$[\subseteq \bigcap_1](e_1, e_2)$	$e_1 \in e_2 \Rightarrow \bigcap e_2 \subseteq e_1$
$[\subseteq \bigcap_2]$	$(\forall\, x : e_1 \bullet e_2 \subseteq x) \Rightarrow \bigcap e_1 \subseteq x$
$[\subseteq \setminus_1](e_1, e_2)$	$e_1 \setminus e_2 \subseteq e_1$
$[\subseteq \setminus_2](e_1, e_2)$	$(e_1 \setminus e_2) \cap e_2 = \varnothing$
$[\subseteq \setminus_3](e_1, e_2, e_3)$	$e_3 \subseteq e_1 \wedge e_3 \cap e_2 = \emptyset \Rightarrow e_3 \subseteq e_1 \setminus e_2$
$[\mathit{Tuple}_n](e_1, \ldots, e'_n)$	$(e_1, \ldots, e_n) = (e'_1, \ldots, e'_n) \Leftrightarrow e_1 = e'_1 \wedge \ldots \wedge e_n = e'_n$
$[\times_n \mathit{Intro}](e_1, \ldots, e_n)$	$\forall\, x_1 : e_1, \ldots, x_n : e_n \bullet (x_1, \ldots, x_n) \in e_1 \times \ldots \times e_n$
$[\times_n \mathit{Elim}](e_1, \ldots, e_n)$	$\forall\, x : e_1 \times \ldots \times e_n \bullet \exists\, x_1 : e_1, \ldots, x_n : e_n \bullet x = (x_1, \ldots, x_n)$
$[\mathbb{P}](e_1, e_2)$	$e_1 \in \mathbb{P}\, e_2 \Leftrightarrow e_1 \subseteq e_2$
$[\mathbb{P}\, \mathit{Mono}](e_1, e_2)$	$e_1 \subseteq e_2 \Rightarrow \mathbb{P}\, e_1 \subseteq \mathbb{P}\, e_2$
$[\mathit{Comp}_1](e, P, e')$	$e \in \{\ \mathit{decls} \bullet e'\ \} \Leftrightarrow \exists\, \mathit{decls} \bullet e = e'$ (si aucune variable déclarée par *decls* n'est libre dans e)
$[\mathit{Comp}_2](P, e)$	$\forall\, x_1 : e_1;\ \ldots;\ x_k : e_k \bullet P \Rightarrow e \in \{\ x'_1 : e_1;\ \ldots;\ x'_k : e_k \mid P[x'_1/x_1, \ldots, x'_k/x_k] \bullet e[x'_1/x_1, \ldots, x'_k/x_k]\ \}$ ($x'_1, \ldots, x'_k$ non libres dans P ou e)
$[\mathit{Comp\ Type}](P, e, e')$	$(\forall\, \mathit{decls} \bullet P \Rightarrow e \in e') \Rightarrow \{\ \mathit{decls} \mid P \bullet e\ \} \in \mathbb{P}\, e'$ (si aucune variable déclarée par *decls* n'est libre dans e')
$[\leftrightarrow](e_1, e_2)$	$(e_1 \leftrightarrow e_2) = \mathbb{P}(e_1 \times e_2)$
$[\mathrel{+\!\!\!\!\rightarrow}](e_1, e_2)$	$(e_1 \mathrel{+\!\!\!\!\rightarrow} e_2) = \{\ f : e_1 \leftrightarrow e_2 \mid (\forall\, x : e_1;\ y, y' : e_2 \bullet (x, y) \in f \wedge (x, y') \in f \Rightarrow y = y') \bullet f\ \}$
$[\rightarrow](e_1, e_2)$	$(e_1 \rightarrow e_2) = \{\ f : e_1 \mathrel{+\!\!\!\!\rightarrow} e_2 \mid (\forall\, x : e_1 \bullet \exists\, y : e_2 \bullet (x, y) \in f) \bullet f\ \}$
$[\mathit{App\ Type}](e_1, e_2, e, e')$	$e \in (e_1 \mathrel{+\!\!\!\!\rightarrow} e_2) \wedge e' \in e_1 \Rightarrow e\ e' \in e_2$
$[\mathrel{+\!\!\!\!\rightarrow} \mathit{App}_1](e_1, e_2, e, e')$	$e \in (e_1 \mathrel{+\!\!\!\!\rightarrow} e_2) \wedge e' \in \operatorname{dom} e \Rightarrow (e', e\ e') \in e$
$[\mathrel{+\!\!\!\!\rightarrow} \mathit{App}_2](e_1, e_2, e, e', e'')$	$e \in (e_1 \mathrel{+\!\!\!\!\rightarrow} e_2) \wedge (e', e'') \in e \Rightarrow e' \in \operatorname{dom} e \wedge e'' = e\ e'$
$[\operatorname{dom}](e_1, e_2, e)$	$e \in (e_1 \leftrightarrow e_2) \Rightarrow \operatorname{dom} e = \{\ x : e_1;\ y : e_2 \mid (x, y) \in e \bullet x\ \}$
$[\operatorname{dom} \varnothing]$	$\operatorname{dom} \varnothing = \varnothing$
$[\operatorname{dom} \mathit{Enum}._n](e_1, \ldots, e'_n)$	$\operatorname{dom}\{(e_1, e'_1), \ldots, (e_n, e'_n)\} = \{e_1, \ldots, e_n\}$
$[\operatorname{dom} \leftrightarrow](e_1, e_2, e)$	$e \in (e_1 \leftrightarrow e_2) \Rightarrow \operatorname{dom} e \subseteq e_1$
$[\operatorname{dom} \rightarrow](e_1, e_2, e)$	$e \in (e_1 \rightarrow e_2) \Rightarrow \operatorname{dom} e = e_1$
$[\operatorname{dom} \cup](e_1, e_2)$	$\operatorname{dom}(e_1 \cup e_2) = \operatorname{dom} e_1 \cup \operatorname{dom} e_2$
$[\operatorname{dom} \cap](e_1, e_2)$	$\operatorname{dom}(e_1 \cap e_2) \subseteq \operatorname{dom} e_1 \cap \operatorname{dom} e_2$
$[\operatorname{dom} \mathit{Mono}](e_1, e_2)$	$e_1 \subseteq e_2 \Rightarrow \operatorname{dom} e_1 \subseteq \operatorname{dom} e_2$

