LE GÉNIE LOGICIEL

Table des matières

Introduction

Le génie logiciel est un domaine de recherche qui a été défini en 1968 sous le parrainage de l'OTAN. Il a pour objectif de répondre à un problème qui s'énonçait en deux constatations : d'une part les logiciels n'étaient pas fiables, d'autre part, il était incroyablement difficile de réaliser dans des délais prévus des logiciels satisfaisant leur cahier des charges.

L'importance d'une approche méthodologique dans le développement des logiciels est apparue avec la crise de l'industrie du logiciel à la fin des années 70 engendrée par :

- l'augmentation des coûts de production,
- la difficulté d'évolution,
- la non fiabilité,
- le non respect des spécifications,
- le non respect des délais.

Les exemples de défaillances dues à un développement mal mené sont nombreux :

- la sonde Mariner vers Vénus s'est perdue dans l'espace à cause d'une erreur de programme FORTRAN ;
- en 1981, le premier lancement de la navette spatiale européenne a été retardé de deux jours à cause d'un problème logiciel. La navette a d'ailleurs été lancée sans que l'on ait localisé exactement le problème (mais les symptômes étaient bien délimités) ;
- la SNCF a rencontré des difficultés importantes (et coûteuses pour le contribuable français) pour la mise en service du système Socrate.
- L'explosion d'Ariane 5, le 4 juin 1996, qui a coûté un demi-milliard de dollars (non assuré !), est due à une faute logicielle d'une composante dont le fonctionnement n'était pas indispensable durant le vol ;
- …

Ces problèmes s'expliquent (en partie) par l'augmentation de la complexité des logiciels avec la montée en puissance des performances du matériel et la multiplication des langages de programmation.

Pour remédier à cela, le génie logiciel doit offrir un (voire plusieurs selon les besoins) niveau(x) d'abstraction. Il doit modéliser notre univers trop complexe pour n'en utiliser que les caractéristiques qui nous intéressent. Il existe de nombreuses modélisations, plus ou moins abstraites selon leurs objectifs, mais toutes sont basées sur la logique. C'est donc la logique que nous étudierons dans un premier temps. Ensuite, nous survolerons une spécification qui s'appuie sur la logique du 1er ordre et sur la théorie naïve des ensembles. Il s'agit de la notation Z qui présente l'avantage de permettre le raffinement de la description des programmes vers des modèles très concrets.

La logique

La logique est un outil très puissant pour abstraire (et donc simplifier) un univers complexe à la fois par la profusion de détails (le plus souvent inutiles) qu'il contient et par les subtilités (engendrant fréquemment des ambiguïtés) qui le régissent. C'est donc intrinsèquement une excellente base pour toute modélisation.

De plus, on peut remarquer que la principale caractéristique des machines reste (jusqu'à aujourd'hui et pour encore longtemps) leur capacité à exécuter de manière répétitive et très organisée des tâches fondées sur la logique formelle et la logique formelle seule. Ce qui signifie donc que pour le génie logiciel –dont le but est l'étude des processus de conception de logiciels devant fonctionner sur ces machines– la logique est d'autant plus importante.

I. Algèbres de termes

Les termes (et donc les algèbres de termes) constituent la notion de base en logique et à fortiori en calcul des prédicats (également appelé logique du 1er ordre). En effet, comme nous le verrons au chapitre II.1.2 page 10, les formules grâce auxquelles nous pourrons établir des démonstrations sont toujours construites à partir de termes.

I.1. Syntaxe

Avant de chercher à savoir ce que peut représenter un terme, il est indispensable de définir exactement ce que l'on entend par « terme ». Il faut donc pour commencer donner à la notion de terme une définition syntaxique.

I.1.1. Signatures

Avant de pouvoir parler de termes, il faut fixer un cadre d'utilisation de cet outil. Ce cadre est posé par la donnée d'une signature.

I.1.1.1. Pour commencer

Donnons d'abord une définition minimale, mais suffisante pour ce qui nous intéresse, de ce qu'est une signature.

Définition
Une signature $(\mathcal{C},\mathcal{F})$ *est la donnée de deux familles de symboles :*
- $\mathcal{C}$ *une famille de symboles de constantes ;*
- $\mathcal{F}$ *une famille de symboles de fonctions d'arité* ≥ 1.

*L'*arité *d'une fonction est le nombre de paramètres requis par celle-ci.*

Une signature est donc l'ensemble des symboles autorisés pour former des mots appelés termes.

Exemple I.1 Signature pour modéliser une pile
Pour modéliser le fonctionnement d'une pile, il faut :
- un seul symbole de constante : ε
- un symbole de fonction unaire pour chaque lettre que l'on peut insérer dans la pile : $push_a$, $push_b$,...
- un symbole de fonction unaire pour dépiler : *pop*

Donc une signature possible serait (ε,$push_a$,$push_b$,...,*pop*)

I.1.1.2. Pour aller un peu plus loin…

Il est possible de donner une définition un peu plus complète de la signature. Cependant, dans le reste de ce dossier, nous ne nous servirons pas des notions supplémentaires introduites ici.

Définition :
Une signature *est une paire $(\mathcal{S},\mathcal{F})$ où :*
- *$\mathcal{S}$ est un ensemble non vide de sortes $(s_1,..,s_n,s)$;*
- *$\mathcal{F}$ est un ensemble non vide de symbole de fonctions, disjoint de $\mathcal{S}$.*

De plus, $\mathcal{F}$ est muni d'une fonction de typage *τ qui associe à chaque symbole de $\mathcal{F}$ une séquence non vide d'élément de $\mathcal{F}$.*
Si $\tau(f) = (s_1,..,s_n,s)$, on notera $f : s_1 \times ... \times s_n \rightarrow s$
$|f| = n$ est appelé arité *de f.*
$s_1 \times ... \times s_n$ est appelé domaine *de f.*
s est appelé codomaine *de f.*
Les fonctions d'arité 0 sont appelées constantes.

Dans cette définition, il est important de bien remarquer que les deux familles $\mathcal{C}$ et $\mathcal{F}$ de la première définition sont regroupées ici en une seule famille $\mathcal{F}$ puisque symboles de constantes et symboles de fonctions sont assimilés. Cette définition est intéressante car elle définit précisément la notion d'arité et de constante.

Exemple I.2 :
Une signature complètement définie pourrait être $(\mathcal{S},\mathcal{F})$ avec :
$\mathcal{S}$={bool,int}
$\mathcal{F}$ est l'ensemble des symboles

0 :		$\rightarrow$ int
vrai, faux :		$\rightarrow$ bool
+ :	int $\times$ int	$\rightarrow$ int
inverse :	int	$\rightarrow$ int
egal :	int $\times$ int	$\rightarrow$ bool
ou :	bool $\times$ bool	$\rightarrow$ bool
et :	bool $\times$ bool	$\rightarrow$ bool

La première définition est un cas particulier de cette seconde définition. En effet, si la famille $\mathcal{S}$ est réduite à un singleton, alors il est possible d'omettre $\mathcal{S}$ et τ puisqu'ils n'ont plus d'utilité (il n'y a plus d'ambiguïté sur la sorte des fonctions et de leur paramètre(s)). C'est justement dans ce cas que nous nous placerons pour la suite de ce dossier et c'est pourquoi nous utiliserons la première définition beaucoup plus simple (facile à comprendre et moins lourde à manipuler) que la seconde.

I.1.2. Termes

Maintenant que le cadre dans lequel on va travailler est posé (la signature), on va pouvoir construire les termes de façon récursive.

Définition :
Soient $\mathcal{S}(\mathcal{C},\mathcal{F})$ une signature
$\mathcal{V}$ un ensemble infini dénombrable de variables
*L'*ensemble des termes *$\mathcal{T}$ est le plus petit ensemble contenant les variables et les constantes ($\mathcal{T} = \mathcal{C} \cup \mathcal{V}$) stable par l'application des symboles de fonctions de $\mathcal{F}$ à des termes.*
Autrement dit, un terme *est un mot qu'on peut obtenir en appliquant (récursivement) un nombre fini de fois les règles :*
- *tout symbole de constante est un terme ;*
- *toute variable est un terme ;*
- *Si f est un symbole de fonction d'arité n et si t_1, …, t_n sont des termes, alors $f(t_1,…,t_n)$ est un terme.*

Rem : Dans la définition ci-dessus des parenthèses ont été utilisées pour appliquer les paramètres $t_1,…,t_n$ à *f*. En fait, il est démontrable que l'emploie de parenthèses n'est pas obligatoire (c'est pourquoi en toute rigueur elles ne font pas parties de l'alphabet nécessaire pour construire les termes) cependant elle facilite grandement la lecture des termes et sont donc fréquemment utilisées en pratique.
De même pour une question de facilité de lecture, on préfère généralement la notation infixe (non prévue dans la définition) à la notation préfixe pour les fonctions binaires (par exemple *x+2* au lieu de *+(x,2)*). Dans ce cas, il est nécessaire d'utiliser des parenthèses.

Exemple I.3

Reprenons la signature de **Exemple I.1** (page 2) que nous avions construit pour modéliser le fonctionnement d'une pile.
Essayons de construire quelques termes avec cette signature (ε,$push_a$,$push_b$,...,pop) :

- ε est un terme évident (tout symbole de constante est un terme) ;
- $pop\ push_a\ push_b\ push_b\ push_a\ pop\ push_b\ push_a\ \varepsilon$ est un terme plus complexe (construit par récurrence) ;
- $pop\ pop\ pop\ \varepsilon$ est aussi un terme correct. Cela peut paraître étrange puisque, même si nous n'avons pas encore parlé d'interprétation de termes (voir le chapitre I.2 page 6), on comprend intuitivement que ce terme ne simule pas comme nous le voulions le fonctionnement correct d'une pile (nous verrons dans **Exemple I.11** page 7 comment remédier à ce problème) ;
- x est un autre terme évident (toute variable est un terme) mais dont on comprend également mal la signification.

Voyons aussi ce que pourrait être des termes mal formés :

- pop est un terme incorrect puisqu'un symbole de fonction (donc d'arité > 1) seul n'est pas un terme ;
- $pop\ \varepsilon\ \varepsilon$ n'est pas plus valide car pop est un symbole de fonction unaire, donc $pop\ \varepsilon\ \varepsilon$ est la concaténation des termes $pop\ \varepsilon$ et ε, mais la concaténation de termes n'est pas un terme.

Définition :
On appelle terme clos *un terme qui ne contient pas de variable*

Exemple I.4

Dans l'exemple précédent, parmi les termes valides seul x n'est pas un terme clos.

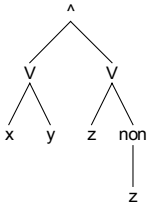
Bien que nous ayons définit un terme comme étant un *mot* composé par des variable, des symboles de constantes et des symboles de fonctions, il existe une autre notation totalement équivalente et parfois plus pratique : la représentation par arbre étiqueté.

Définition :
Un terme *est un* arbre étiqueté *tel que :*

- *tout arbre réduit à sa racine et étiqueté par un symbole de constante est un terme ;*
- *tout arbre réduit à sa racine et étiqueté par une variable est un terme ;*
- *Si f est un symbole de fonction d'arité n et si t_1, ..., t_n sont des termes, alors l'arbre de racine étiquetée par f et ayant n fils dont les sous-arbre associées sont t_1, ..., t_n est un terme.*

Exemple I.5

Soient la signature ($\bot$,non,$\wedge$,$\vee$) et les variables x, y et z
Un terme pour une telle signature est $(x \vee y) \wedge (z \vee non(z))$ c'est à dire :



Les deux définitions que nous avons données d'un terme se basent sur une récurrence. Il est possible de construire des ensembles de termes en suivant cette récurrence :

- On pose $\mathcal{T}_0 = \mathcal{C} \cup \mathcal{V}$ (avec $\mathcal{C}$ l'ensemble de symboles de constantes et $\mathcal{V}$ celui des variables)
- Pour chaque $k \in \mathbb{N}$, $\mathcal{T}_{k+1} = \mathcal{T}_k \cup \bigcup_{n \in \mathbb{N}^*} \{f(t_1,...,t_n) \mid f \in \mathcal{T}_k, t_1 \in \mathcal{T}_k, \ldots, t_n \in \mathcal{T}_k\}$

On a alors : $\mathcal{T} = \bigcup_{k \in \mathbb{N}^*} \mathcal{T}_k$

Définition :
La hauteur *d'un terme t est le plus petit entier k tel que $t \in \mathcal{T}_k$.*
Autrement dit, la hauteur *d'un terme est la hauteur de l'arbre étiqueté équivalent.*

Exemple I.6

Soient la signature ($\bot$,non,$\wedge$,$\vee$) et les variables x, y et z
Le terme $(x \vee y) \wedge (z \vee non(z))$ a pour hauteur 3 ce qui se voit aisément sur l'arbre de Exemple I.5 (page 4).

Rem : Pour une signature donnée, il y a toujours des termes de hauteur nulle (i.e. : on a toujours $\mathcal{T}_0 \neq \emptyset$) car $\mathcal{V}$ est un ensemble infini dénombrable donc il est toujours possible de former des termes constitués d'une seule variable. Cependant $\mathcal{F}$ (l'ensemble des fonctions d'arité ≥ 1) peut être nul donc, il peut ne pas y avoir de termes de hauteur non nulle.

I.1.3. Substitutions

Pour de multiples raisons (souvent d'ordre sémantique), il est intéressant de modifier un terme en remplaçant une ou plusieurs variables par d'autre(s) terme(s) (ou encore, en *substituant* à une ou plusieurs variables autant de terme(s)). Pour cela on définit une opération syntaxique : la substitution.

Définition :

Soient : - $k \in \mathbb{N}^*$
- $x_1, \ldots, x_k$ *des variables deux à deux distinctes*
- $t, t_1, \ldots, t_k$ *des termes*

On définit le mot $t[t_1/x_1,\ldots,t_k/x_k]$ *comme le résultat de la* substitution *des termes* $t_1,\ldots,t_k$ *aux variables* $x_1,\ldots,x_k$, *respectivement, dans toutes les occurrences de celles-ci dans le terme t, par récurrence :*

- *si t est un symbole de constante ou une variable autre que* $x_1,\ldots,x_k$, *alors :* $t[t_1/x_1,\ldots,t_k/x_k]=t$;
- *si* $t = x_i$ $(1 \le i \le k)$, *alors :* $t[t_1/x_1,\ldots,t_i/x_i,\ldots,t_k/x_k]=t_i$;
- *si* $t=f(u_1,\ldots,u_n)$ *(f une fonction d'arité* $n \ge 1$ *et* $u_1,\ldots,u_n$ *des termes), alors :*
  $t[t_1/x_1,\ldots,t_k/x_k]=f(u_1[t_1/x_1,\ldots,t_k/x_k],\ldots,u_n[t_1/x_1,\ldots,t_k/x_k])$.

