Fondations mathématiques : Lazi

Emmanuel Chantréau,
email : echant.lazi@bobu.eu

5 août 2016

"*La simplicité est la sophistication ultime.*" Léonard de Vinci

Table des matières

Chapitre 1

Introduction

Ce texte définit une fondation des mathématiques très différente de celles généralement rencontrées : sans ensemble ni propriété au sens usuel, pas le même calcul des prédicats ni le même système de déductions que le calcul des séquents. Aborder ces fondations peut avoir le même effet pour un mathématicien qu'un chimiste du passé venant de découvrir que les atomes sont composés de particules élémentaires, c'est fort déroutant mais j'espère que le lecteur en sentira toute la puissance potentielle.

On pourra trouver ce texte inutilement pointilleux. C'est pour deux raisons qui se rejoignent : Contrairement aux mathématiques classiques la définition de Lazi est utilisée à l'intérieur même de Lazi et doit donc être formalisée. D'autre part, à partir de la page 15 les définitions sont incorporées au logiciel de calcul "lazi-compute". C'est à dire que la définition des fondations mathématiques que vous lirez sera directement les sources (en lazi) de fonctions calculant la validité d'une preuve lazi. Cela est possible car 6 des 7 règles de déductions sont calculatoires.

Je conseille vivement d'aborder ce texte sans entrer dans les détails techniques en premier abord, ils ne doivent pas cacher l'architecture globale de ces fondations.

Chapitre 2

Structure de la définition de Lazi

Cette partie décrit quelles sont les étapes de la définition de la fondation des mathématiques “Lazi”.

Avant cela nous allons récapituler les étapes de la définition courante des mathématiques utilisées actuellement :

1. définition du langage formel
   (a) définition du système de règles permettant de définir le langage (par exemple le système EBNF)
   (b) utilisation du système de règle pour définir :
      i. un alphabet
      ii. les formules valides
2. définition du système de production de vérités
   (a) définition du système de règles primaires permettant de produire des vérités : (par ex. le calcul des séquents)
   (b) utilisation du système de règles primaires pour définir des règles secondaires :
      i. calcul des prédicats
         - calcul des propositions
         - calcul des quantificateurs (règle d’introduction et d’élimination)
      ii. axiomes (par ex. ZFC)

Structure de la fondation mathématique Lazi :

1. définition du langage formel
   (a) définition du système de règles permettant de définir le langage : EBNF
   (b) utilisation du système de règles pour définir :
      i. les mots
      ii. les formules valides
2. définition du système de production de vérités
   (a) définition de ce qu’est une liste de vérité
   (b) définition de ce qu’est une règle de déduction
   (c) définition des six règles de déductions primaires
   (d) définition de la septième règle.

Remarque : Pour des raisons pédagogiques la dernière règle est donnée avec la liste des 6 premières et les notations nécessaires à cette règle sont définies par la suite.

Chapitre 3

Système de règles permettant de définir le langage formel

Nous utilisons ici pour sa simplicité et rigueur le formalisme EBNF (“Extended Backus-Naur Form”, voir `http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form`).

Chapitre 4

Syntaxe Lazi

4.1 Les mots

basicWords = "equal" | "distribute" | "one" | "zero" | "if"

4.2 Les formules

basicFormula = basicWords | (“(” , basicFormula , “)” , “(” , basicFormula , “)”) ;

Le langage Lazi est constitué des expressions basicFormula.

Remarque : L’assemblage est uniquement par deux : la partie gauche est la fonction et la partie droite est l’argument. Nous avons donc un système à la lambda calcul : “$((if)(one))(zero)$” signifie “on applique la fonction if à l’argument one et le résultat est vue comme une fonction que l’on applique à zero”.

Chapitre 5

Syntaxe étendue pour les notations

Nous allons définir ici une sytaxe plus riche qui nous servira pour les notations.

5.1 Les mots

alphaLetter = toute lettre unicode des catégories “Lettre, majuscule” et “Lettre, minuscule”
numericLetter = toute lettre unicode de la catégorie “Chiffre décimal”
alpha = (alphaLetter | ":") , {alphaLetter, numericLetter , ":" }
word = alphaLetter { alpha }

5.2 Les formules

formula = word | (“(” , formula , “)” , “(” , formula , “)”) ;

Chapitre 6

Lazi-Nota : les notations

6.1 Présentation

Lazi n'est pas humainement simple à lire. C'est pourquoi nous introduisons très tôt des notations.

Les langages mathématiques usuels possèdent un système de notation. Ce sera aussi le cas ici, et avec le même système.

Lazi-Nota est définit par un ensemble des symboles utilisables et un ensemble de règles de notations.

Ces règles, appelées "notations", permettent de traduire des formules Lazi-Nota en formules Lazi-Nota plus simples pour aboutir au final à une "formula" unique, quelque soit l'ordre de traduction appliqué. De plus toutes les définitions par notation de ce texte se traduisent en "basicFormula".