FIG. 10: Axiomes ensemblistes (2)

$[\mathrm{ran}](e_1, e_2, e)$	$e \in (e_1 \leftrightarrow e_2) \Rightarrow \mathrm{ran}\, e = \{\ x : e_1;\ y : e_2 \mid (x, y) \in e \bullet y\ \}$
$[\mathrm{ran} \leftrightarrow](e_1, e_2, e)$	$e \in (e_1 \leftrightarrow e_2) \Rightarrow \mathrm{ran}\, e \subseteq e_2$
$[\mathrm{ran}\, \varnothing]$	$\mathrm{ran}\, \varnothing = \varnothing$
$[\mathrm{ran}\, \mathit{Enum}._n](e_1, \ldots, e'_n)$	$\mathrm{ran}\{(e_1, e'_1), \ldots, (e_n, e'_n)\} = \{e'_1, \ldots, e'_n\}$
$[\mathrm{ran} \cup](e_1, e_2)$	$\mathrm{ran}(e_1 \cup e_2) = \mathrm{ran}\, e_1 \cup \mathrm{ran}\, e_2$
$[\mathrm{ran} \cap](e_1, e_2)$	$\mathrm{ran}(e_1 \cap e_2) \subseteq \mathrm{ran}\, e_1 \cap \mathrm{ran}\, e_2$
$[\mathrm{ran}\, \mathit{Mono}](e_1, e_2)$	$e_1 \subseteq e_2 \Rightarrow \mathrm{ran}\, e_1 \subseteq \mathrm{ran}\, e_2$
$[\circ_1](e_1, e_2, e_3, e, e')$	$e \in (e_1 \leftrightarrow e_2) \wedge e' \in (e_2 \leftrightarrow e_3) \Rightarrow$ $e' \circ e = \{\ x : e_1;\ y : e_2;\ z : e_3 \mid (x, y) \in e \wedge (y, z) \in e' \bullet (x, z)\}$
$[\circ_2](e_1, e_2, e_3, e, e')$	$(e_1, e_2) \in e \wedge (e_2, e_3) \in e' \Rightarrow (e_1, e_3) \in e' \circ e$
$[\circ \mathit{App}](e_1, e_2, e_3)$	$(e_1 \circ e_2)(e_3) = e_1(e_2 e_3)$
$[\circ \mathit{Ass}](e_1, e_2, e_3)$	$(e_1 \circ e_2) \circ e_3 = e_1 \circ (e_2 \circ e_3)$
$[\mathit{id} \circ \mathit{id}](e_1, e_2)$	$\{\ x : e_1 \bullet (x, x)\ \} \circ \{\ y : e_2 \bullet (y, y)\ \} = \{\ z : e_1 \cap e_2 \bullet (z, z)\ \}$
$[\circ \mathit{id}](e_1, e_2)$	$e_1 \circ \{\ x : e_2 \bullet (x, x)\ \} = e_2 \lhd e_1$
$[\circ \mathit{Type}_1](e_1, e_2, e_3, e, e')$	$e \in (e_1 \leftrightarrow e_2) \wedge e' \in (e_2 \leftrightarrow e_3) \Rightarrow e' \circ e \in (e_1 \leftrightarrow e_3)$
$[\circ \mathit{Type}_2](e_1, e_2, e_3, e, e')$	$e \in (e_1 \nrightarrow e_2) \wedge e' \in (e_2 \nrightarrow e_3) \Rightarrow e' \circ e \in (e_1 \nrightarrow e_3)$
$[\circ \mathit{Type}_3](e_1, e_2, e_3, e, e')$	$e \in (e_1 \rightarrow e_2) \wedge e' \in (e_2 \rightarrow e_3) \Rightarrow e' \circ e \in (e_1 \rightarrow e_3)$
$[\lhd](e_1, e_2, e, e')$	$e' \in (e_1 \leftrightarrow e_2) \Rightarrow$ $e \lhd e' = \{\ x : e_1;\ y : e_2 \mid (x, y) \in e' \wedge x \in e \bullet (x, y)\ \}$
$[\lhd \mathit{Type}_1](e_1, e_2, e, e')$	$e \in (e_1 \leftrightarrow e_2) \wedge e' \in \mathbb{P}\, e_1 \Rightarrow e' \lhd e \in (e_1 \leftrightarrow e_2)$
$[\lhd \mathit{Type}_2](e_1, e_2, e, e')$	$e \in (e_1 \nrightarrow e_2) \wedge e' \in \mathbb{P}\, e_1 \Rightarrow e' \lhd e \in (e_1 \nrightarrow e_2)$
$[\lhd_1](e_1, e_2, e, e')$	$e' \in (e_1 \leftrightarrow e_2) \Rightarrow e \lhd e' = (e \times e_2) \cap e'$
$[\mathrm{dom} \lhd](e_1, e_2)$	$\mathrm{dom}(e_1 \lhd e_2) = e_1 \cap \mathrm{dom}\, e_2$