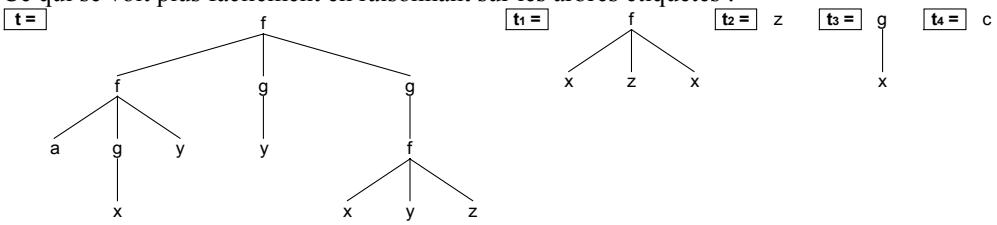
Exemple I.7

Soit la signature $(\mathcal{C},\mathcal{F})$ avec $\mathcal{C}=\{a,b,c\}$ et $\mathcal{F}=\{f,g\}$ (g unaire et f d'arité 3)

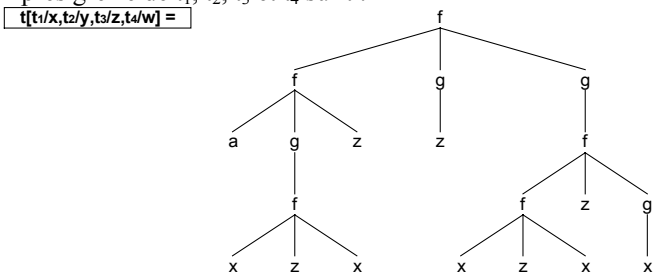
Soient les termes : - $t=f(f(a,g(x),y),g(y),g(f(x,y,z)))$
- $t_1=f(x,z,x)$, $t_2=z$, $t_3=g(x)$ et $t_4=c$

Alors $t[t_1/x,t_2/y,t_3/z,t_4/w]=f(f(a,g(f(x,z,x)),z),g(z),g(f(f(x,z,x),z,g(x))))$

Ce qui se voit plus facilement en raisonnant sur les arbres étiquetés :



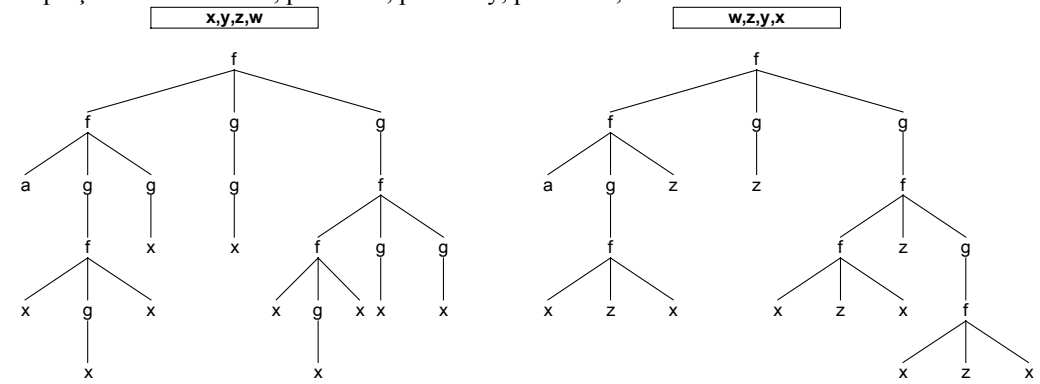
Après greffe de t_1, t_2, t_3 et t_4 sur t :



Il est important de noter que les substitutions se font simultanément (c'est à dire en parallèle). Sans quoi le résultat d'une opération de substitution serait différent suivant l'ordre dans lequel les variables sont remplacées.

Exemple I.8

Si on reprend la substitution de Exemple I.7 page 5, en remplaçant d'abord les x, puis les y, puis les z, puis les w ou en remplaçant d'abord les w, puis les z, puis les y, puis les x, on obtient deux résultats différents :



Bien qu'il ne soit pas impossible qu'il existe un ordre dans lequel si on remplace les variables séquentiellement on obtient le même résultat qu'avec une substitution en parallèle, ce n'est généralement pas le cas. Ainsi on voit que les deux résultats ci-dessus sont faux.

I.2. Sémantique

La définition syntaxique des termes était nécessaire puisque c'est elle qui permet de construire des mots corrects. Cependant, une fois ceci fait, il est surtout intéressant de donner un sens (et ainsi une utilité) à ces mots. Il faut donc donner une définition sémantique aux termes.

I.2.1. Interprétations d'algèbres de termes

Il est possible d'associer à un même terme plusieurs sens différents. On formalise cette notion intuitive en introduisant les interprétations.

Définition :
Soit une signature $\mathcal{S}(\mathcal{C},\mathcal{F})$
Une interprétation *$\mathcal{I}$ de l'algèbre de termes $\mathcal{A}$ basée sur $\mathcal{S}$ est la donnée de :*
- *un domaine $\mathcal{D}$ non vide ;*
- *un élément $c_i \in \mathcal{D}$ pour chaque symbole de constante i de $\mathcal{C}$;*
- *une fonction $F_f : \mathcal{D}^n \to \mathcal{D}$ pour chaque symbole de fonction f de $\mathcal{F}$.*

Exemple I.9
Soit la signature $\mathcal{S}(\mathcal{C},\mathcal{F})$ avec $\mathcal{C}=\{\varepsilon\}$ et $\mathcal{F}=\{S_a,S_b\}$ telle que S_a et S_b soient des symboles de fonction unaire.
Soit $\mathcal{A}$ l'algèbre basée sur $\mathcal{S}$.
Il existe une infinité d'interprétations pour $\mathcal{A}$ (qui est pourtant une algèbre très simple).
Par exemple :
- $\mathcal{I}_1$: $\mathcal{D}=\aleph$, $\varepsilon=0$ (le zéro des entiers), $S_a : x \to 2.x$, $S_b : x \to 2.x+1$;
- $\mathcal{I}_2$: $\mathcal{D}=\aleph$, $\varepsilon=0$ (le zéro des entiers), $S_a : x \to 2.x+1$, $S_b : x \to 2.x+2$;
- $\mathcal{I}_3$: $\mathcal{D}=\Re$, $\varepsilon=0$ (le zéro des entiers), $S_a : x \to 2.x+1$, $S_b : x \to 1/x$;
- $\mathcal{I}_4$: $\mathcal{D}$=un repère cartésien, ε=O (l'origine),
  S_a : translation de vecteur (1,1,1), S_b : rotation d'angle 90° autour de O ;
- …

I.2.2. Valeur d'un terme

Nous avons maintenant tous les outils nécessaires pour donner un sens à un terme *t*. C'est à dire pour lui faire correspondre une valeur. On note *val(t,I)* cette valeur car celle-ci dépend de l'interprétation que l'on s'est fixé.

On construit cette valeur de façon récursive :

- Si $t=i$ avec i un symbole de constante, alors $val(t,\mathcal{I})=c_i$ où c_i est l'élément de $\mathcal{D}$ associé à i dans $\mathcal{I}$
- Si $t=f(t_1,\ldots,t_k)$ avec $\begin{cases} x_1 \cdots x_n & \text{les variables de } f(t_1,\ldots,t_k) \\ x_1^1 \cdots x_{n_1}^1 & \text{les variables de } t_1 \\ \ldots \quad \ldots & \ldots \\ x_1^k \cdots x_{n_k}^k & \text{les variables de } t_k \end{cases}$ $(\sum_{i=1}^{k} n_i = n)$,

  alors $val(t,\mathcal{I})$ est une fonction de $\mathcal{D}^n$ dans $\mathcal{D}$

  et $\forall a_1,..,a_n \in \mathcal{D}$, $val(t,\mathcal{I})(a_1,\ldots,a_n) = F_f(val(t_1,\mathcal{I})(a_1^1,\ldots,a_{n_1}^1), \ldots, val(t_k,\mathcal{I})(a_1^k,\ldots,a_{n_k}^k))$

Rem : Si n=0, c'est à dire si t ne contient aucune variable (donc $t_1,\ldots,t_k$ non plus) (t est clos),
alors $val(t,\mathcal{I})$ est un élément de $\mathcal{D}$ et $val(t,\mathcal{I}) = F_f(val(t_1,\mathcal{I}), \ldots, val(t_k,\mathcal{I}))$

Exemple I.10
Reprenons **Exemple I.9** (page 6) où nous avons définit plusieurs interprétations possibles pour une algèbre $\mathcal{A}$.
Soit $t=S_a(S_a(S_b(S_b(S_b(S_b(\varepsilon))))))$ un terme de cette algèbre.
Alors, $val(t,\mathcal{I}_1) = 60$. En effet, t est clos donc $val(t,\mathcal{I}_1) \in \aleph$ et on a par récurrence : $S_a(S_a(S_b(S_b(S_b(S_b(\underbrace{\varepsilon}_{0}))))))$

$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}_{60}$
$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}_{30}$
$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}_{15}$
$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}_{7}$
$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}_{3}$
$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}_{1}$
$\underbrace{\quad\quad\quad\quad\quad\quad\quad\quad}_{0}$

De même, val(t,I_2)=123 car $S_a(S_a(S_b(S_b(S_b(S_b(\underbrace{\varepsilon}_{0})))))_{2})_{6})_{14})_{30})_{61})_{123}$

Soit u=$S_a(S_a(S_b(S_b(S_b(S_b(x))))))$ un autre terme de cette algèbre.

Le terme u n'est pas clos donc val(u,I_1) : $\aleph \to \aleph$ et val(u,I_1)(x)=64x+60 car $S_a(S_a(S_b(S_b(S_b(\underbrace{S_b(x)}_{2x+1})))))_{4x+3})_{8x+7})_{16x+15})_{32x+30})_{64x+60}$

Exemple I.11

Reprenons **Exemple I.3** (page 4). Nous y avions construit quelques termes appartenant à la signature $\mathcal{S}$: (ε,$push_a$, $push_b$,...,*pop*) censée simuler le fonctionnement d'une pile. Nous avions remarqué à cette occasion que certains de ces termes ne semblaient pas correspondre à un fonctionnement correct d'une pile. Essayons dans cet exemple de préciser cette intuition et de remédier à ce 'disfonctionnement'.

Posons l'interprétation I telle que : - $\mathcal{D}$ = les mots de l'alphabet
- ε = le mot vide
- $push_\alpha$: u→uα (la fonction α qui ajoute en fin de pile) avec α lettre de l'alphabet
- *pop* la fonction partielle qui enlève la tête de la pile

Grâce à I (qui est l'interprétation la plus intuitive que l'on puisse faire), il est possible faire correspondre à tout fonctionnement de la pile, entre les instants 0 et t, un terme u en composant au-dessus du symbole de constante ε les symboles de fonctions naturellement associées aux opérations de la pile. Ainsi l'opération qui consiste à « empiler a, empiler b, dépiler a, empiler a, empiler b, empiler b, empiler a et enfin dépiler b » correspond au terme $pop\ push_a\ push_b\ push_b\ push_a\ pop\ push_b\ push_a\ \varepsilon$ comme nous l'avions déjà pressenti dans **Exemple I.3**.

Maintenant nous aimerions aussi que chaque terme de $\mathcal{S}$ corresponde à un fonctionnement correct d'une pile. Cependant, I ne nous permet pas de réaliser ce souhait. Par exemple $pop\ pop\ pop\ \varepsilon$ correspond à un fonctionnement où la pile vide est dépilée trois fois ce qui est absurde. Il faudrait donc interdire ce terme ainsi que tous les termes dont le nombre de *pop* en partant du ε est supérieur au nombre de $push_\alpha$. Le problème, c'est que justement, nous voulions que tous les termes soient interprétables.

Pour rendre cela possible, nous allons légèrement modifier I. Nous allons donc compléter I en ajoutant « undef » au domaine et tous les termes qui s'évalueront à undef seront des termes simulent un fonctionnement incorrect de pile.

Ce qui nous donne I^+ : - $\mathcal{D}$ = les mots de l'alphabet ∪ undef
- ε = le mot vide
- $push_\alpha$: $\begin{cases} u \to ua \\ \text{undef} \to \text{undef} \end{cases}$
- *pop* : $\begin{cases} au \to u \\ \varepsilon \to \text{undef} \\ \text{undef} \to \text{undef} \end{cases}$

Cette fois, tous les termes sont interprétables :
- $pop\ push_a\ push_b\ push_b\ push_a\ pop\ push_b\ push_a\ \varepsilon$ correspond toujours à « empiler a, empiler b, dépiler a, empiler a, empiler b, empiler b, empiler a et enfin dépiler b » et s'évalue à « abba » ce qui est bien le contenu de la pile après ces opérations ;
- $pop\ pop\ pop\ \varepsilon$ ne correspond plus à rien et s'évalue à « undef » ce qui montre bien que ce terme ne représente pas un fonctionnement correct d'une pile.

Rem : Dans l'exemple précédent, il est remarquable que le temps soit internalisé dans un terme. En effet, l'interprétation d'un terme u donne l'état de la pile à l'instant t (avec t = hauteur de u) et l'interprétation des sous-termes de u donne les états intermédiaires.

II. Logique du 1er ordre

Le travail de base du logicien est d'examiner des structures, d'énoncer des propriétés à leur sujet et de vérifier si celles-ci sont valides ou non. Le calcul des prédicats, ou logique du 1er ordre, est la première étape (ce qui signifie donc qu'il y en a d'autres, mais nous ne nous y intéresserons pas) dans la formalisation de cette activité et par extension dans la formalisation des problèmes du génie logiciel.

Tout comme pour l'algèbre de terme, le calcul des prédicats comporte deux volets. D'abord, on se donne les outils nécessaires pour nommer les objets sur lesquels on va travailler (ce sont les termes que nous avons vus au précédent chapitre) et pour écrire (certaines de) leurs propriétés (ce sont les formules, sujet du chapitre II.1 Syntaxe qui suit). Ensuite, on étudie la satisfaction de ces propriétés dans les structures considérées (chapitre II.2 Sémantique page 16). Autrement dit, nous allons, encore une fois, commencer par présenter les aspects syntaxiques de ce domaine puis nous nous intéresserons à sa sémantique.

Rem : Il n'est pas possible d'énoncer toutes les propriétés imaginables pour un objet dans la logique du 1er ordre, c'est pour cela qu'il existe des logiques d'ordre supérieur. Cependant, en ce qui nous concerne cela sera suffisant.

II.1. Syntaxe

II.1.1. Langages du 1er ordre

A l'image de ce que nous avons fait pour les algèbres de termes, il faut définir précisément le cadre dans lequel les formules vont être utilisées. Ce cadre est ce que l'on appelle un langage du 1er ordre car il définit en quelque sorte le vocabulaire qu'on pourra utiliser dans les formules.

II.1.1.1. Signatures

Les langages (nous omettrons de préciser « du 1er ordre » dans le reste de ce dossier) sont toujours constitués de deux parties : une partie commune à tous les langages et une partie, variant d'un langage à l'autre, que l'on appelle sa signature.