Les symboles suivants sont des symboles de ponctuations :
( ) [ ] , ; : ‘ ’
En dehors des symboles de ponctuations, les autres symboles font parti du même mot s'il n'y a aucun espace entre eux. Par exemple "=w" est un mot.

Remarque : Lazi est prévu pour être extensible, nous avons besoin de ne pas figer le langage et donc de ne pas bloquer des symboles comme "=" dans une ou deux utilisations (traditionnellement "=" et "==" en informatique). Il faudra donc veiller à changer d'habitude et écrire "x = y" plutôt que "x=y".

Définition informelle n° 1, word utilisé : Un word w est utilisé dans une formule Lazi-Nota f s'il est un des mots de la traduction de f en formula.

6.2 Symboles Lazi-Nota

Les symboles de Lazi-Nota sont tous les symboles Unicode imprimables plus le symbole d'espace ordinaire ‘ ’.

Remarque : "Unicode" est une table de symboles attribuant un code (un ou plusieurs nombres) à quelques milliers de symboles. (voir `http://en.wikipedia.org/wiki/Unicode`).

6.3 Définir des notations

La notation suivante sera utilisée pour la suppression des parenthèses inutiles :

Exemple : Notation n°1 : variable(s): x, y
Condition: y est un word
“ $x\ y$ ” → “ $x(y)$ ”.

Une définition d'une notation consiste à fournir une/des règle de traduction, il nous faut donc :

- une formule à traduire A
- sa traduction B
- des variables $v_1, \ldots, v_n$ (éventuellement aucune) pour avoir des règles génériques. Ces variables sont constituées de lettres et chiffres de l'alphabet anglais et commencent par une lettre. Dans l'exemple précédent ce sont ‘x’ et ‘y’.
- des conditions sur les valeurs possibles des variables

La définition d'une notation aura la syntaxe suivante : :

Notation n°2 : variable(s): $v_1, \ldots, v_n$
Condition: conditions sur $v_1, \ldots, v_n$
" A " → " B ".

et le sens suivant :

- Si on a n formules Lazi-Nota (n peut être égal à 0) $f_1, \ldots, f_n$,
- si les conditions sont valides en remplaçant $v_1, \ldots, v_n$ par $f_1, \ldots, f_n$ dans les conditions,
- alors A où l'on a remplacé $v_1, \ldots, v_n$ par $f_1, \ldots, f_n$ se traduit par B où l'on a remplacé $v_1, \ldots, v_n$ par $f_1, \ldots, f_n$.

6.4 Les notations de définition d'un mot

La plupart des ajout de notations seront des définitions de mots, par exemple si nous voulons définir la notation "$identity" de sorte qu'elle soit remplacée par $if\ zero\ zero$, nous écrirons ceci :

/*
Fonction 'identité'
*/
$**Def identityF** = $if\ zero\ zero$

Remarque : Nous utiliserons couramment le signe $ pour les notations de manière à ne pas interférer avec les mots lazi.

Une définition de mot (par notation) est une notation utilisant une syntaxe spéciale :

/*
comment
*/
$**Def f** = *value*

Où "f" est un word et value une formule Lazy-Nota et où f est un mot inutilisé dans value. On la traduit en la notation :

Notation n°3 : variable(s): ;
" #1 " → " *value* ".

6.5 Notations pour les mots de base

Notation n°4 : variable(s): x, y , associativité: no, priorité: 10
"$x = y$" → "*equal x y*".

Exemple : "$a = b$" se traduira par "*equal a b*".

Notation n°5 : variable(s): ;
" 1 " → " *one* ".

Notation n°6 : variable(s): ;
" 0 " → " *zero* ".

6.6 Notations pour les opérateurs

Lazi-Nota pourra utiliser des opérateurs infixs associatifs ou non et ayant une priorité ("précédence" en informatique) sur les opérateurs. La priorité d'un opérateur est toujours moindre par rapport à la priorité de l'application fonction/argument Lazi, donc "$x\ op\ y\ z$" se lit "$x\ op$ ($y\ z$)".

Nous n'utiliserons un nombre très restreint de niveaux de priorité de manière à ne par rendre le langage difficile à mémoriser.

Exemple de définition de l'opérateur "=" :

$**Nota** = infix left 10 equal

définit l'opérateur "=" comme infix, associatif à gauche, de priorité 10 et égal à la formule "equal". Donc "x = y" sera traduit par "equal x y".

Les opérateurs peuvent être "infix", "prefix", "postfix", "nonOp" (sans argument, c'est l'équivalent de la définition d'un mot mais sans obligation d'un $ devant. Le sens de l'associativité est un argument uniquement pour les infix.

6.7 Suppression des parenthèses inutiles

Notation n°7 : variable(s): x, y
Condition: y n'est de la forme ni "$(a)(b)$" ni "$a\ b$" (a et b des formules Lazi-Nota)
" $x\ y$ " → " $(x)(y)$ ".