$[\lhd \subseteq](e_1, e_2)$	$e_1 \lhd e_2 \subseteq e_2$
$[\lhd\lhd](e_1, e_2, e_3)$	$e_1 \lhd (e_2 \lhd e_3) = (e_1 \cap e_2) \lhd e_3$
$[\lhd \mathit{Mono}._1](e_1, e_2, e_3)$	$e_1 \subseteq e_2 \Rightarrow e_1 \lhd e_3 \subseteq e_2 \lhd e_3$
$[\lhd \mathit{Mono}._2](e_1, e_2, e_3)$	$e_1 \subseteq e_2 \Rightarrow e_3 \lhd e_1 \subseteq e_3 \lhd e_2$
$[\mathbin{\lhd\!\!\!-}](e_1, e_2, e, e')$	$e' \in (e_1 \leftrightarrow e_2) \Rightarrow$ $e \mathbin{\lhd\!\!\!-} e' = \{\ x : e_1;\ y : e_2 \mid (x, y) \in e' \wedge \neg\, x \in e \bullet (x, y)\ \}$
$[\mathbin{\lhd\!\!\!-} \mathit{Type}_1](e_1, e_2, e, e')$	$e \in (e_1 \leftrightarrow e_2) \wedge e' \in \mathbb{P}\, e_1 \Rightarrow e' \mathbin{\lhd\!\!\!-} e \in (e_1 \leftrightarrow e_2)$
$[\mathbin{\lhd\!\!\!-} \mathit{Type}_2](e_1, e_2, e, e')$	$e \in (e_1 \nrightarrow e_2) \wedge e' \in \mathbb{P}\, e_1 \Rightarrow e' \mathbin{\lhd\!\!\!-} e \in (e_1 \nrightarrow e_2)$
$[\mathbin{\lhd\!\!\!-}_1](e_1, e_2, e, e')$	$e' \in (e_1 \leftrightarrow e_2) \Rightarrow e \mathbin{\lhd\!\!\!-} e' = (e_1 \setminus e) \lhd e'$
$[\mathbin{\lhd\!\!\!-} \subseteq](e_1, e_2)$	$e_1 \mathbin{\lhd\!\!\!-} e_2 \subseteq e_2$
$[\mathbin{\lhd\!\!\!-} \mathit{Antim}](e_1, e_2, e_3)$	$e_2 \subseteq e_1 \Rightarrow e_1 \mathbin{\lhd\!\!\!-} e_3 \subseteq e_2 \mathbin{\lhd\!\!\!-} e_3$
$[\mathbin{\lhd\!\!\!-} \mathit{Mono}](e_1, e_2, e_3)$	$e_1 \subseteq e_2 \Rightarrow e_3 \mathbin{\lhd\!\!\!-} e_1 \subseteq e_3 \mathbin{\lhd\!\!\!-} e_2$
$[\lhd \mathbin{\lhd\!\!\!-}](e_1, e_2)$	$(e_1 \lhd e_2) \cup (e_1 \mathbin{\lhd\!\!\!-} e_2) = e_2$
$[\oplus](e, e')$	$e \oplus e' = (\mathrm{dom}\, e' \mathbin{\lhd\!\!\!-} e) \cup e'$
$[\oplus \mathit{Type}_1](e_1, e_2, e, e')$	$e \in (e_1 \leftrightarrow e_2) \wedge e' \in (e_1 \leftrightarrow e_2) \Rightarrow e \oplus e' \in (e_1 \leftrightarrow e_2)$
$[\oplus \mathit{Type}_2](e_1, e_2, e, e')$	$e \in (e_1 \nrightarrow e_2) \wedge e' \in (e_1 \nrightarrow e_2) \Rightarrow e \oplus e' \in (e_1 \nrightarrow e_2)$
$[\oplus \mathit{Type}_3](e_1, e_2, e, e')$	$e \in (e_1 \nrightarrow e_2) \wedge e' \in (e_1 \rightarrow e_2) \Rightarrow e \oplus e' \in (e_1 \rightarrow e_2)$
$[\oplus \mathit{Type}_4](e_1, e_2, e, e')$	$e \in (e_1 \rightarrow e_2) \wedge e' \in (e_1 \nrightarrow e_2) \Rightarrow e \oplus e' \in (e_1 \rightarrow e_2)$
$[\oplus \mathit{Idemp}](e)$	$e \oplus e = e$
$[\oplus \mathit{Ass}](e_1, e_2, e_3)$	$(e_1 \oplus e_2) \oplus e_3 = e_1 \oplus (e_2 \oplus e_3)$
$[\varnothing \oplus](e)$	$\varnothing \oplus e = e$
$[\oplus \varnothing](e)$	$e \oplus \varnothing = e$
$[\oplus \cup](e_1, e_2)$	$\mathrm{dom}\, e_1 \cap \mathrm{dom}\, e_2 = \varnothing \Rightarrow e_1 \oplus e_2 = e_1 \cup e_2$

FIG. 11: Axiomes ensemblistes (3)