Définition
La signature $\mathcal{S}$ *d'un langage est la donnée de :*
- $\mathcal{C}$ *ensemble de* symboles de constante ;
- $(\mathcal{F}_n)_{n\in\mathbb{N}^*}$ *une série d'ensembles des* symboles de fonction *(ou* fonctionnels*) d'*arité *n ;*
- $(\mathcal{R}_n)_{n\in\mathbb{N}}$ *une série d'ensembles des* symboles de relation *(ou* de prédicat *ou* relationnels*) d'*arité *n.*

Ces ensembles sont deux à deux disjoints.
On note $\mathcal{F}$ *l'ensemble de tous les fonctionnels et* $\mathcal{R}$ *celui de tous les relationnels.*
On appelle parfois symboles non logiques *les symboles de constante, de fonction et de relation.*

Rem : On utilise le même symbole $\mathcal{S}$ pour désigner à la fois la signature d'une algèbre de termes et celle d'un langage. Cependant cela n'est pas gênant car en fait la signature d'une algèbre de termes est toujours contenue dans celle d'un langage, si bien que nous ne parlerons plus que de signature de langage (sauf mention explicite du contraire).

Il existe une infinité de signatures, donc également une infinité de langages.

On remarque que par rapport à la définition de la signature d'une algèbre de termes, nous avons juste rajouté, dans la signature d'un langage, une famille de symboles de relation. En fait, on peut déjà dire (nous formaliserons cela plus tard) que tandis que les symboles de fonction servent à construire les termes, les symboles de relation, eux, sont utilisés pour fabriquer les formules.

Il est possible d'admettre dans la définition de la signature deux symboles de prédicat particuliers d'arité 0 : $\bot$ (« faux ») et $\top$ (« vrai »). Cependant cela n'est pas obligatoire car ces symboles sont toujours simulables par des formules toujours fausses ou toujours vraies. Nous considérerons quand même que nos signatures posséderont toujours ces symboles car ils permettent d'écrire le faux et le vrai de manière canonique.

II.1.1.2. Langages

Définition
Un langage $\mathcal{L}$ *est la donnée d'une signature* $\mathcal{S}$*, à laquelle on ajoute :*
- *des* symboles de connecteurs *:* $\neg$ *(non),* $\wedge$ *(et),* $\vee$*(ou),* $\Rightarrow$*(implique),* $\Leftrightarrow$*(bi-implique ou équivalant) ;*
- *des* symboles de quantifications *:* $\forall$ *(quantificateur universel),* $\exists$ *(quantificateur existentiel) ;*
- *des parenthèses : '(', ')' ;*
- *un ensemble infini dénombrable* $\mathcal{V}$ *de* variables.

A la lumière de cette définition et de celle de la signature, il apparaît clairement que le calcul de prédicats est basé sur l'algèbre de termes mais étend très largement son pouvoir d'expression.

Définition
Un langage égalitaire *est un langage auquel on rajoute le symbole =, appelé* symbole d'égalité*, qui est un symbole de relation binaire (*$=\in\mathcal{R}_2$*).*

Rem : Dans la suite du dossier, tous les langages étudiés seront des langages égalitaires et possèderont donc le symbole = implicitement.

Rem : Pour des questions de simplicité, nous commettrons l'abus d'identifier le langage à sa signature. Ainsi, lorsque nous parlerons du langage :
« $\mathcal{L}=\{c,f,g,R\}$ où c est un symbole de constante, f et g des fonctionnels unaires et R un relationnel binaire »
il faudra traduire par :
« $\mathcal{L}=\mathcal{V}\cup\{\},(,\neg,\wedge,\vee,\Rightarrow,\Leftrightarrow,\forall,\exists\}\cup\mathcal{C}\cup\bigcup_{n\in\mathbb{N}^*}\mathcal{R}_n\cup\bigcup_{n\in\mathbb{N}^*}\mathcal{F}_n$ où

$\mathcal{C}=\{c\}$, $\mathcal{R}_2=\{=,R\}$, $\mathcal{R}_n=\emptyset$ pour $n\neq2$ et $\mathcal{F}_1=\{f,g\}$, $\mathcal{F}_n=\emptyset$ pour $n\neq1$ ».
Devant la lourdeur d'une telle notation, on comprend aisément pourquoi on commet cet abus.

Exemple II.1
Un langage très naturel (de la théorie des corps ordonnés) est $\mathcal{L}=\{0,1,+,\times,-,^{-1},<\}$ où :
- 0 et 1 sont des symboles de constante
- $-$ (négation) et $^{-1}$ (inverse) sont des symboles de fonction unaire
- +, $-$ (soustraction) et $\times$ sont des symboles de fonction binaire
- $>$ est un symbole de relation binaire

On peut noter que le même symbole $-$ est utilisé deux fois. Ce sont en fait deux symboles différents et pour éviter toute ambiguïté, on aurait pu choisir deux notations séparées (c'est par exemple ce qui est fait dans les calculatrices), mais il est de coutume de les mélanger et de lever les ambiguïtés grâce au contexte.
On peut aussi remarquer (mais cela ne sera plus répéter ensuite) que le symbole = est implicite.

Exemple II.2
Un langage de la théorie de ensemble est $\mathcal{L}=\{\emptyset,\cup,\cap,^c,\in,\subset\}$ où :
- $\emptyset$ est un symbole de constante
- ^{c} (complémantaire) est un symbole de fonction unaire
- $\cup$ et $\cap$ sont des symboles de fonction binaire
- $\in$ et $\subset$ sont des symboles de relation binaire

Exemple II.3
Essayons maintenant de construire un langage un peu moins 'classique'.
Regardons ce que pourrait être un langage qui simule le fonctionnement statique d'une machine de Turing.
Une machine de Turing est constituée de :
- un contenu du ruban (l'alphabet)
- un ruban avec des cases
- une tête de lecture / écriture
- des états

Donc le langage doit être capable de modéliser tout cela.
Prenons alors le langage $\mathcal{L}=\{Pos, Cont_a,Cont_b,\ldots,Cont_\#, Etat_1,\ldots,Etat_n,\geq,\leq\}$ où :
- Pos est un symbole de fonction unaire
- $Etat_1,\ldots,Etat_n$ sont des symboles de prédicat 0-aires
- $Cont_a,Cont_b,\ldots,Cont_\#$ sont des symboles de prédicat unaires
- $\geq$ et $\leq$ sont des symboles de prédicat binaires

Sans rentrer dans les détails de l'interprétation de ces symboles et des mots qu'on pourra construire avec (sujet du chapitre II.2 page 16), signalons simplement que :
- Pos indique la position de la tête de lecture / écriture
- $Etat_1,\ldots,Etat_n$ indique dans quel état on se trouve

- $Cont_a, Cont_b, \ldots, Cont_\#$ permettent de reconstituer le contenu du ruban
- $\geq$ et $\leq$ ont leur interprétation usuelle

II.1.2. Formules du 1er ordre

Partant d'un langage du 1er ordre pris comme alphabet, nous allons construire, suivant la méthode récursive déjà utilisée pour les algèbres de termes, une famille de mots appelés formules du 1er ordre (nous ne préciserons plus par la suite « du 1er ordre »).

II.1.2.1. Formules

Définition
Soient $\mathcal{L}$ un langage et $\mathcal{M}(\mathcal{L})$ l'ensemble des mots du langage
Un mot $F \in \mathcal{M}(\mathcal{L})$ est une formule atomique *si $F=R(t_1,\ldots,t_n)$ avec :*
- *R un symbole de relation d'arité n ;*
- *$t_1,\ldots,t_n$ n termes.*

Notons $\mathcal{At}(\mathcal{L})$ l'ensemble des formules atomiques de $\mathcal{L}$.

On peut remarquer que, de même que pour les termes, il y a unicité de lecture des formules atomiques. C'est à dire que les parenthèses ne sont pas obligatoires si on adopte pour tous les symboles de relation une notation préfixe. On aurait donc pu écrire dans la définition $M=Rt_1\ldots t_n$ sans faire apparaître d'ambiguïté. Les parenthèses facilitent simplement la lecture et ne sont nécessaires que lorsqu'on utilise pour certains symboles (de relation ou de fonction) binaires une notation infixe (par ex a>b à la place de >ab).

Définition
Soient $\mathcal{L}$ un langage et $\mathcal{M}(\mathcal{L})$ l'ensemble des mots du langage
On construit $\mathcal{F}(\mathcal{L})$ l'ensemble des formules *du langage $\mathcal{L}$ ($\mathcal{F}(\mathcal{L})\subset\mathcal{M}(\mathcal{L})$) par récurrence :*
- *si* F *est une formule atomique, alors* F *est une formule ;*
- *si* F *est une formule, alors* $\neg$F *est une formule ;*
- *si* F *et* G *sont des formules, alors* F$\vee$G, F$\wedge$G, F$\Rightarrow$G *et* F$\Leftrightarrow$G *sont des formules ;*
- *si* F *est une formule et x une variable, alors* $\forall x$F *et* $\exists x$F *sont des formules.*

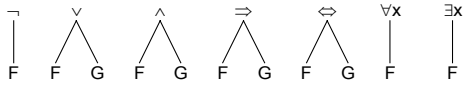
Exemple II.4
Si on reprend le langage de **Exemple II.1** (page 9), alors :
- $(x+1)\times(-0^{-1})$ est un terme (même s'il ne veut rien dire)
- $(x+y) = 1$ est une formule (atomique)

mais $(x+1)\times(-0^{-1}) \vee ((x+y) = 1)$ n'est rien du tout car on ne peut mettre un connecteur entre une formule et un terme.

Exemple II.5
Des formules pour le langage de **Exemple II.2** (page 9) sont par exemple :
$\forall x \forall y\ ((x \cup y)^c = (x^c \cap y^c))$
$\forall x \forall y\ ((x \cap y = \varnothing) \Rightarrow (x = \varnothing \wedge y = \varnothing))$

Pour les formules non atomiques, les parenthèses ne sont pas non plus indispensables si on se borne à une notation préfixe pour tous les symboles (l'unicité de lecture des formules est une propriété démontrable du calcul des prédicats). En toute rigueur, nous n'aurions donc pas dû les mettre dans la définition des langages, mais puisqu'elles sont toujours utilisées en pratique certains ouvrages les y inclus tout de même ; nous avons donc adopté la même approche dans ce dossier.

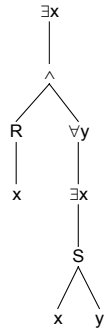
Nous avions vu au paragraphe I.1.2 (page 3) que les termes peuvent être représentés par des arbres. Il est possible d'étendre cette représentation aux formules. Voici comment construire les sous-arbres à partir des nouveaux symboles introduits :



Exemple II.6

Supposons un langage $\mathcal{L}$ approprié ; x et y sont des variables.

La formule $\exists x\ (R(x) \wedge \forall y\ \exists x\ S(x,y))$ est équivalente à l'arbre



II.1.2.2. Sous-formules

De part la construction par récurrence de l'ensemble des formules ($\mathcal{F}(\mathcal{L})$), les formules sont des mots très structurés (ce qui explique pourquoi il est possible de représenter ces mots par des arbres). Elles font en faite décomposables en sous-mots qui ont la particularité d'être aussi des formules. Ce sous-mots sont appelés sous-formules.

Définition

Soient $\mathcal{L}$ *un langage ;*
F,G *et* H *des formules de ce langage ;*
x une variable.

sf(F) *est l'ensemble des* sous-formules *de* F. *Il est défini par récurrence comme suit :*

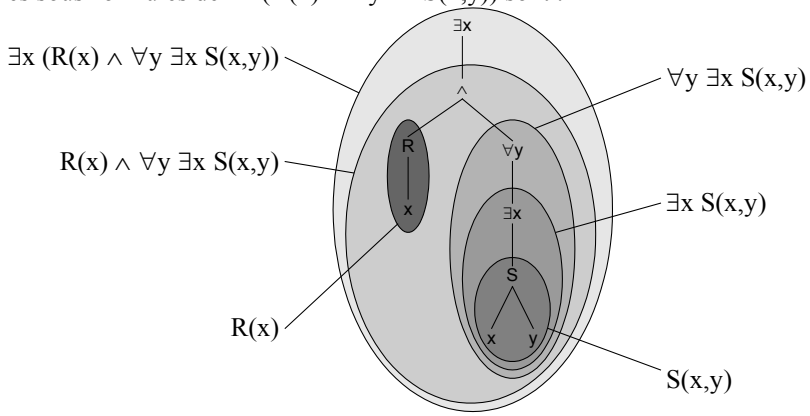
- *si* F *est atomique, alors* $sf(F) = \{F\}$;
- *si* $F=\neg G$, *alors* $sf(F) = \{F\} \cup sf(G)$;
- *si* $F=(G \alpha H)$ *avec* $\alpha \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$, *alors* $sf(F) = \{F\} \cup sf(G) \cup sf(H)$;
- *si* $F=\forall x G$ *ou* $F=\exists x G$, *alors* $sf(F) = \{F\} \cup sf(G)$.

Les sous-formules d'une formule sont généralement plus faciles à voir à partir de la représentation arborescente de la formule. En effet, les sous-formules correspondent aux sous-arbres dont la racine est étiquetée par un symbole de relation, un connecteur ou un quantificateur.

Exemple II.7

Reprenons **Exemple II.6** ci-dessus.

Les sous-formules de $\exists x\ (R(x) \wedge \forall y\ \exists x\ S(x,y))$ sont :



II.1.3. Variables libres / liées

L'existence des quantificateurs engendre un certain nombre de difficultés. Par exemple dans la formule « $\exists x\ (R(x) \wedge \forall y\ \exists x\ S(x,y))$ » (**Exemple II.6** page 11) le « x » de « S(x,y) » fait-il référence au 1er ou au 2ème « ∃ » ou peut-être est-il complètement indépendant ? C'est pour répondre à ce genre de problème qu'il faut distinguer le statut (libre ou lié) de chaque occurrence des variables.

II.1.3.1. Statut des occurrences de variables

Définition

Soient $\mathcal{L}$ *un langage ;*
F *et G des formules de ce langage ;*
x une variable.

On définit le statut des occurrences de variables comme suit :

- *toutes les occurrences de variables dans les formules atomiques sont* libres *;*
- *les occurrences* libres *(resp.* liées*) des variables de* $\neg$F *sont celles de* F
- *les occurrences* libres *(resp.* liées*) des variables de* FαG *où* $\alpha \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$ *sont celles de* F *et de* G *;*
- *les occurrences* libres *(resp.* liées*) des variables y distincte de x dans* $\exists x$F *et* $\forall x$F *sont celles de* F *;*
- *les occurrences de x dans* $\exists x$F *et* $\forall x$F *sont toutes* liées*.*

Exemple II.8

Dans les formules suivantes les occurrences libres des variables sont écrites en bleu tandis que les occurrences liées sont en rouge.
F : $\forall$x (x.y = y.x)
G : ($\forall$x$\exists$y (x.z = z.y)) $\wedge$ (x = z.z)
Il est possible de faire plusieurs remarques sur ces deux formules :

- dans F, on voit qu'une occurrence de variable peut être liée dans une formule et libre dans une sous-formule (ici x.y = y.x) ;
- dans G, on voit qu'une variable (ici x) peut avoir des occurrences libres et d'autres liées.

On emploie parfois l'expression « occurrence muette » à la place de « occurrence liée ». Cette appellation a le mérite de souligner un aspect important de ce type d'occurrence : les occurrences muettes peuvent toujours être renommées (en prenant tout de même quelques précautions) sans que cela ne modifie le sens de la formule (nous définirons au chapitre II.2 Sémantique page 16 ce qu'est le sens d'une formule).