$[\oplus App_1](e_1, e_2, e_3)$	$e_3 \in \operatorname{dom} e_1 \setminus \operatorname{dom} e_2 \Rightarrow (e_1 \oplus e_2)e_3 = e_1\ e_3$
$[\oplus App_2](e_1, e_2, e_3)$	$e_3 \in \operatorname{dom} e_2 \Rightarrow (e_1 \oplus e_2)e_3 = e_2\ e_3$
$[\operatorname{dom} \oplus](e_1, e_2)$	$\operatorname{dom}(e_1 \oplus e_2) = \operatorname{dom} e_1 \cup \operatorname{dom} e_2$
$[\operatorname{ran} \oplus](e_1, e_2)$	$\operatorname{ran}(e_1 \oplus e_2) = \operatorname{ran}((\operatorname{dom} e_2) \lhd e_1) \cup \operatorname{ran} e_2$
$[\lhd\oplus](e_1, e_2, e_3)$	$e_1 \lhd (e_2 \oplus e_3) = (e_1 \lhd e_2) \oplus (e_1 \lhd e_3)$
$[Iter\ Type_1](e_1, e, e')$	$e \in (e_1 \leftrightarrow e_1) \wedge e' \in \mathbb{N} \wedge e' \neq 0 \Rightarrow e^{e'} \in (e_1 \leftrightarrow e_1)$
$[Iter\ Type_2](e_1, e, e')$	$e \in (e_1 \nrightarrow e_1) \wedge e' \in \mathbb{N} \wedge e' \neq 0 \Rightarrow e^{e'} \in (e_1 \nrightarrow e_1)$
$[Iter\ Type_3](e_1, e, e')$	$e \in (e_1 \rightarrow e_1) \wedge e' \in \mathbb{N} \wedge e' \neq 0 \Rightarrow e^{e'} \in (e_1 \rightarrow e_1)$
$[Iter1](e)$	$e^1 = e$
$[Iter+](e, e_1, e_2)$	$e^{e_1+e_2} = e^{e_1} \circ e^{e_2}$
$[Iter*](e, e_1, e_2)$	$e^{e_1*e_2} = (e^{e_1})^{e_2}$
$[Iter\ Comm_1](e_1, e_2, e_3)$	$e_1 \circ e_2 = e_2 \circ e_1 \Rightarrow (e_1 \circ e_2)^{e_3} = e_1^{e_3} \circ e_2^{e_3}$
$[Iter\ Comm_2](e_1, e_2, e_3, e_4)$	$e_1 \circ e_2 = e_2 \circ e_1 \Rightarrow e_1^{e_3} \circ e_2^{e_4} = e_2^{e_4} \circ e_1^{e_3}$
$[\_^+](e)$	$e^+ = \bigcup\{\ n : \mathbb{N} \mid n \neq 0 \bullet e^n\ \}$
$[\_^+ Type_1](e_1, e)$	$e \in (e_1 \leftrightarrow e_1) \Rightarrow e^+ \in (e_1 \leftrightarrow e_1)$
$[\_^+ Type_2](e_1, e)$	$e \in (e_1 \nrightarrow e_1) \Rightarrow e^+ \in (e_1 \nrightarrow e_1)$
$[\_^+ Type_3](e_1, e)$	$e \in (e_1 \rightarrow e_1) \Rightarrow e^+ \in (e_1 \rightarrow e_1)$
$[\subseteq \_^+](e)$	$e \subseteq e^+$
$[\_^+\circ_1](e)$	$e^+ \circ e \subseteq e^+$
$[\_^+\circ_2](e)$	$e \circ e^+ \subseteq e^+$
$[\_^+\_^+](e)$	$(e^+)^+ = e^+$
$[\_^+ Comm](e_1, e_2)$	$e_1 \circ e_2 = e_2 \circ e_1 \Rightarrow (e_1 \circ e_2)^+ = e_1^+ \circ e_2^+$
$[\_^+ Ind_1](e_1, e, e')$	$e' \circ e \subseteq e' \Rightarrow e^+ \subseteq e'$
$[\_^+ Ind_2](e_1, e, e')$	$e \circ e' \subseteq e' \Rightarrow e^+ \subseteq e'$
$[\_^+ Ind_3](e_1, e, e')$	$e \subseteq e' \wedge e' \circ e' \subseteq e' \Rightarrow e^+ \subseteq e'$
$[\_^+ Mono](e_1, e_2)$	$e_1 \subseteq e_2 \Rightarrow e_1^+ \subseteq e_2^+$
$[\#\varnothing]$	$\#\varnothing = 0$
$[\#\{\_\}](e)$	$\#\{e\} = 1$
$[\# \cup \{\_\}_1](e_1, e_2)$	$e_1 \in e_2 \Rightarrow \#(e_2 \cup \{e_1\}) = \#e_2$
$[\# \cup \{\_\}_2](e_1, e_2)$	$\neg\ e_1 \in e_2 \Rightarrow \#(e_2 \cup \{e_1\}) = \#e_2 + 1$
$[\#\cup](e_1, e_2)$	$\#(e_1 \cup e_2) = \#e_1 + \#e_2 - \#(e_1 \cap e_2)$
$[\#Enum](e_1, \ldots, e_n)$	$\#\{e_1, \ldots, e_n\} \in \mathbb{N} \wedge \#\{e_1, \ldots, e_n\} \leq n$
$[\#Seq](e, e_1)$	$e_1 \in \mathbb{P}\, e \wedge \#e_1 \in \mathbb{N} \Rightarrow \exists\, s : \operatorname{seq} e \bullet e_1 = \operatorname{ran} s$
$[..](e_1, e_2)$	$e_1 \,..\, e_2 = \{\ n : \mathbb{N} \mid e_1 \leq n \wedge n \leq e_2 \bullet n\ \}$
$[..Type](e_1, e_2)$	$e_1 \in \mathbb{N} \wedge e_2 \in \mathbb{N} \Rightarrow (e_1 \,..\, e_2) \in \mathbb{P}\mathbb{N}$
$[..\varnothing](e_1, e_2)$	$e_1 < e_2 \Rightarrow e_2 \,..\, e_1 = \varnothing$
$[..\{\_\}](e)$	$e \in \mathbb{N} \Rightarrow e \,..\, e = \{e\}$
$[..Mono](e_1, e_2, e_3, e_4)$	$e_1 \in \mathbb{N} \wedge e_2 \in \mathbb{N} \wedge e_3 \in \mathbb{N} \wedge e_4 \in \mathbb{N}$ $\quad \Rightarrow e_3 \leq e_1 \wedge e_2 \leq e_4 \Rightarrow (e_1 \,..\, e_2) \subseteq (e_3 \,..\, e_4)$
$[..=](e_1, e_2, e_3, e_4)$	$e_1 \in \mathbb{N} \wedge e_2 \in \mathbb{N} \wedge e_3 \in \mathbb{N} \wedge e_4 \in \mathbb{N}$ $\quad \Rightarrow e_1 \leq e_2 \wedge e_3 \leq e_4$ $\qquad \Rightarrow (e_1 \,..\, e_2) = (e_3 \,..\, e_4) \Rightarrow e_1 = e_3 \wedge e_2 = e_4$
$[\#..](e_1, e_2)$	$e_1 \in \mathbb{N} \wedge e_2 \in \mathbb{N} \Rightarrow e_1 \leq e_2 \Rightarrow \#(e_1 \,..\, e_2) = e_2 - e_1 + 1$

FIG. 12: Axiomes ensemblistes (4)

$$[+]\quad \begin{array}{cccc} 0+0=0 & 1+0=1 & 2+0=2 & \dots \\ 0+1=1 & 1+1=2 & 2+1=3 & \dots \\ 0+2=2 & 1+2=3 & 2+2=4 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array} \qquad [-]\quad \begin{array}{cccc} 0-0=0 & 1-0=1 & 2-0=2 & \dots \\ & 1-1=0 & 2-1=1 & \dots \\ & & 2-2=0 & \dots \\ & & & \ddots \end{array}$$

$$[*]\quad \begin{array}{cccc} 0*0=0 & 1*0=0 & 2*0=0 & \dots \\ 0*1=0 & 1*1=1 & 2*1=2 & \dots \\ 0*2=0 & 1*2=2 & 2*2=4 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array} \qquad [\mathrm{div}]\quad \begin{array}{cccc} 0 \,\mathrm{div}\, 1=0 & 1 \,\mathrm{div}\, 1=1 & 2 \,\mathrm{div}\, 1=2 & \dots \\ 0 \,\mathrm{div}\, 2=0 & 1 \,\mathrm{div}\, 2=0 & 2 \,\mathrm{div}\, 2=1 & \dots \\ 0 \,\mathrm{div}\, 3=0 & 1 \,\mathrm{div}\, 3=0 & 2 \,\mathrm{div}\, 3=0 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

$$[\neq \mathbb{N}]\quad \begin{array}{cccc} \neg\,(0\neq 0) & 1\neq 0 & 2\neq 0 & \dots \\ 0\neq 1 & \neg\,(1\neq 1) & 2\neq 1 & \dots \\ 0\neq 2 & 1\neq 2 & \neg\,(2\neq 2) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array} \qquad [\mathrm{mod}]\quad \begin{array}{cccc} 0 \,\mathrm{mod}\, 1=0 & 1 \,\mathrm{mod}\, 1=0 & 2 \,\mathrm{mod}\, 1=0 & \dots \\ 0 \,\mathrm{mod}\, 2=0 & 1 \,\mathrm{mod}\, 2=1 & 2 \,\mathrm{mod}\, 2=0 & \dots \\ 0 \,\mathrm{mod}\, 3=0 & 1 \,\mathrm{mod}\, 3=1 & 2 \,\mathrm{mod}\, 3=2 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{array}$$

$$[\leq]\quad \begin{array}{ccccc} 0\leq 0 & \neg\,1\leq 0 & \neg\,2\leq 0 & \neg\,3\leq 0 & \dots \\ 0\leq 1 & 1\leq 1 & \neg\,2\leq 1 & \neg\,3\leq 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} \qquad [<]\quad \begin{array}{ccccc} \neg\,0<0 & \neg\,1<0 & \neg\,2<0 & \neg\,3<0 & \dots \\ 0<1 & \neg\,1<1 & \neg\,2<1 & \neg\,3<1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

$$[\geq]\quad \begin{array}{ccccc} 0\geq 0 & 1\geq 0 & 2\geq 0 & 3\geq 0 & \dots \\ \neg\,0\geq 1 & 1\geq 1 & 2\geq 1 & 3\geq 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array} \qquad [>]\quad \begin{array}{ccccc} \neg\,0>0 & 1>0 & 2>0 & 3>0 & \dots \\ \neg\,0>1 & \neg\,1>1 & 2>1 & 3>1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

$$[0\mathbb{N}]\ 0\in\mathbb{N} \quad [1\mathbb{N}]\ 1\in\mathbb{N} \quad [2\mathbb{N}]\ 2\in\mathbb{N} \quad \dots$$