Exemple II.9

Il est possible de clarifier la formule G de **Exemple II.8** ci-dessus de façon à ce que la variable x n'ait pas à la fois des occurrences libres et d'autres liées en renommant ses occurrences muettes (liées) :
G : ($\forall\clubsuit\exists$y (♣.z = z.y)) $\wedge$ (x = z.z)

Exemple II.10

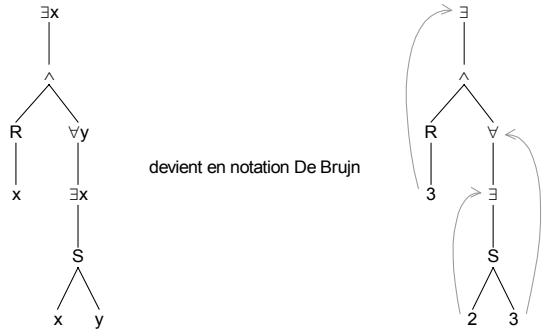
En anticipant sur le chapitre II.2 Sémantique, voici un exemple pour illustrer qu'il faut prendre quelques précautions pour renommer les occurrences muettes d'une formule.
On devine que si, dans F : $\forall$x$\exists$y (x.z = z.y), on renomme y par x (F' : $\forall$x$\exists$x (x.z = z.x)) le sens de F n'est pas conservé.

Il existe un algorithme qui permet de savoir si une occurrence de, par exemple, x est libre ou liée : à partir de l'occurrence, on remonte vers la racine et si sur le chemin on rencontre $\forall$x ou $\exists$x alors l'occurrence est liée sinon elle est libre.

Les deux idées précédentes (le caractère muet des occurrences liées et leur relation avec les quantificateurs $\forall$ et $\exists$) ont donné naissance à une notation différente de celle que nous avons utilisée jusqu'ici : la notation de De Brujn. Dans cette dernière, les occurrences liées perdent leur nom et sont remplacées par des pointeurs vers le quantificateur auquel elles sont liées.

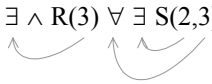
Exemple II.11

La formule F : $\exists$x (R(x) $\wedge$ $\forall$y $\exists$x S(s,y)) se traduit en notation De Brujn par $\exists$ (R(3) $\wedge$ $\forall$ $\exists$ S(2,3)).
Cette notation n'est pas évidente à comprendre sous forme de mot mais elle paraît beaucoup plus claire en représentation sous forme d'arbre.



On peut remarquer, à propos de la clarté de représentation des mots, qu'en notation De Brujn, il est plus commode d'utiliser les fonctions / relations binaires sous forme préfixe plutôt qu'infixe car cette dernière désorganise l'ordre d'apparition des différents symboles. Ainsi, dans le mot $\exists \wedge$ R(3) $\forall \exists$ S(2,3) il est relativement simple de retrouver à quel quantificateur

chaque pointeur fait référence : il suffit de compter les symboles de droite à gauche à partir du pointeur.



II.1.3.2. Formules closes

Intuitivement une formule close est une formule « qui ne varie pas ». Cela ne signifie pas comme dans le cas des termes que la formule ne puisse pas être constituée de variables, mais dans ce cas, ces dernières ne doivent pas être modifiables depuis l'extérieur de la formule. Formalisons cette intuition.

Définition
Soient $\mathcal{L}$ *un langage ;*
F *une formule de ce langage ;*
x une variable apparaissant dans F.
Alors x est une variable libre *de* F *si au moins une occurrence de x dans* F *est libre.*

En d'autres termes, les variables libres sont des variables qui sont « modifiables depuis l'extérieur de la formule ».

Exemple II.12
Si on reprend **Exemple II.8** (page 12) où nous avions recherché les occurrences libres et liées dans les formules F $(\forall x\ (x.y = y.x))$ et G $((\forall x \exists y\ (x.z = z.y)) \wedge (x = z.z))$, alors les variables libres sont :
- pour F : y
- pour G : x (bien que certaines occurrences soient liées) et z

Les variables libres sont très faciles à voir en notation De Brujn puisque ce sont les seules variables qui apparaissent explicitement. En effet, toutes les occurrences liées sont remplacées par des pointeurs donc les variables non libres disparaissent complètement.

Exemple II.13
Traduisons la formule G ci-dessus en notation De Brujn : $(\forall \exists (4.z = z.3)) \wedge (x = z.z)$
Les variables *x* et *z* apparaissent explicitement tandis que *y* a disparue. Cela correspond bien aux variables libres que nous avions trouvés dans **Exemple II.12**.

Définition
Une formule close *est une formule qui ne contient aucune variable libre.*

Exemple II.14
La formule F : $\exists x\ (R(x) \wedge \forall y\ \exists x\ S(s,y))$ est close. Elle ne contient aucune variable libre comme le montre son écriture en notation De Brujn : $\exists\ (R(3) \wedge \forall\ \exists\ S(2,3))$ (**Exemple II.11** page 12).

Dans le cas général, une formule close ne contient pas forcément de quantificateur (par exemple, $1 < (1+1+1)$ est une formule atomique close). Cependant dans un langage sans symbole de constante, il n'y a pas de formule close sans quantificateur (notamment aucune formule atomique n'est close).

II.1.4. Substitutions

Pour de multiples raisons (souvent d'ordre sémantique), il est intéressant de modifier une formule en remplaçant une ou plusieurs variables par un ou plusieurs terme(s). La substitution est une opération syntaxique qui nous permet de réaliser cela.

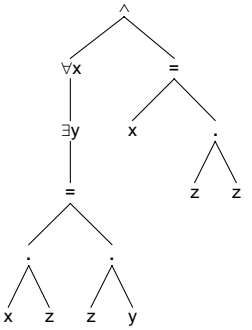
Cette opération en logique du 1er ordre est bien plus compliquée que dans les algèbres de termes. Deux raisons à cela :

- Il ne faut pas remplacer toutes les occurrences de la variable qu'on veut substituer : les occurrences libres ne sont pas concernées par cette opération puisqu'elles sont muettes.
- La substitution ne doit pas engendrer de nouvelles occurrences liées.

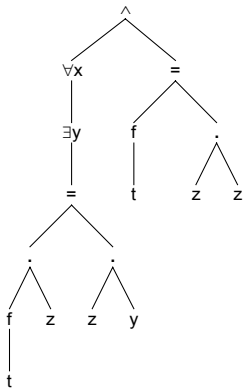
Exemple II.15

Pour illustrer le premier problème, essayons de substituer f(t) à x dans la formule F : (∀x∃y (x.z = z.y)) ∧ (x = z.z).

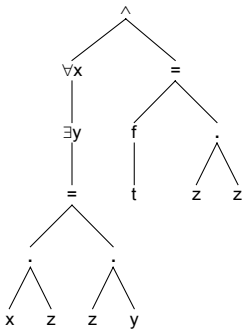
Si on substitue $\frac{f}{t}$ à toutes les occurrences de x dans



on obtient



au lieu de

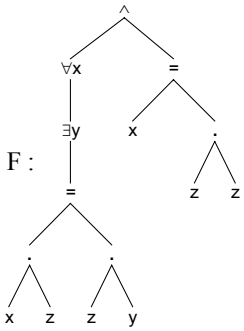


(substitution correcte)

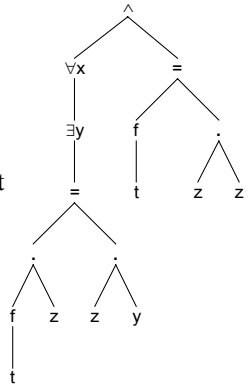
Cette substitution brutale remplace trop d'occurrences de x.

On peut aussi comprendre cela d'un point de vue sémantique. F' : (∀ 💀 ∃y (💀 .z = z.y)) ∧ (x = z.z) est une formule sémantiquement équivalente à F (on exploite le caractère muet des occurrences liées). Donc intuitivement on voudrait que la même substitution sur F ou sur F' donne deux formules de sens équivalent. Or ce n'est pas le cas avec cette substitution brutale :

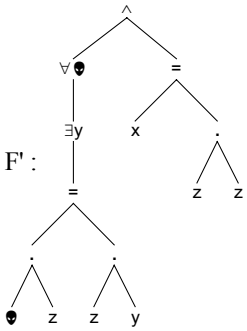
F :



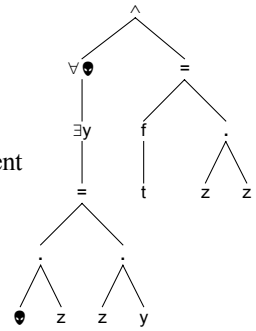
devient



et F' :



devient

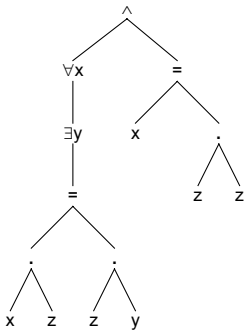


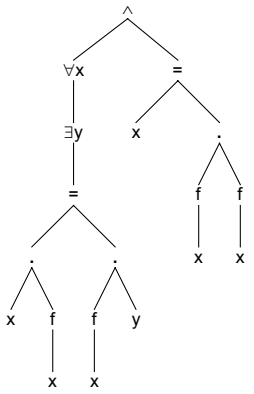
Clairement, même si nous n'avons pas encore définit la sémantique du calcul des prédicats, les deux résultats ne peuvent avoir le même sens.

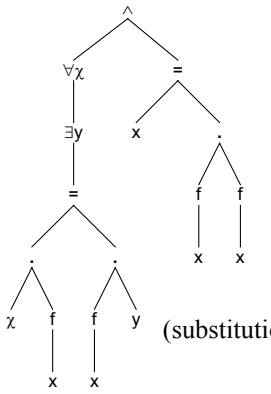
Exemple II.16

Pour illustrer le second problème, essayons de substituer t : f(x) à z dans la formule F : (∀x∃y (x.z = z.y)) ∧ (x = z.z).

Si on substitue $\frac{f}{x}$ à toutes les occurrences libres de z dans



on obtient 

au lieu de 

(substitution correcte)

Cette fois-ci le problème vient du fait que le terme qu'on veut substituer à z contient une variable (x) dont une occurrence est liée dans F. Les quantificateurs de la formule ne doivent pas porter sur les variables de t. Pour éviter cela, il suffit de renommer dans F les occurrences muettes des variables qui apparaissent à la fois dans la formule et dans le terme (ici, $x \rightarrow \chi$).

La définition de la substitution doit donc tenir compte de ces deux problèmes et la façon la plus facile pour faire cela est de la construire par récurrence.

Définition

Soient $\mathcal{L}$ *un langage ;*
F *une formule de ce langage ;*
$x_1,\ldots,x_k$ *k variables libres de* F *deux à deux distinctes*
$t_1,\ldots,t_k$ *k termes*
$y_1,\ldots,y_l$ *les variables contenues dans les k termes*
x *une variable à priori quelconque*
z *une variable n'appartenant ni à* F*, ni à aucun des termes* $t_1,\ldots, t_k$

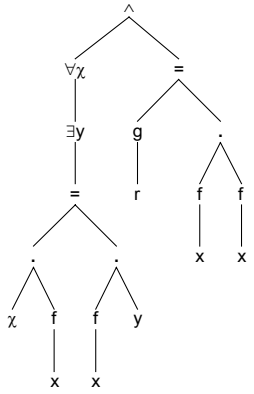
On note $F[t_1/x_1,\ldots,t_k/x_k]$ *le résultat de la* substitution *des termes* $t_1,\ldots, t_k$ *aux variables* $x_1,\ldots,x_k$ *, respectivement, dans toutes les occurrences libres de celles-ci dans* F.

Construction de $F[t_1/x_1,\ldots,t_k/x_k]$ *:*

- *si* F *est une formule atomique* $R(u_1,\ldots,u_n)$ *avec n≥1, alors* $F[t_1/x_1,\ldots,t_k/x_k] = R(u_1[t_1/x_1,\ldots,t_k/x_k],\ldots,u_n[t_1/x_1,\ldots,t_k/x_k])$ *;*
- *si* $F=\neg G$*, alors* $F[t_1/x_1,\ldots,t_k/x_k]= \neg G[t_1/x_1,\ldots,t_k/x_k]$ *;*
- *si* $F=G\alpha H$ *où* $\alpha \in \{\wedge,\vee,\Rightarrow,\Leftrightarrow\}$*, alors* $F[t_1/x_1,\ldots,t_k/x_k]= G[t_1/x_1,\ldots,t_k/x_k]\ \alpha\ H[t_1/x_1,\ldots,t_k/x_k]$ *;*
- *si* $F=\forall xG$ *,*
  - *si* $x \notin \{x_1,\ldots,x_k\}$*,*
    - *si* $x \notin \{y_1,\ldots,y_l\}$*, alors* $F[t_1/x_1,\ldots,t_k/x_k]= \forall xG[t_1/x_1,\ldots,t_k/x_k]$ *;*
    - *si* $x \in \{y_1,\ldots,y_l\}$*, alors* $F[t_1/x_1,\ldots,t_k/x_k]= \forall zG'[t_1/x_1,\ldots,t_k/x_k]$*, où* G' *est la formule* G *dans laquelle toutes les occurrence de* x *ont été renommées par* z *(caractère muet des occurrence liées) ;*
  - *si* $x \in \{x_1,\ldots,x_k\}$*, mettons* $x=x_i$*, alors*
    - *si* $x \notin \{y_1,\ldots,y_l\}$*, alors* $F[t_1/x_1,\ldots,t_k/x_k]= \forall xG[t_1/x_1,\ldots ,t_{i-1}/x_{i-1},t_{i+1}/x_{i+1},\ldots,t_k/x_k]$ *;*
    - *si* $x \in \{y_1,\ldots,y_l\}$*, alors* $F[t_1/x_1,\ldots,t_k/x_k]= \forall zG'[t_1/x_1,\ldots ,t_{i-1}/x_{i-1},t_{i+1}/x_{i+1},\ldots,t_k/x_k]$ *avec* G' *défini plus haut ;*
- *si* $F=\exists xG$ *, même raisonnement que pour* $F=\forall xG$.

Exemple II.17

Illustrons le cas où $F=\forall xG$ et $x \in \{x_1,\ldots,x_k\}$ et $x \in \{y_1,\ldots,y_l\}$.
Substituons f(r) à x et g(x) à z dans F : $(\forall x \exists y\ (x.z = z.y)) \wedge (x = z.z)$.

F[g(r)/x,f(x)/z]= 

II.2. Sémantique

Maintenant que nous savons écrire les formules et réaliser une opération essentielle dessus (la substitution), nous allons nous intéresser au sens que ces formules peuvent avoir et donc à quoi elles servent.