FIG. 13: Axiomes de l'arithmétique (1)

L'axiome (ou plutôt la collection d'axiomes) $[^+Ind._1]$ exprime le principe de récurrence (*induction* en anglais) sur la clôture transitive d'une relation. (Exercice: le redémontrer à partir de la définition $[\_^+](e)$, de l'axiome de récurrence sur les entiers $[\mathbb{N}Ind._1]$, des autres axiomes sur les entiers, et des autres axiomes sur les ensembles n'utilisant pas $\_^+$.) L'axiome $[^+Ind._1]$ exprime plus précisément que R^+ est la *plus petite relation* telle que $R^+ \circ R \subseteq R^+$. Les axiomes $[^+Ind._2]$ et $[^+Ind._3]$ sont similairement des axiomes de récurrence, et expriment aussi que R^+ est la plus petite relation vérifiant certaines conditions.

C.4 Arithmétique

Les axiomes sont ceux des figures 13 (ce ne sont que les tables d'addition, de multiplication, etc.), et 14.

L'axiome $[\mathbb{N}Ind._1]$ est l'*axiome de récurrence* sur les entiers, et est valable pour toute formule F et toute variable x. Typiquement, si F est de la forme $P(x)$, cet axiome s'écrit $P(0) \wedge (\forall\, n : \mathbb{N} \bullet P(n) \Rightarrow P(n+1)) \Rightarrow \forall\, n : \mathbb{N} \bullet P(n)$. Autrement dit, si une propriété est vrai de 0, et est préservée par incrémentation de son argument, alors elle est vraie de tous les entiers. L'axiome $[\mathbb{N}Ind._2]$ est l'axiome de récurrence généralisé, et l'axiome $[\mathbb{N}Ind._3]$ est une variante de ce dernier exprimant que s'il existe un entier vérifiant une propriété, alors il en existe un minimal.

$[+\mathit{Type}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 + e_2 \in \mathbb{N}$
$[0+](e)$	$0 + e = e$
$[+0](e)$	$e + 0 = e$
$[+=0](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 + e_2 = 0 \Rightarrow e_1 = 0 \land e_2 = 0$
$[+\mathit{Comm}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 + e_2 = e_2 + e_1$
$[+\mathit{Ass}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow (e_1 + e_2) + e_3 = e_1 + (e_2 + e_3)$
$[*\mathit{Type}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 * e_2 \in \mathbb{N}$
$[0*](e)$	$0 * e = 0$
$[*0](e)$	$e * 0 = 0$
$[1*](e)$	$e \in \mathbb{N} \Rightarrow 1 * e = e$
$[*1](e)$	$e \in \mathbb{N} \Rightarrow e * 1 = e$
$[*\mathit{Comm}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 * e_2 = e_2 * e_1$
$[*\mathit{Ass}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow (e_1 * e_2) * e_3 = e_1 * (e_2 * e_3)$
$[* + \mathit{DistD}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 * (e_2 + e_3) = e_1 * e_2 + e_1 * e_3$
$[* + \mathit{DistG}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow (e_1 + e_2) * e_3 = e_1 * e_3 + e_2 * e_3$
$[\leq +](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 \leq e_2 + e_3$
$[< \mathit{Def}](e_1, e_2)$	$e_1 < e_2 \Leftrightarrow e_1 \leq e_2 \land \neg\, e_1 = e_2$
$[-\mathit{Type}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 \leq e_2 \Rightarrow e_2 - e_1 \in \mathbb{N}$
$[-\mathit{Elim}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 \leq e_2 \Rightarrow e_1 + (e_2 - e_1) = e_2$
$[-\mathit{Intro}_1](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 + e_2 = e_3 \Rightarrow e_2 = e_3 - e_1$
$[-\mathit{Intro}_2](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 + e_2 = e_3 \Rightarrow e_1 = e_3 - e_2$
$[\mathrm{div}\, \mathit{Type}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 \;\mathrm{div}\; e_2 \in \mathbb{N}$
$[\mathrm{mod}\, \mathit{Type}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \Rightarrow e_1 \;\mathrm{mod}\; e_2 \in \mathbb{N}$
$[\mathrm{div}\ \mathrm{mod}\ \mathit{Elim}](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_2 \neq 0 \Rightarrow e_1 = e_2 * (e_1 \;\mathrm{div}\; e_2) + (e_1 \;\mathrm{mod}\; e_2)$
$[\mathrm{mod} \leq](e_1, e_2)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_2 \neq 0 \Rightarrow e_1 \;\mathrm{mod}\; e_2 \leq e_2 - 1$
$[\mathrm{div}\ \mathrm{mod}\ \mathit{Intro}](e_1, e_2, e_3, e_4)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \land e_4 \in \mathbb{N}$ $\Rightarrow e_4 < e_2 \land e_1 = e_2 * e_3 + e_4$ $\Rightarrow e_3 = e_1 \;\mathrm{div}\; e_2 \land e_4 = e_1 \;\mathrm{mod}\; e_2$
$[0 \leq](e)$	$e \in \mathbb{N} \Rightarrow 0 \leq e$
$[\leq \mathit{Refl}](e)$	$e \leq e$
$[\leq \mathit{Antisym}](e_1, e_2)$	$e_1 \leq e_2 \land e_2 \leq e_1 \Rightarrow e_1 = e_2$
$[\leq \mathit{Trans}](e_1, e_2, e_3)$	$e_1 \leq e_2 \land e_2 \leq e_3 \Rightarrow e_1 \leq e_3$
$[+ \leq \mathit{MonoG}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 \leq e_2 \Rightarrow e_3 + e_1 \leq e_3 + e_2$
$[+ \leq \mathit{MonoD}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 \leq e_2 \Rightarrow e_1 + e_3 \leq e_2 + e_3$
$[* \leq \mathit{MonoG}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 \leq e_2 \Rightarrow e_3 * e_1 \leq e_3 * e_2$
$[* \leq \mathit{MonoD}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 \leq e_2 \Rightarrow e_1 * e_3 \leq e_2 * e_3$
$[< \mathit{Irrefl}](e)$	$\neg\, e < e$
$[< \mathit{Antisym}](e_1, e_2)$	$\neg\, (e_1 < e_2 \land e_2 < e_1)$
$[< \mathit{Trans}](e_1, e_2, e_3)$	$e_1 < e_2 \land e_2 < e_3 \Rightarrow e_1 < e_3$
$[+ < \mathit{MonoG}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 < e_2 \Rightarrow e_3 + e_1 < e_3 + e_2$
$[+ < \mathit{MonoD}](e_1, e_2, e_3)$	$e_1 \in \mathbb{N} \land e_2 \in \mathbb{N} \land e_3 \in \mathbb{N} \Rightarrow e_1 < e_2 \Rightarrow e_1 + e_3 < e_2 + e_3$
$[\geq](e_1, e_2)$	$e_1 \geq e_2 \Leftrightarrow e_2 \leq e_1$
$[>](e_1, e_2)$	$e_1 > e_2 \Leftrightarrow e_2 < e_1$
$[\mathbb{N}\mathit{Ind}._1](F, x)$	$F[0/x] \land (\forall\, n : \mathbb{N} \bullet F[n/x] \Rightarrow F[n+1/x]) \Rightarrow \forall\, n : \mathbb{N} \bullet F[n/x]$
$[\mathbb{N}\mathit{Ind}._2](F, x)$	$(\forall\, n : \mathbb{N} \bullet (\forall\, m : \mathbb{N} \bullet m < n \Rightarrow F[m/x]) \Rightarrow F[n/x])$ $\Rightarrow \forall\, n : \mathbb{N} \bullet F[n/x]$
$[\mathbb{N}\mathit{Ind}._3](F, x)$	$(\exists\, n : \mathbb{N} \bullet F[n/x]) \Rightarrow$ $\exists\, n : \mathbb{N} \bullet F[n/x] \land \forall\, m : \mathbb{N} \bullet m < n \Rightarrow \neg\, F[m/x]$

Fig. 14: Axiomes de l'arithmétique (2)

$[\langle \_, \ldots, \_ \rangle](e_1, \ldots, e_n)$	$\langle e_1, \ldots, e_n \rangle = \{(1, e_1), \ldots, (n, e_n)\}$
$[\mathrm{seq}\ \mathit{Intro}](e, e_1, \ldots, e_n)$	$e_1 \in e \wedge \ldots \wedge e_n \in e \Rightarrow \langle e_1, \ldots, e_n \rangle \in \mathrm{seq}\ e$
$[\mathrm{seq}\ \mathit{Ind}._1](e_1, e_2)$	$\langle\rangle \in e_1 \wedge (\forall\, x : e_2;\ l : e_1 \bullet \langle x \rangle \frown l \in e_1) \Rightarrow \mathrm{seq}\ e_2 \subseteq e_1$
$[\mathrm{seq}\ \mathit{Ind}._2](e_1, e_2)$	$\langle\rangle \in e_1 \wedge (\forall\, x : e_2;\ l : e_1 \bullet l \frown \langle x \rangle \in e_1) \Rightarrow \mathrm{seq}\ e_2 \subseteq e_1$
$[\mathrm{seq}\ \mathit{Ind}._3](e_1, e_2)$	$\langle\rangle \in e_1 \wedge (\forall\, x : e_2 \bullet \langle x \rangle \in e_1) \wedge (\forall\, l, l' : e_1 \bullet l \frown l' \in e_1) \Rightarrow \mathrm{seq}\ e_2 \subseteq e_1$
$[\frown](e_1, e_2)$	$e_1 \frown e_2 = e_1 \cup \{\ n : \mathrm{dom}\, e_2 \mid (n + \#s, e_2\ n)\ \}$
$[\frown \mathit{Type}](e, e_1, e_2)$	$e_1 \in \mathrm{seq}\ e \wedge e_2 \in \mathrm{seq}\ e \Rightarrow e_1 \frown e_2 \in \mathrm{seq}\ e$
$[\langle \_, \ldots, \_ \rangle \frown](e_1, \ldots, e'_n)$	$\langle e_1, \ldots, e_m \rangle \frown \langle e'_1, \ldots, e'_n \rangle = \langle e_1, \ldots, e_m, e'_1, \ldots, e'_n \rangle$
$[\langle\rangle \frown](e_1, e_2)$	$e_1 \in \mathrm{seq}\ e_2 \Rightarrow \langle\rangle \frown e_1 = e_1$
$[\frown \langle\rangle](e)$	$e \frown \langle\rangle = e$
$[\frown \mathit{Ass}](e, e_1, e_2, e_3)$	$e_1 \in \mathrm{seq}\ e \wedge e_2 \in \mathrm{seq}\ e \wedge e_3 \in \mathrm{seq}\ e \Rightarrow (e_1 \frown e_2) \frown e_3 = e_1 \frown (e_2 \frown e_3)$
$[\# \frown](e, e_1, e_2)$	$e_1 \in \mathrm{seq}\ e \wedge e_2 \in \mathrm{seq}\ e \Rightarrow \#(e_1 \frown e_2) = \#e_1 + \#e_2$
$[\mathrm{dom}\ \mathit{Seq}](e, e_1)$	$e_1 \in \mathrm{seq}\ e \Rightarrow \mathrm{dom}\, e_1 = 1 \mathrel{..} \#e_1$
$[\mathit{tail}](e)$	$\mathit{tail}\ e = \{\ n : 1 \mathrel{..} \#e - 1 \bullet (n, e(n+1))\ \}$
$[\mathit{tail}\ \mathit{Type}](e, e_1)$	$e_1 \in \mathrm{seq}\ e \Rightarrow \mathit{tail}\ e_1 \in \mathrm{seq}\ e$
$[\frown \langle \_ \rangle (1)](e, e_1, e_2)$	$e_1 \in e \wedge e_2 \in \mathrm{seq}\ e \Rightarrow (\langle e_1 \rangle \frown e_2)(1) = e_1$
$[\frown (1)](e, e_1, e_2)$	$e_1 \in \mathrm{seq}\ e \wedge e_2 \in \mathrm{seq}\ e \wedge e_1 \neq \langle\rangle \Rightarrow (e_1 \frown e_2)(1) = e_1(1)$
$[\mathit{tail} \frown](e, e_1, e_2)$	$e_1 \in e \wedge e_2 \in \mathrm{seq}\ e \Rightarrow \mathit{tail}(\langle e_1 \rangle \frown e_2) = e_2$
$[\frown \mathit{tail}](e, e_1)$	$e_1 \in \mathrm{seq}\ e \wedge e_1 \neq \langle\rangle \Rightarrow e_1 = \langle e_1(1) \rangle \frown \mathit{tail}\ e_1$