II.2.1. Structures

Définition

Une structure $\mathcal{M}$ *(ou* réalisation *ou* interprétation*) pour un langage* $\mathcal{L}$ *est la donnée de :*

- *un ensemble non vide* M*, appelé* domaine *(ou* ensemble de base *ou* ensemble sous-jacent*) ;*
- *un élément* c_M *pour chaque symbole de constante* c *de* $\mathcal{L}$ *;*
- *une fonction* $f_M : M^k \to M$ *pour chaque symbole de fonction* f *de* $\mathcal{L}$ *d'arité* $k \geq 1$ *;*
- *une relation* $R_M \subseteq M^k$ *pour chaque symbole de relation* R *de* $\mathcal{L}$ *d'arité* $k \geq 1$ *;*
- *une des valeurs « Vrai » ou « Faux » pour chaque symbole de relation de* $\mathcal{L}$ *d'arité* $k=0$.

Pour comprendre quelle est la relation entre une structure et des formules, il est possible d'adopter deux points de vue :

- Une formule ne prend de sens que dans une structure (par exemple, est-ce que $\exists x\ldots$ signifie « il existe un entier x tel que … » ou « il existe un réel x tel que … » ou « il existe un graphe x tel que … » etc. ?). Donc une structure est là pour donner une signification à une formule.
- Une structure existe en tant que telle (ce peut être par exemple l'ensemble des entiers relatifs muni de l'opération d'addition) et les formules (dans un langage adéquat servent à exprimer des propriétés de cette structure (par exemple le fait que 0 soit l'élément neutre de l'addition se dira $\forall x(x+0=x \wedge 0+x=x)$).

Ces deux points de vue montrent la complémentarité des structures et des formules. Bien que les uns puissent exister sans les autres syntaxiquement, ce n'est pas le cas sémantiquement : les formules doivent être liées à une structure pour avoir un sens et les structures doivent être caractérisées par des formules pour avoir des propriétés intéressantes.

Exemple II.18

A partir de la structure du corps ordonné des nombres réel $\langle \Re, \leq, +, \times, 0, 1 \rangle$, cherchons un langage adapté pour certaines de ses propriétés.

Ce langage doit disposer :

- d'une relation binaire R (destinée à représenter l'ordre ≤),
- de deux symboles de fonction binaire f et g (pour les opérations + et ×)
- et des symboles de constante a et b (pour 0 et 1).

Nous pouvons désormais exprimer que 1 est élément neutre pour × en disant que la formule $\forall x(g(x,b)=x \wedge g(b,x)=x)$ est toujours satisfaite (nous formaliserons cette notion au chapitre II.2.3 page 18).

De même 0, est élément neutre de + se dit $\forall x(f(x,a)=x \wedge f(a,x)=x)$ est toujours satisfaite.

Ou encore, la formule $\forall x \forall y(R(x,y) \Rightarrow R(y,x))$ n'est pas satisfaite traduit le fait que ≤ n'est pas symétrique.

Remarquons que s'il est plus clair, car sans ambiguïté possible, de donner des noms aux symboles du langage différents de ceux de la structure (R pour ≤ par exemple), en pratique il est plus courant de les mélanger (≤ représentera à la fois l'ordre du corps des réels et le symbole de relation binaire associé dans notre langage).

Exemple II.19

Essayons maintenant de réaliser l'exercice inverse en cherchant une famille de structures d'interprétation à un langage. Pour cela reprenons le langage définit dans **Exemple II.3** page 9 servant à simuler le fonctionnement statique d'une machine de Turing.

On procède en deux étapes :

- on fixe un domaine d'interprétation ;
- on fait correspondre à chaque élément de la signature du langage un élément dans la famille de structures.

Ce qui donne en résumé :

Signature du langage		Famille de structures
		Domaine : $\aleph$ (pour représenter les cases du ruban)
Pos	(fonction unaire)	Pos^δ est le numéro de la case où se trouve la tête de lecture
$Etat_1,\ldots,Etat_n$	(prédicat 0-aires)	$1 \leq i \leq n,\ Etat_i^\delta = \begin{cases} \text{vrai si l'état courant est } q_i \\ \text{faux sinon} \end{cases}$
$Cont_a, Cont_b, \ldots, Cont_\#$	(prédicat unaires)	$Cont_\alpha^\delta = \{i \in \aleph \mid \text{la case i contient la lettre } \alpha\},\ \alpha \in \{a,b,\ldots,\#\}$
≥ et ≤	(prédicat binaires)	≥ et ≤ avec leur interprétation usuelle

Toutes les descriptions instantanées de la machine de Turing peuvent être traduites par un membre de la famille de structures définie ci-dessus. Cependant, la réciproque est fausse : toutes les structures de cette famille ne correspondent pas forcément à une description instantanée. Pour cela, il faut restreindre cette famille en imposant à ses structures certaines propriétés particulières comme nous le verrons dans **Exemple II.20** page 18.

II.2.2. Valeurs des formules

Comme nous l'avons déjà fait pour les termes, nous allons donner une valeur à chaque formule. Cette valeur correspond en fait au sens de la formule F dans une structure $\mathcal{M}$. On note val(F, $\mathcal{M}$) l'ensemble des valeurs que peut prendre F dans $\mathcal{M}$.

Avant de définir val(F, $\mathcal{M}$) remarquons que dans chaque structure $\mathcal{M}$ (au sens du calcul des prédicats) se cache une interprétation $\mathcal{I}$ (au sens des algèbres des termes). Ainsi, pour tout terme t, val(t, $\mathcal{M}$) = val(t, $\mathcal{I}$) (voir le chapitre I.2.2 page 6 pour la définition de val(t, $\mathcal{I}$)).

Soit une structure $\mathcal{M}$, un langage $\mathcal{L}$ et une formule F$\in\mathcal{L}$, alors val(F, $\mathcal{M}$) se définit par récurrence :

- Si F est une formule atomique $R(t_1,...,t_k)$ où $t_1,...,t_k$ sont des termes.

  Les variables de F sont celles de $t_1,...,t_k$ et toutes les occurrences sont libres

  Soient $\begin{vmatrix} x_1,...,x_n & \text{les variables de } F \\ x_1^1,...,x_{n_1}^1 & \text{les variables de } t_1 \\ \dots & \dots \\ x_1^k,...,x_{n_k}^k & \text{les variables de } t_k \end{vmatrix}$ avec $(\sum_{i=1}^{k} n_i = n)$

  Alors, $val(F, \mathcal{M}) = \{(a_1,...,a_n)\in M^n \mid (val(t_1, \mathcal{M})(a_1^1,...,a_{n_1}^1), \dots, val(t_k, \mathcal{M})(a_1^k,...,a_{n_k}^k))\in R_M$

  Rem : Si n=0, les termes $t_1,...,t_k$ sont clos, alors $val(F, \mathcal{M}) = \begin{cases} Vrai & \text{si } (val(t_1,\mathcal{M}),...,val(t_k,\mathcal{M}))\in R_M \\ Faux & \text{si } (val(t_1,\mathcal{M}),...,val(t_k,\mathcal{M}))\notin R_M \end{cases}$

- Si $F=\neg G$, alors F et G ont les mêmes occurrences de variables libres et

  $val(F, \mathcal{M}) = \begin{cases} M^n / val(G,\mathcal{M}) & \text{si } G \text{ a n}\geq 1 \text{ variables libres} \\ \begin{cases} Vrai & \text{si } val(G,\mathcal{M})=Faux \\ Faux & \text{si } val(G,\mathcal{M})=Vrai \end{cases} & \text{si } G \text{ a 0 variable libre} \end{cases}$

- Si $F=G\wedge H$, alors les occurrences libres des variables de F proviennent de G ou (le 'ou' logique) de F.

  On note $\begin{vmatrix} x_1,...,x_m & \text{les variables libres de } G \\ y_1,...,y_n & \text{les variables libres de } H \\ y_1,...,y_p & \text{les variables libres communes à } G \text{ et } H \end{vmatrix}$

  Donc $F = G(x_1,...,x_m,z_1,...,z_p) \wedge H(y_1,...,y_n,z_1,...,z_p)$, et

  $val(F, \mathcal{M}) = \{(a_1,...,a_m,b_1,...,b_n,c_1,...,c_p)\in M^{m+n+p} \mid (a_1,...,a_m,c_1,...,c_p)\in val(G, \mathcal{M}) \text{ et } (b_1,...,b_n,c_1,...,c_p)\in val(H, \mathcal{M})\}$

  Ce que l'on notera plus simplement :

  $val(F, \mathcal{M}) = \{(\vec{a},\vec{b},\vec{c})\in M^{m+n+p} \mid (\vec{a},\vec{c})\in val(G, \mathcal{M}) \text{ et } (\vec{b},\vec{c})\in val(H, \mathcal{M})\}$

  Rem : En reprenant la notion ensembliste de la jointure, $val(F, \mathcal{M}) = val(G, \mathcal{M}) \bowtie val(H, \mathcal{M})$

- Si $F=G\vee H$, alors, comme dans le cas ci-dessus, les occurrences libres des variables de F proviennent de G ou de F.

  $val(F, \mathcal{M}) = \{(\vec{a},\vec{b},\vec{c})\in M^{m+n+p} \mid (\vec{a},\vec{c})\in val(G, \mathcal{M}) \text{ ou } (\vec{b},\vec{c})\in val(H, \mathcal{M})\}$

- Si $F=G\Rightarrow H$, alors

  $val(F, \mathcal{M}) = \{(\vec{a},\vec{b},\vec{c})\in M^{m+n+p} \mid (\vec{a},\vec{c})\notin val(G, \mathcal{M}) \text{ ou } (\vec{b},\vec{c})\in val(H, \mathcal{M})\}$

- Si $F=G\Leftrightarrow H$, alors

  $val(F, \mathcal{M}) = \{(\vec{a},\vec{b},\vec{c})\in M^{m+n+p} \mid [(\vec{a},\vec{c})\in val(G,\mathcal{M}) \text{ et } (\vec{b},\vec{c})\in val(H,\mathcal{M})] \text{ ou } [(\vec{a},\vec{c})\notin val(G,\mathcal{M}) \text{ et } (\vec{b},\vec{c})\notin val(H,\mathcal{M})]\}$

- Si $F=\exists x\, G$, alors il faut distinguer deux cas :

- Si x ne fait pas parti des variables libres de G

  $val(F, \mathcal{M}) = val(G, \mathcal{M})$

- Si x est une des variables libres de G (on notera ces variables : $x, x_1, \ldots, x_k$)

$$val(F, \mathcal{M}) = \begin{cases} \{(a_1,\ldots,a_k) \in M^k \mid \text{il existe a} \in \text{M tq } (a,a_1,\ldots,a_k) \in val(G,\mathcal{M})\} & \text{si } k>1 \\ Vrai \quad \text{si } val(G,\mathcal{M}) \neq \varnothing & \\ Faux \quad \text{sinon} & \text{si } k=0 \end{cases}$$

Rem : $\{(a_1,\ldots,a_k) \in M^k \mid \text{il existe a} \in \text{M tq } (a,a_1,\ldots,a_k) \in val(G,\mathcal{M})\}$ se note aussi $proj_{x_1,\ldots,x_k}(val(G,\mathcal{M}))$.

- Si $F = \forall x\ G$, alors il faut aussi distinguer deux cas :

  - Si x ne fait pas parti des variables libres de G

    $val(F, \mathcal{M}) = val(G, \mathcal{M})$

  - Si x est une des variables libres de G (on notera ces variables : $x, x_1, \ldots, x_k$)

$$val(F, \mathcal{M}) = \begin{cases} \{(a_1,\ldots,a_k) \in M^k \mid \text{pour tout a} \in \text{M tq } (a,a_1,\ldots,a_k) \in val(G,\mathcal{M})\} & \text{si } k>1 \\ Vrai \quad \text{si } val(G,\mathcal{M}) = \text{M} & \\ Faux \quad \text{sinon} & \text{si } k=0 \end{cases}$$

Exemple II.20

Dans **Exemple II.19** page 16, nous avions construit une famille $\mathcal{F}$ de structures permettant d'interpréter le langage définit dans **Exemple II.3** page 9 servant à simuler le fonctionnement statique d'une machine de Turing. Nous avions vu que toutes les descriptions instantanées de la machine de Turing pouvaient être traduites par une structure de $\mathcal{F}$. En revanche, toutes les structures de $\mathcal{F}$ ne correspondaient pas forcément à une description valide.

Dans cet exemple, nous allons chercher à caractériser quelles sont les structures de $\mathcal{F}$ qui correspondent à une description valide de la machine de Turing. Pour cela, nous allons définir des formules et nous dirons que, parmi les structures de $\mathcal{F}$, seules celles pour lesquelles ces formules sont vraies (val(F, $\mathcal{M}$)=Vrai) correspondent à une description valide.

Regardons pourquoi une structure ne correspond pas toujours à une description valide. Soit une structure δ, alors :

$$\delta = (\aleph, \underbrace{Pos^\delta}_{1}, \underbrace{Etat_1^\delta,\ldots,Etat_n^\delta}_{2}, \underbrace{Cont_a^\delta, Cont_b^\delta,\ldots,Cont_\#^\delta}_{3})$$

1: Il ne peut y avoir de # à gauche de la tête
2: Seul un état peut être vrai
3: $Cont_a^\delta, Cont_b^\delta,\ldots,Cont_\#^\delta$ doivent constituer une partition de ℵ

Si une structure de $\mathcal{F}$ ne respecte pas l'une des ces trois conditions, alors elle ne correspond pas à une description valide.

Traduisons alors ces trois conditions en formules (F_1, F_2, F_3) :

$$F_1 = \forall x\ (Cont_\#(x) \Rightarrow x \geq Pos)$$

$$F_2 = (Etat_0 \vee \ldots \vee Etat_n) \wedge \neg(Etat_0 \wedge Etat_1) \wedge \ldots \wedge \neg(Etat_0 \wedge Etat_n) \wedge \neg(Etat_1 \wedge Etat_2) \wedge \ldots \wedge \neg(Etat_1 \wedge Etat_n) \wedge \ldots \wedge \neg(Etat_{n-1} \wedge Etat_n)$$

$$F_3 = \forall x\ (Cont_a(x) \vee \ldots \vee Cont_\#(x)) \wedge \forall x\ [\ (Cont_a(x) \Rightarrow \neg Cont_b(x) \wedge \neg Cont_c(x) \wedge \ldots \wedge \neg Cont_z(x) \wedge \neg Cont_\#(x)) \wedge (Cont_b(x) \Rightarrow \neg Cont_a(x) \wedge \neg Cont_c(x) \wedge \ldots \wedge \neg Cont_z(x) \wedge \neg Cont_\#(x)) \wedge \ldots \wedge (Cont_\#(x) \Rightarrow \neg Cont_a(x) \wedge \neg Cont_b(x) \wedge \ldots \wedge \neg Cont_z(x))\ ]$$

Donc toutes les structures $\mathcal{M}$ de $\mathcal{F}$ telles que val(F_1,$\mathcal{M}$)=Vrai et val(F_2,$\mathcal{M}$)=Vrai et val(F_3,$\mathcal{M}$)=Vrai correspondent à une description valide de la machine de Turing.

Rem : On dit que « $\mathcal{M}$ satisfait F de $a_1,\ldots,a_n$ » ou « F est satisfaite dans $\mathcal{M}$ par $(a_1,\ldots,a_n)$ » et on note $\mathcal{M} \models F(a_1,\ldots,a_n)$ si $(a_1,\ldots,a_n) \in val(F,\mathcal{M})$.

Dans le cas d'une formule closes, $\mathcal{M} \models F$ si val(F,$\mathcal{M}$)=Vrai.

Donc dans l'exemple précédant, on aurait pu dire : « toutes les structures $\mathcal{M}$ de $\mathcal{F}$ qui satisfont F_1, F_2 et F_3 correspondent à une description valide de la machine de Turing ».

II.2.3. Formules valides

Une notion importante pour les démonstrations que nous pourrons faire grâce à la logique du 1er ordre est celle des formules valides.

Définition :

Soient $\mathcal{L}$ *un langage ;*

$\mathcal{S}$ *sa signature ;*

F *une formule de ce langage ;*
$x_1,\ldots,x_k$ *les variables libres de* F.
F *est* valide *si pour toute interprétation (ou structure)* $\mathcal{M}$ *de* $\mathcal{S}$*, on a :*
- *si* $k \neq 0$, $val(F, \mathcal{M})=M^k$
- *si* $k = 0$, $val(F, \mathcal{M})=Vrai$.

Exemple II.21

Soit H : $(\forall x\ (F(x) \Rightarrow G(x))) \Rightarrow (\forall x\ F(x) \Rightarrow \forall x\ G(x))$
Montrons que H est valide (i.e. vraie pour toute interprétation $\mathcal{M}$).
Décomposons H : $\underbrace{(\forall x\ (F(x) \Rightarrow G(x)))}_{H_1} \Rightarrow \underbrace{(\underbrace{\forall x\ F(x)}_{H_{2a}} \Rightarrow \underbrace{\forall x\ G(x)}_{H_{2b}})}_{H_2}$

Considérons dans un 1er temps que x soit la seule variable libre dans F et G.
Donc H, H_1, H_2, H_{2a} et H_{2b} sont des formules closes.
H est une implication donc :

- si H_1 est fausse dans $\mathcal{M}$
  alors H est immédiatement vrai ;
- si H_1 est vraie dans $\mathcal{M}$
  alors il faut montrer que H_2 l'est aussi :
  - si H_{2a} est fausse
    alors H_2 est immédiatement vraie
    donc H est vraie ;
  - si H_{2a} est vraie
    alors $\mathcal{M} \models (\forall x\ F(x))$ ou encore $val(F(x), \mathcal{M})=M$
    or H_1 est vraie dans $\mathcal{M} \Leftrightarrow \mathcal{M} \models (\forall x\ F(x) \Rightarrow G(x))$
    $\Leftrightarrow val(F(x) \Rightarrow G(x), \mathcal{M}) = M$
    $\Leftrightarrow val(\neg F(x) \vee G(x), \mathcal{M}) = M$
    $\Leftrightarrow val(\neg F(x), \mathcal{M}) \cup val(G(x), \mathcal{M}) = M$
    $\Leftrightarrow val(G(x), \mathcal{M}) \supseteq M \setminus (M \setminus val(F(x), \mathcal{M}))$
    $\Leftrightarrow val(G(x), \mathcal{M}) \supseteq val(F(x), \mathcal{M})$
    donc $val(G(x), \mathcal{M})=M$ ou encore $\mathcal{M} \models (\forall x\ G(x))$, c'est à dire H_{2b} est vraie
    donc H_2 est vraie
    donc H aussi.

Pour montrer que H est valide, il faut encore montrer que H est vraie si F et G ont d'autres variables libres.
Pour cela, il faut considérer des éléments de $\mathcal{M}$ pour geler ces autres variables libres, et effectuer la même preuve que ci-dessus.

Exemple II.22

Soit H : $(\forall x\ F(x) \Rightarrow \forall x\ G(x)) \Rightarrow (\forall x\ (F(x) \Rightarrow G(x)))$
Montrons que H n'est pas valide.
Pour cela, donnons un contre-exemple.
Prenons la structure $\mathcal{M} = (\aleph, +, \times)$ avec F(x) : x est pair et G(x) : x est impair.
Alors, dans $\mathcal{M}$, $(\forall x\ F(x) \Rightarrow \forall x\ G(x))$ est vraie car $\forall x\ F(x)$ est fausse.
Or $\forall x\ (F(x) \Rightarrow G(x))$ est fausse dans $\mathcal{M}$.
Donc H est fausse dans $\mathcal{M}$.

A première vue, il semble difficile dans le cas général de pouvoir vérifier une telle propriété puisqu'il y a à priori une infinité de structures possibles pour un langage donné. En fait, il est possible de définir de manière uniforme pour chaque langage une liste d'axiomes et de règles de déduction telle qu'une formule est valide si et seulement si elle est prouvable à partir des axiomes à l'aide des règles de déduction. On dit alors qu'une telle formule est un théorème.

Il y a équivalence entre dire :
- qu'une formule est valide,
- qu'il s'agit d'un théorème et
- qu'il existe une preuve pour la démontrer.

Pour savoir si une formule est un théorème, il suffit de tester tous les arbres de preuve et de regarder si cette formule figure dans un tel arbre. C'est un algorithme sans assurance de terminaison.

La notation Z

Maintenant que nous avons décrit assez précisément la logique du 1er ordre, il est intéressant de voir comment celle-ci peut nous aider à spécifier un programme. Etudions pour cela la notation Z. Celui-ci n'est pas un langage au sens où il n'existe pas de compilateur permettant d'exécuter ce code. Il s'agit seulement d'un modèle, relativement simple à comprendre, permettant à toutes les personnes travaillant sur un projet (les programmeurs, les testeurs, les personnes devant rédiger la documentation… et bien sûr le client) de se mettre d'accord sur le fonctionnement du programme sans entrer dans les détails de l'implémentation.

Pour se libérer de l'implémentation, la notation Z opère sur des objets mathématiques et les effets des fonctions du programme sont décrits par des formules du calcul des prédicats. Et pour simplifier la compréhension du modèle, elle décompose le programme en petits modules appelés schémas (chapitre II page 25).

I. Les notations mathématiques

Il n'est pas possible dans ce dossier de voir toutes les notions mathématiques (et leur notation) présentes en Z. Aussi, nous nous concentrerons sur celles qui sont particulièrement importantes et que nous utiliserons plus tard dans nos exemples de spécification de programme.

I.1. Ensembles et prédicats

I.1.1. Ensembles

La notation Z fournit comme ensembles de départ celui des entiers naturels $\mathbb{N}$ et de ses dérivés $\mathbb{Z}$ et $\mathbb{Q}$.

L'ensemble vide est noté par $\varnothing$.

Il est possible de créer de nouveaux ensembles :

- explicitement : $\{x_1,\ldots,x_n\}$;
- implicitement : $\{x : T \mid P \bullet E\}$ qui correspond à la notation usuelle $\{E(x) \mid x \in T \text{ et } P(x)\}$.

L'ensemble des sous-ensembles d'un ensemble S se note $\mathbb{P}S$ ou $\mathbb{P}_1 S$ pour les sous-ensembles non nuls.

Les opérations sur les ensembles sont notées de manières usuelles :

- l'inclusion : $\subseteq$
- l'inclusion stricte : $\subset$
- l'appartenance : $\in$
- la non-appartenance : $\notin$
- le produit cartésien : $\times$
- l'union : $\cup$
- l'intersection : $\cap$
- la différence: $\setminus$

I.1.2. Prédicats

Les termes sont construits de la même manière que ce que nous avons vu dans la première partie et les connecteurs s'écrivent comme d'habitude : $\neg$ (non), $\wedge$ (et), $\vee$ (ou), $\Rightarrow$ (implique) et $\Leftrightarrow$ (bi-implique ou équivalant).

Les formules peuvent se noter de manière quasi normale :

- $\exists\ T \bullet P$, où T introduit une variable locale (par ex $\exists\ x : \mathbb{N} \bullet \text{existe}(x)$) et où $\bullet$ se traduit par « tel que » ;
- $\forall\ T \bullet P$, où T introduit aussi une variable locale.

Elles peuvent aussi se noter de manière moins ordinaire :

- $\exists\ T \mid P \bullet Q$ équivalent à $\exists\ T \bullet P \wedge Q$;
- $\forall\ T \mid P \bullet Q$ équivalent à $\forall\ T \bullet P \Rightarrow Q$.

Le quantificateur d'existence unique se note $\exists_1$.

I.2. Relations

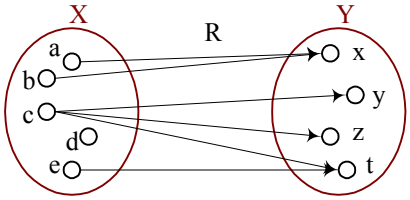
Les relations sont une notion très importante en Z. Il s'agit d'ensembles de paires ordonnées.

I.2.1. Définitions

On utilise le symbole $\leftrightarrow$ pour parler de relation.

Exemple I.1

Pour déclarer R comme relation entre les ensembles X et Y on note $R : X \leftrightarrow Y$

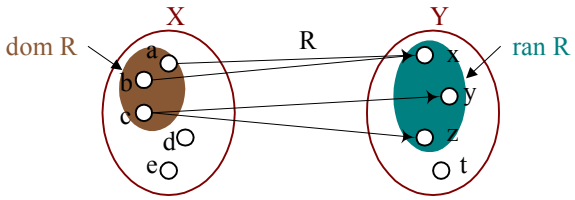


Pour signifier que (a,x) fait parti de cette relation, on écrit de manière équivalente :

- $(a,x) \in R$
- $a \rightarrow x \in R$
- $a\ R\ x$

Le domaine de départ se note `dom` R et celui d'arrivé `ran` R.

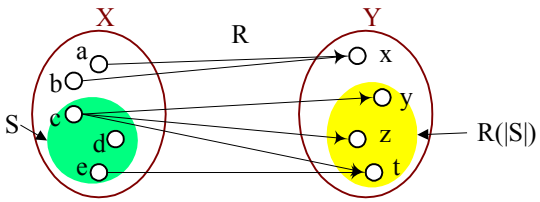
Exemple I.2



L'image d'un ensemble S d'éléments par la relation R se note $R(S)$.

Exemple I.3

Image par R :



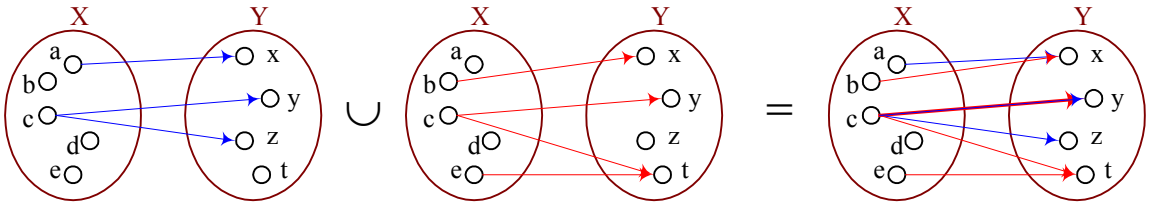
I.2.2. Opérations

I.2.2.1. Les opérations ensemblistes

Les trois opérations ensemblistes sont possibles sur les relations.

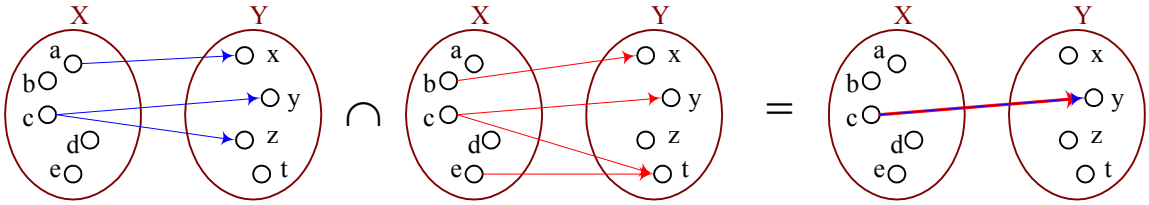
Exemple I.4

L'union :



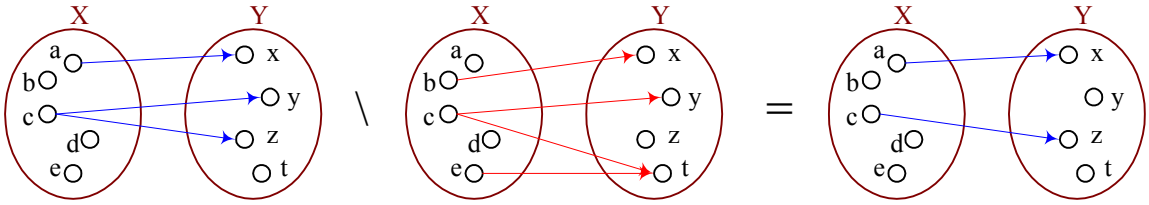
Exemple I.5

L'intersection



Exemple I.6

La différence



I.2.2.2. Restrictions sur les domaines

Il est possible de restreindre le domaine d'arrivée (resp. de départ) d'une relation R à un ensemble S grâce à l'opérateur $\triangleright$ (resp. $\triangleleft$). On remarque que $R \triangleright S$ (resp. $S \triangleleft R$) est un sous-ensemble de R.

Plus formellement :

- $R : X \leftrightarrow Y$ et $S : \mathbb{P} Y$

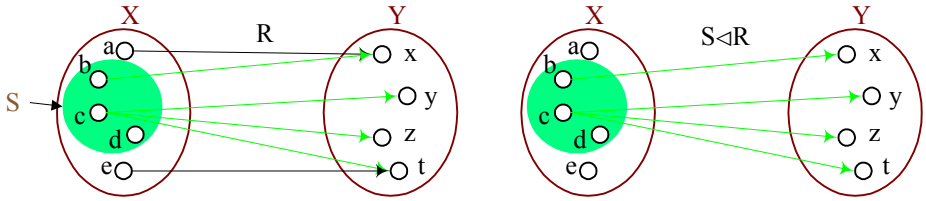
  $R \triangleright S = \{x : X ; y : S \mid x \mapsto y \in R \bullet x \mapsto y\}$

- $R : X \leftrightarrow Y$ et $S : \mathbb{P} X$

  $S \triangleleft R = \{x : S ; y : Y \mid x \mapsto y \in R \bullet x \mapsto y\}$

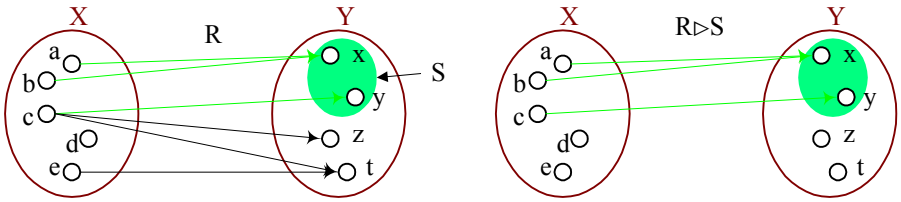
Exemple I.7

Restriction du domaine de départ



Exemple I.8

Restriction du domaine d'arrivée



I.2.2.3. Soustractions sur les domaines

La soustraction sur les domaines de départ (⩤) et d'arrivée (⩥) est une opération similaire à celle précédemment vue. Mais cette fois, la restriction se fait sur les éléments n'appartenant pas à un ensemble *S*.