FIG. 15: Axiomes sur les suites

C.5 Suites

Les axiomes sont ceux de la figure 15. On remarquera encore une fois la présence d'axiomes de récurrence.

D Différences avec Z

Notre version de Z est une version simplifiée de Z. Parfois, nous avons supprimé certaines constructions du langage que nous n'utilisions pas ($\mathbb{Z}$, $\frown/$, multi-ensembles par exemple), et parfois nous avons trahi la logique de Z pour en rendre la présentation plus simple. Les principales différences de la seconde sorte sont:

- Notre logique est la logique classique, plus un fragment de la théorie des ensembles de Zermelo-Frænkel. La logique de Z est plus complexe. Notamment, ici une valeur indéfinie signifie une valeur non spécifiée: par exemple $1 \mathrm{\ div\ } 0 = 1 \mathrm{\ div\ } 0$ est vrai par réflexivité de l'égalité, alors qu'en Z $1 \mathrm{\ div\ } 0 = 1 \mathrm{\ div\ } 0$ est faux (ce qui complique les règles de preuve).

- Pour nous, type égale ensemble. En Z, les types sont des ensembles particuliers, et tout ensemble définit un type sous-jacent particulier (le plus petit type contenant l'ensemble, au sens de l'inclusion ensembliste). Le type d'une expression Z est alors unique et est constructible par algorithme (pas chez nous), et certaines opérations sur les schémas (disjonction, conjonction, etc.) ne sont syntaxiquement acceptables que si les schémas en argument sont type-compatibles. Notre choix simplifie les règles de preuve, mais se paie par une plus grande difficulté à détecter des incohérences statiquement.

- Notre calcul des schémas est purement textuel, c'est un langage de macros. En Z, il y a quelques subtilités: si $S \mathrel{\widehat=} [x : A \nrightarrow B \mid P(x)]$, alors nous avons défini la négation de S comme $[x : A \nrightarrow B \mid \neg\, P(x)]$, mais en Z, S est en fait une abréviation de $[x : \mathbb{P}(A \times B) \mid x \in A \nrightarrow B \wedge P(x)]$, et $\neg\, S$ est alors $[x : \mathbb{P}(A \times B) \mid \neg\, (x \in A \nrightarrow B \wedge P(x))]$, ce qui est différent.

- Δ, Ξ sont des opérateurs s'appliquant à des schémas. En Z, Δ et Ξ sont juste des lettres, et dès qu'on écrit une définition de schéma S, il est implicitement écrit une définition de ΔS et une de ΞS.

- La condition d'adéquation lors du raffinement de données n'est pas mentionnée dans [Spi89].

Références

[Cou90] Patrick Cousot. Methods and logics for proving programs. In van Leeuwen [vL90], chapter 15, pages 841–993.

[Eme90] E. Allen Emerson. Temporal and modal logics. In van Leeuwen [vL90], chapter 16, pages 995–1072.

[KT90] Dexter Kozen and Jerzy Tiuryn. Logics of programs. In van Leeuwen [vL90], chapter 14, pages 789–840.

[Spi88] J.M. Spivey. *Understanding Z — A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[Spi89] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International, 1989.

[vL90] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science.* Elsevier Science Publishers b.v., 1990.