Plus formellement :

- $R : X \leftrightarrow Y$ et $S : \mathbb{P} Y$

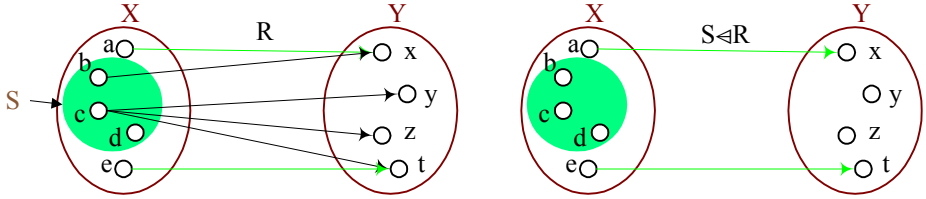
  $R \rhd S = \{x : X ; y : Y \mid y \notin S \land x \mapsto y \in R \bullet x \mapsto y\}$

- $R : X \leftrightarrow Y$ et $S : \mathbb{P} X$

  $S \lhd R = \{x : X ; y : Y \mid x \notin S \land x \mapsto y \in R \bullet x \mapsto y\}$

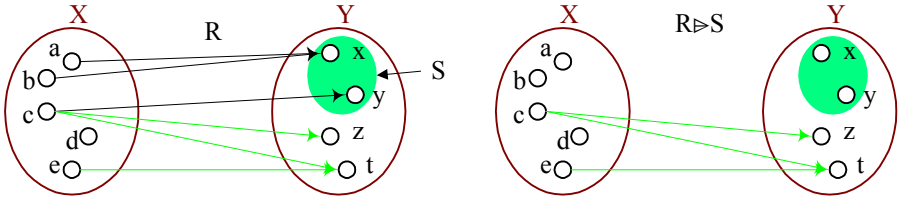
Exemple I.9

Soustraction sur le domaine de départ



Exemple I.10

Soustraction sur le domaine d'arrivée



I.2.2.4. La composition

Enfin, il est possible de composer deux relations *R* et *T* de deux manières :

- Par l'opérateur ; , dit « avant »

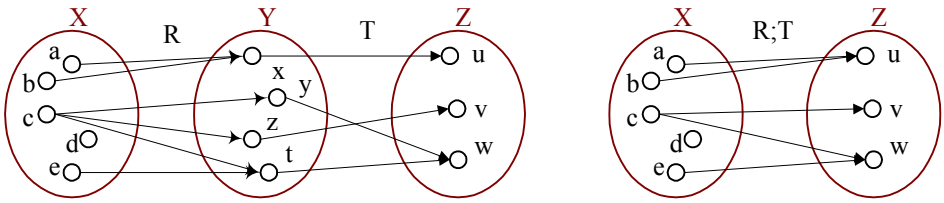
  Si $R : X \leftrightarrow Y$ et $T : Y \leftrightarrow Z$

  alors $R ; S = \{x : X ; y : Y ; z : Z \mid x \mapsto y \in R \land y \mapsto z \in T \bullet x \mapsto z\}$

- Par l'opérateur ∘ , dit « arrière » tel que $R \circ S = S ; R$

Exemple I.11

Composition avant



A partir de cette composition, on a immédiatement :

- l'itération : R^n

- la clôture transitive : R^+
- la clôture transitive réflexive : R^*

I.3. Fonctions

I.3.1. Définitions

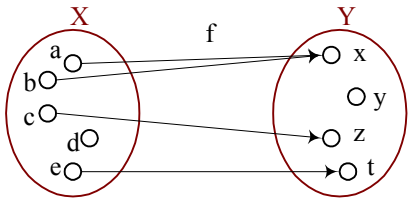
Les fonctions sont aussi une notion très importante de Z.

Il s'agit de relations dont chaque élément du domaine de départ est lié à, au plus, un élément du domaine d'arrivée. Il en existe deux types :

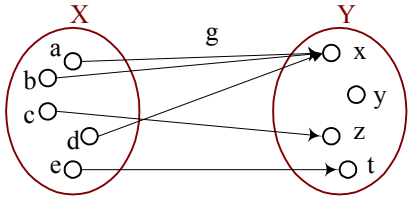
- les fonctions totales, notées $\rightarrow$, dont tous les éléments du domaine de départ possèdent une image ;
- les fonctions partielles, notées $\nrightarrow$, pour lesquelles ce n'est pas le cas.

Exemple I.12

Fonction partielle : $f : X \nrightarrow Y$



Fonction totale : $g : X \rightarrow Y$



En plus de cela, on peut aussi indiquer que la fonction est :

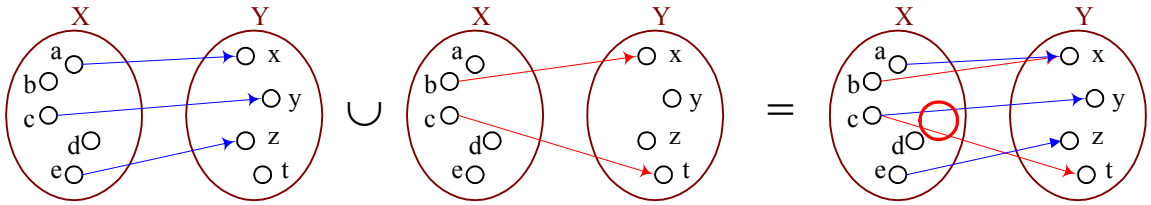
- injective : $\rightarrowtail$ ou $\nrightarrow$ (dans ce cas, on note $f^{\sim}x$ l'antécédent de x par f)
- surjective : $\twoheadrightarrow$ ou $\nrightarrow$
- bijective : $\rightarrowtail$ (pas de fonction partielle possible)

I.3.2. Opérations

Toutes les opérations sur les relations sont réalisables sur les fonctions exceptée l'union dont le résultat risque de ne pas être une fonction.

Exemple I.13

Union de deux fonctions

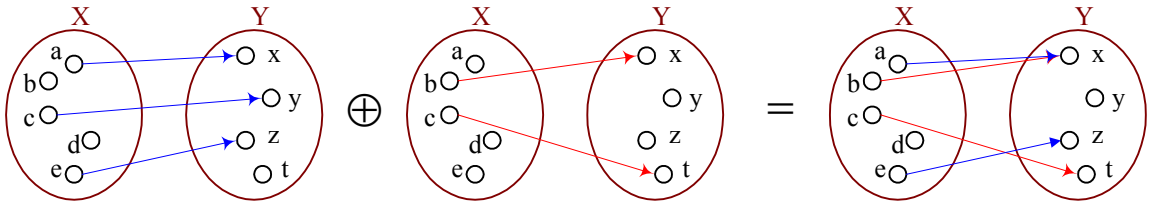


Dans l'union c possède deux images donc ce n'est pas une fonction.

Pour remédier à cela, Z propose une autre opération : $\oplus$ (overriding en anglais) définie par :

$$(f \oplus g)(x) = \begin{cases} g(x) & \text{si } x \in \text{dom } g \\ f(x) & \text{si } x \notin \text{dom } g \wedge x \in \text{dom } f \end{cases}$$

Exemple I.14



II. Le découpage des programmes

Bien que nous ayons vu quels outils Z met à notre disposition, nous ne savons toujours pas comment modéliser un programme. Le principe de la spécification en Z est de découper le programme en sous-tâches. Ces sous-tâches sont ensuite spécifiées grâce à des schémas. C'est l'ensemble de ces schémas qui constitue notre programme.

II.1. Les schémas

Il en existe de trois types :

- Pour spécifier une sous-tâche :

  titre[param]
  déclarations
  ---
  prédicats

- Pour créer des objets de porté globale, c'est à dire valable dans tous les autres schémas de la modélisation :

  déclarations
  ---
  prédicats

  Ces schémas sont dits axiomatiques.

- Pour établir des définitions génériques, c'est à dire à instancier dans un (ou plusieurs) autre(s) schéma(s) :

  [params]
  déclarations
  ---
  prédicats

  Ces schémas sont dits génériques.

Les prédicats les plus utilisés sont de loin ceux du premier type. Nous ne considèrerons donc que ceux-ci dans cette première approche de Z.

Ces schémas sont constitués de :

- Au moins une déclaration de variable locale au schéma ou une référence à un autre schéma qui contient des variables. Cela permet donc d'indiquer sur quels objets le module doit travailler.
- Eventuellement un ou plusieurs prédicat(s) pour spécifier ce que le module est sensé faire. Il est possible qu'il n'y ait pas de prédicats s'il s'agit d'un schéma d'état ou d'initialisation. En revanche, un schéma d'opération ne peut se passer de prédicats sinon cela reviendrait à dire que l'opération effectuée consiste à ne rien faire.

Rem : Pour les schémas axiomatiques et génériques aussi il est nécessaire d'avoir au moins une déclaration mais pas forcément d'avoir un prédicat.

Nous avons parlé juste au-dessus de trois nouvelles catégories de schéma :

- les schémas d'état servent à indiquer quelles sont les données que le programme va manipuler ;
- les schémas d'initialisation s'occupent d'initialiser ces données au lancement du programme ;
- les schémas d'opération spécifient le fonctionnement du programme.

Un programme contient à priori un schéma d'état et un schéma d'initialisation (éventuellement plusieurs pour une application un peu complexe (voir chapitre III.2.2 page 34)) et plusieurs schémas d'opérations.

Parmi les schémas d'opérations, on en distingue deux types :

- Ceux qui modifient l'état du programme. Il y a un Δ devant la référence du schéma auquel ils sont rattachés.
- Ceux qui retournent une réponse sans modifier le programme. Il y a un Ξ devant la référence du schéma auquel ils sont rattachés.

II.2. Exemples

Afin de bien comprendre la spécification en notation Z essayons de modéliser deux programmes assez simples.

II.2.1. Livre des records

Nous voulons ici créer une application qui mémorise tous les records mondiaux pour tous les sports. On veut également mémoriser les ex æquo.

II.2.1.1. Schémas d'état et d'initialisation

II.2.1.1.1. Travail préliminaire

Avant toute chose, on commence par indiquer les types utilisés dans le programme. Il y a certains types dont on ne veut rien connaître car on suppose qu'ils contiennent ce que l'on veut. Ce sont les types abstraits.

Ici nous en avons cinq :

- NOM qui devra contenir toutes les informations utiles (nom, prénom, date de naissance…) sur le ou les (s'il s'agit d'un sport en équipe) champion(s) du monde ;
- EPREUVE qui indiquera de quelle épreuve il est question ;
- DATE, la date à laquelle le record a été établi (peut être en jour, peut être en heure…)
- LIEU, le lieu où le record a été établi (pays, ville, stade par exemple)
- UNITE, l'unité de mesure dans les sports considérés.

Pour mettre ces nouveaux types dans notre modélisation, on les met entre crochets :

```
[NOM, EPREUVE, DATE, LIEU, UNITE]
```

Nous avons également besoin d'un nouveau type concret : dans quel sens le record s'améliore (en augmentant ou en diminuant ?). On indique ce nouveau type par :

```
meilleur == {"+","-"}
```

II.2.1.1.2. Schéma d'état

ALivreRecords

$qui\_qd\_ou : EPREUVE \rightarrow \mathbb{P}(NOM \times DATE \times LIEU)$
$mesure : EPREUVE \nrightarrow UNITE$
$sens : EPREUVE \nrightarrow meilleur$
$valeur : EPREUVE \nrightarrow \mathbb{Q}$
$connues : \mathbb{P}\ EPREUVE$
$existes : \mathbb{P}\ EPREUVE$

$\forall e : EPREUVE \circ e \in dom\ valeur \Leftrightarrow qui\_qd\_ou(e) \neq \emptyset$
$connues = dom\ valeur$
$existes = dom\ mesure$
$dom\ sens = dom\ mesure$
$dom\ mesure \supseteq connues$

Notre application doit être capable de nous dire qui est le champion du monde, quand et où le record a été établi et quel est ce record et dans quelle unité il est exprimé. C'est à cela que servent les fonctions `valeur`, `mesure` et `qui_qd_ou`. On peut faire deux remarques à propos de cette dernière :

- Elle ne peut être séparée à cause des ex æquo. Sinon on ne pourrait pas savoir qui a fait quel record et où.
- Elle n'a pas besoin d'être partielle car pour tous les sports dont on ne connaît pas de record il suffit de renvoyer par défaut $\emptyset$.

La fonction `sens` n'est pas directement utile pour l'utilisateur mais elle est indispensable pour contrôler que les nouveaux records sont bien meilleurs que les anciens.

Les ensembles `connues` et `existes` ne sont pas indispensables mais ils sont utiles pour connaître facilement si une épreuve respectivement a un record connu, existe dans la base de données (même si éventuellement il n'y a pas encore de record).

II.2.1.1.3. Schéma d'initialisation

AInitLivreRecords

ALivreRecords

$connues = \emptyset$

Tout d'abord, on peut constater que ce schéma initialise bien notre programme et non pas un autre schéma d'état possédant aussi une variable `connues`. Cela se voit au fait que dans le champ des déclarations on fait référence à `ALivreRecords`.

Ensuite, l'initialisation se contente de dire en mettant l'ensemble `connues` à vide qu'on ne connaît rien au lancement du programme. En effet, `connues`$=\emptyset$ est bien équivalent à : $\forall$`e:EPREUVE • qui_qd_ou(e)`$=\emptyset$ ce qui signifie clairement que la base de données est vide.

II.2.1.2. Schémas d'opérations

Voyons quelques opérations sur le livre des records.

Il y a bien d'autres opérations à modéliser mais nous nous limiterons à celles-ci pour voir un autre exemple ensuite.

II.2.1.2.1. Ajouter une épreuve

AAjouterEpreuve

ΔALivreRecords
épreuve? : EPREUVE
mesure? : UNITE
sens ? : meilleur

épreuve? ∉ existes
mesure' = mesure ⊕ {épreuve? → mesure?}
sens' = sens ⊕ {épreuve? → sens?}

Remarquons d'abord que ce schéma modifie l'état du programme. Cela se voit :

- D'une part avec le Δ devant `ALivreRecords`.
- D'autre part, avec la présence de noms de fonctions terminés par `'` qui correspondent aux nouvelles valeurs de ces fonctions après modification.

Les modifications sont faites selon les valeurs rentrées par l'intermédiaire des variables dans le champ des déclarations dont le nom se termine par `?` (`épreuve?`, `mesure?`, `sens?`).

Avant insertion de l'épreuve dans le livre, on vérifie (`épreuve? ∉ existes`) que celle-ci n'existe pas déjà.

Ce module réalise bien ce qu'on attend de lui :

- Il augmente l'ensemble des épreuves existantes et dont on connaît l'unité et le sens (`existes' ⊇ existes`)
- Il ne modifie pas l'ensemble des épreuves dont on connaît un record (`connues'=connues`)

II.2.1.2.2. Ajouter un premier record

APremierRecord

ΔALivreRecords
épreuve? : EPREUVE
valeur? : ℚ
qui? : NOM
quand? : DATE
où? : LIEU

épreuve? ∉ connues
valeur' = valeur ⊕ {épreuve? → valeur?}
qui_qd_ou' = qui_qd_ou ⊕ {épreuve? → {(qui?,quand?,où?)}}
connues' = connues ∪ {épreuve?}

Il s'agit encore une fois d'un schéma qui modifie l'état du programme et qui est assez similaire au schéma précédent. On peut cependant noter que la dernière ligne (`connues' = connues ∪ {épreuve?}`) n'est pas indispensable puisqu'elle est redondante avec l'avant-dernière ligne. Cependant, la redondance n'est pas interdite en notation Z ; elle est même encouragée si cela apporte plus de clarté.

II.2.1.2.3. Consulter le livre

AConsult

ΞALivreRecords
épreuve? : EPREUVE
quiqdou! : ℙ(NOM×DATE×LIEU)
valeur! : ℚ
unité! : UNITE

épreuve? ∈ connues
valeur! = valeur épreuve?
quiqdou! = qui_qd_ou épreuve?
unite! = mesure épreuve?

Cette fois, il s'agit d'un schéma ne modifiant pas l'état du programme mais retournant des valeurs. Les valeurs sont retournées par l'intermédiaire des variables dans le champ des déclarations dont le nom se termine par `!`.

II.2.2. Gestionnaire de tâches

Le but de l'application est de gérer un ensemble de n processus de sorte que le processeur ne soit pas toujours occupé avec le même.

Nous reprendrons cet exemple au chapitre III page 32, où il nous permettra de voir plusieurs choses intéressantes. Ayant déjà détaillé les schémas de l'exemple précédent, nous ne commenterons que les schémas intéressants dans cet exemple.

II.2.2.1. Schémas d'état et d'initialisation

II.2.2.1.1. Travail préliminaire

Puisque notre programme ne doit travailler que sur n processus, on commence par déclarer n en tant que paramètre global :

n : ℕ

La déclaration se fait dans un schéma dit axiomatique (cf. II.1 page 25) sans prédicat.

On pose ensuite les types avec lesquels on va travailler :

```
Pid == 1..n
NullId == 0
OptId == Pid ∪ {nullId}
```

II.2.2.1.2. Schéma d'état

AScheduler

courant : OptId (numéro de processus courant ou nullId s'il n'y en a pas)
ready : ℙPId (procressus en attente de traitement)
blocked : ℙPId (processus bloqué en attente d'un évènement)
free : ℙPId (numéros de processus libres)

<{courant}\{nullId}, ready, blocked, free > partition PId

On peut remarquer que le prédicat utilise une notation un peu spéciale. Celle-ci, spécifique à Z, permet de dire que les ensembles `{courant}\{nullId}`, `ready`, `blocked` et `free` doivent former une partition.

II.2.2.1.3. Schéma d'initialisation

AInitScheduler

AScheduler

---

courant = nullId
ready = ∅
blocked = ∅
free = PId (inutil à cause de la partition dans le schéma d'état mais améliore la lisibilité)

Au lancement du programme, il n'y a ni processus courant, ni processus bloqué, ni processus en attente d'être traité ; tous les numéros de processus sont libres.

II.2.2.2. Opérations

Voici une liste non exhaustive de ce qu'un gestionnaire de processus peut faire.

Passage d'un état sans processus en activité à un état actif :

ADispatch

Δ AScheduler
p! : PId

---

courant = nullId (Le processeur est inoccupé)
ready ≠ ∅ (Il reste des processus en attente de traitement)
courant' ∈ ready
ready' = ready \ {courant}
p! = courant'

Rem : On suppose que tous les modules de notre programme dialogue avec une entité supérieure qui est capable de manipuler les processus. Ainsi nos modules se contentent de dire à cette entité quel processus doit être activé, quel autre doit être bloqué…

Arrêt d'un processus qui a été actif suffisamment longtemps :

ATimeOut

Δ AScheduler
p! : PId

---

courant ≠ nullId (Un processus est en activité)
courant' = nullId
ready' = ready ∪ {courant}
p! = courant

Le processus courant est bloqué en attente d'un événement :

ABlock

Δ AScheduler
p! : PId

---

courant ≠ nullId (Un processus est en activité)
courant' = nullId
blocked' = blocked ∪ {courant}
p! = courant

Création d'un nouveau processus :

ACreat

Δ AScheduler
p! : PId

---

free $\neq \varnothing$ (Il reste au moins un numéro de processus libre)
p! $\in$ free
ready' = ready $\cup$ {p!}
free' = free \ {p!}

Destruction du processus courant :

ADestroyCurrent

Δ AScheduler
p? : PId

---

p? = courant
courant' = nullId
free' = free $\cup$ {p!}

Destruction d'un processus bloqué :

ADestroyBlocked

Δ AScheduler
p? : PId

---

p? $\in$ blocked
blocked' = blocked \ {p!}
free' = free $\cup$ {p!}

Destruction d'un processus en attente de traitement :

ADestroyReady

Δ AScheduler
p? : PId

---

p? $\in$ ready
ready' = ready \ {p!}
free' = free $\cup$ {p!}

Destruction de n'importe quel type de processus :

`AdestroyAll ≙ AdestroyCurrent ∨ AdestroyBlocked ∨ ADestroyReady`

Réveil d'un processus bloqué :

AWakeUp

Δ AScheduler
p? : PId

---

p? $\in$ blocked (Le numéro de processus entré correspond à un processus endormi)
blocked' = blocked \ {p!}
ready ' = ready $\cup$ {p!}

III. Le raffinement

Ce qui est intéressant dans la notation Z, c'est qu'il est possible, à partir d'une modélisation très abstraire de raffiner cette modélisation afin de se rapprocher de l'implémentation du programme.

III.1. De l'abstrait vers le concret

L'idée du raffinement est de se diriger d'une modélisation abstraite dont on est sûr vers une modélisation plus concrète (c'est à dire plus proche de l'implémentation). Cependant, comment peut-on être aussi sûr de cette nouvelle modélisation qu'on l'était de la précédente ? C'est là tout l'intérêt de Z. En effet, il est possible de démontrer si une modélisation concrète correspond bien à une modélisation plus abstraite. De plus cela peut se faire de manière automatique.

Puisque pour une modélisation abstraite, il existe plusieurs implémentations concrètes, le principe de la vérification est de retrouver chaque opération abstraite grâce aux opérations concrètes et à un schéma de relation entre les deux modélisations.

III.2. Exemples

Pour illustrer le raffinement, reprenons l'exemple du gestionnaire de processus. Nous allons essayer d'implémenter les ensembles de processus par des chaînes. Donc dans un premier temps définissons ce que peut être une chaîne.

III.2.1. Les chaînes

III.2.1.1. L'initialisation

Chain

start : OptId	(Début de la chaîne)	
end : OptId	(Fin de la chaîne)	
links : PId ⤖ PId	(Les liens de la chaîne)	(1)
set : ℙPId	(L'ensemble des noeuds de la chaîne)	

set = dom links ∪ ran links ∪ ({start} \ {nullId})	(2)
links = ∅ ⇒ start = end	(3)
links ≠ ∅ ⇒ {start} = {dom links \ ran links} ^ {end} = {ran links \ dom links}	(4)
∀ e:set | e ≠ start ∘ start → e ∈ links$^+$	(5)

Les points :

- 1 et 4 garantissent que la chaîne ne comporte pas de boucle
- 2 définit `set` dans les trois cas possibles :
  - chaîne vide : `set = ∅`
  - chaîne de longueur 1 : `set = {start}`
  - chaîne de longueur n>1 : `set = dom links ∪ ran links`
- 3 interdit que la chaîne soit constituée de deux singletons
- 5 garantit qu'il existe une suite de liens non nulle entre `start` et `end` (pas de discontinuité dans la chaîne).

InitChain

Chain

---

start = nullId (Commence avec une chaîne vide)

III.2.1.2. Les opérations

Remplissage de la chaîne avec tous les processus :

FullChain

Chain

---

set = PId

Insertion d'un élément dans une chaîne vide :

PushEmpty

Δ Chain
p? : PId

---

end = nullId (La chaîne est vide)
end' = p?
start' = p?

Insertion d'un élément à la fin d'une chaîne non vide :

PushNonEmpty

Δ Chain
p? : PId

---

end ≠ nullId (La chaîne est non vide)
p? ∉ set (L'élément n'est pas déjà dans la chaîne)
links' = links ⊕ {end → p?}
end' = p?

Suppression du dernier élément d'une réduite à un singleton :

PopSingleton

Δ Chain
p! : PId

---

start ≠ nullId (La chaîne n'est pas vide)
links = ∅ (La longueur de la chaîne est < 2) (Donc la chaîne est réduite à un singleton)
start' = nullId
p! = start

Suppression de la tête d'une chaîne non réduite à un singleton :

PopMultiple

Δ Chain
p! : PId

---

links ≠ ∅ (La chaîne contient plus d'un élément)
start' = links start
links' = {start} ⩤ links
p! = start

Suppression de la tête d'une chaîne :

`Pop ≙ PopSingleton ∨ PopMultiple`

Suppression d'un processus de la chaîne s'il se trouve en tête :

DeleteStart

Δ Chain
p? : PId

---

p? = start
$\exists$p! : PId $\circ$ Pop

Rem : Le quantificateur existentiel permet ici de masquer la sortie de Pop

Suppression d'un processus de la chaîne s'il se trouve en fin :

DeleteEnd

Δ Chain
p? : PId

---

p? $\neq$ start
p? = end
links' = links $\rhd$ {end}
end' = links$^{\sim}$ p?

(Permet d'éviter un conflit entre DeleteStart et DeleteEnd lorsque la chaîne est réduite à un singleton)

Suppression d'un processus de la chaîne s'il se trouve au milieu:

DeleteMiddle

Δ Chain
p? : PId

---

p? $\notin$ {start,end}
p? $\in$ set
links' = {p?} $\lhd$ links $\oplus$ links$^{\sim}$ p? $\rightarrow$ links p?

Suppression d'un processus de la chaîne :

`Delete ≙ DeleteStart ∨ DeleteEnd ∨ DeleteMiddle`

III.2.2. Raffinement du gestionnaire de processus

III.2.2.1. Schémas d'état et de relation

Pour ranger dans des listes les trois ensembles du gestionnaire de processus, il nous faut trois nouvelles instances de `Chain` qu'on obtient par renommage des variables :

- `ReadyChain ≙ Chain [rstart/start,rend/end,rlink/link,rset/set]`
- `BlockedChain ≙ Chain [bstart/start,bend/end,blink/link,bset/set]`
- `FreeChain ≙ Chain [fstart/start,fend/end,flink/link,fset/set]`

CScheduler

ReadyChain
BlockedChain
FreeChain
courant : OptId
ChainStore : PId $\rightarrow$ OptId

<{courant}\{nullId}, rset, bset, fset > partition PId
rlinks = rset $\lhd$ ChainStore $\rhd$ rset
blinks = bset $\lhd$ ChainStore $\rhd$ bset
flinks = fset $\lhd$ ChainStore $\rhd$ fset
courant $\neq$ nullId $\Rightarrow$ ChainStore courant = nullId

(Le processus courant est isolé, il ne doit pointer vers aucun autre processus)

CScheduler est donc un schéma d'état plus proche de l'implémentation que AScheduler. Pour pouvoir comparer les deux spécifications, il faut construire un schéma de relation qui permet de retrouver le modèle abstrait à partir du modèle concret. Ce schéma est assez simple :

RelationAC

AScheduler
CScheduler

---

ready = rset
blocked = bset
free = fset

Rem : Dans le schéma d'état Ascheduler (page 29), il y a une quatrième variable : courant. Etant donné que cette variable existe dans les deux spécifications, et puisque sa valeur est identique, il est possible de l'omettre dans le schéma de relation.

III.2.2.2. Schéma d'initialisation

Pour initialiser le programme, nous allons réutiliser des opérations écrites pour les chaînes :

ReadyInitChain ≙ InitChain[rstart/start,rend/end,rlink/link,rset/set]

BlockedChain ≙ InitChain[bstart/start,bend/end,blink/link,bset/set]

FreeFullChain ≙ FullChain[fstart/start,fend/end,flink/link,fset/set]

Grâce à cela on peut écrire l'initialisation aussi simplement que AInitChain (page 30) :

CInitScheduler

CScheduler

ReadyInitChain
BlockedInitChain
FreeFullChain
courant = nullId

On voit immédiatement que, compte tenu du schéma de relation, CInitChain respecte AInitChain :

- courant est initialiser de la même manière
- ReadyInitChain met rstart à nullId, donc rset à ∅. Or d'après RelationAC, ready = rset. Donc ready est bien initialisé.
- On applique le même raisonnement pour blocked.
- FreeFullChain met fset à PId. Or d'après RelationAC, free = fset. Donc free est bien initialisé.

III.2.2.3. Opération

La vérification du respect des opérations abstraites étant assez rébarbative et habituellement faite par une machine, nous nous contenterons de traduire ADispatch (page 30) dans la spécification plus concrète. Pour cela, définissons déjà :

PopReadyChain ≙ PopChain[rstart/start,rend/end,rlink/link,rset/set]

Puis :

CDispatch

Δ CScheduler
p! : PId

---

courant = nullId (Le processeur est inoccupé)
rset ≠ ∅ (Il reste des processus en attente de traitement)
PopReadyChain
courant' = p!

Vérifions que `CDispatch` respecte `ADispatch` :

- Puisque `rset=ready`, les pré-conditions sont les mêmes.
- `PopReadyChain` retourne `rstart` donc un élément de `rset`. Donc `courant ∈ rset`. Or `rset = ready`, donc `courant ∈ ready`.
- De plus, `PopReadyChain` supprime `rstart` de `rset`, donc `ready' =ready\{courant}`.

Donc `CDispatch` respecte `ADispatch` .

Conclusion

Nous venons de passer rapidement en revu le calcul des prédicats pur qui est le système à la base de toutes les théories mathématiques un peu évoluées. Cependant ce système est trop simple pour fonder les mathématiques. Il ne contient que des prédicats d'ordre 1. Il ne contient pas de constructeurs permettant de construire une nouvelle variable d'individu à partir d'autres, comme la fonction successeur par exemple. Il ne permet pas non plus de raisonner sur les prédicats eux-mêmes. C'est la raison pour laquelle tous les langages de spécification s'appuient sur la logique du 1er ordre et l'étendent avec d'autres notions comme les opérations ensemblistes dans le cas de la notation Z.

A propos de cette dernière, il est difficile de tirer des conclusions sérieuses. En effet, la présentation de Z a été pour le moins rapide et est très loin d'être exhaustive. De plus, il serait intéressant de le comparer à d'autres systèmes de modélisation. Aussi, nous nous contenterons de souligner deux de ses avantages. D'abord il s'agit d'une notation assez facile à prendre en main, ce qui est important puisqu'il s'agit d'un outil de dialogue privilégié entre les différents intervenants sur le projet qui peuvent avoir des cultures très différentes. Ensuite, Z permet de faire varier le degré d'abstraction désiré qui n'est à priori pas le même pour tous les intervenants.