Remarque : Donc si y et z sont des word, "x y z" se traduira par "(x y)(z)', qui se traduira par "((x)(y))(z)"

Remarque : Si une notation traduisait y en "a b" alors on pourrait traduire "x y" en "x a b" qui se traduit en "(x a)(b)". Cela poserait problème car on pourrait traduire aussi en "(x)(y)" puis "(x)(a b)".
Nous voyons donc que les notations futures devront parenthéser certains résultats de traduction (quand le résultat est de la forme "a b" ou "(a)(b)"). La notation devrait donc être y -> "(a b)". Mais alors on introduit un nouveau problème : les parenthèses ne sont plus par paire (comme dans "(x)(y)'). Il nous faut donc des notations supplémentaires pour gérer les parenthèses en trop.

Notation n°8 : variable(s): x;
" $((x))$ " → " (x) ".

Remarque : En reprenant l'exemple de la remarque ci-dessus, on traduit $x\ y \rightarrow x\ (a\ b) \rightarrow (x)((a\ b)) \rightarrow (x)(a\ b) \rightarrow (x)((a)(b))$

Nous allons ajouter une autre notation pour éclaircir la lecture des parenthèses :

Notation n°9 : variable(s): $x_1, ..., x_n$;
" $(x_1, x_2, ..., x_n)$ " → " $(x_1)(x_2)...(x_n)$ ".

Remarque : Cette notation rejoint en quelque sorte la notation des mathématiques classiques f(x,y,z) où l'on applique une fonction à des argumenst, puisque f(x,y,z)=f(x)(y)(z). Mais il ne faudrait pas confondre les notations, par exemple si f est un mot en lazi f(x,y,z)=(f,x,y,z)=f(x)(y,z)

Chapitre 7

Définition du système de production de vérités

7.1 Une liste de vérité

Nous utilisons le terme de liste au sens usuel : une suite finie et ordonnée d'éléments.

- Une liste vide est une liste de vérité.
- Si l est une liste de vérités, si f est une formule déduite à partir d'une règle de déduction lazi et de l, alors la liste où f est ajoutée au début de l est une liste de vérités.

Remarque : On reconnaît là le processus mathématique habituel : on déduit des vérités que l'on peut utiliser pour en déduire d'autres.

7.2 Les règles de déduction

Voici un exemple de règle de déduction : $\frac{x, x = y}{y}$ qui se lit : si on a les conditions "x" et "x=y" alors on déduit "y".

Une règle de déduction est composée :

- De conditions : une liste de formules (entre zéro et deux éléments).
- D'une formule conclusion.

La liste de variable(s) d'une règle est la liste des mots "x", "y" et "z" figurant dans les formules des conditions et de la conclusion (les noms des variables sont limités à ces trois là).

Soit l une liste de vérités, r une règle de déduction Lazi, n le nombre de variables dans r et $v_1, ..., v_n$ n formules. Soit f la formule où la conclusion de r a eu ses variables remplacées par les valeurs (v_1 pour x, v_2 pour y et v_3 pour z). On dit que f est déduite à partir de r et l si les conditions, où l'on aura remplacées les variables par les valeurs, sont dans l.

Exemple : Si l est la liste de 3 formules "if 1 1 1 = 1", "if 1 0 0 = 0" , "(if 1 1 1 = 1) = (1 = 1)", en utilisant la règle $\frac{x, x = y}{y}$ avec v_1=(if 1 1 1 = 1) et v_2=(1 = 1), on en déduit 1 = 1.

Chapitre 8

Les règles de déduction Lazi

La dernière règle est donnée ici à titre d'information, les notations nécessaires pour la définir sont l'objet du reste de cet article.

Nom	Règle
égalité et vérité	$\frac{x, x = y}{y}$
arguments égaux	$\frac{x = y}{z\ x = z\ y}$
fonctions égales	$\frac{x = y}{x\ z = y\ z}$
'distribute'	$distribute\ x\ y\ z = (x\ z)\ (y\ z)$
'if' sur 1	$if\ 1\ x\ y = x$
'if' sur 0	$if\ 0\ x\ y = y$
descente	$\frac{checkProofOfFormula\ laziMaths\ laziAxioms\ x\ y}{translateFormulaMaths\ laziMaths\ y}$

Je vais donner une idée du sens de la dernière règle, des explications plus détaillées seront données dans cet article. Voici 3 versions de plus en plus précises :

1. Une preuve en Lazi prouve une vérité.
2. Si x est une représentation d'une preuve Lazi alors la traduction de la conclusion de x est vrai.

Remarque : Les premières parenthèses pour distribute sont formellement inutiles.

Remarque : On peut vérifier les propriétés de réflexivité, symétrie et transitivité de l'égalité.
Par exemple, en résumé, pour la réflexivité (x est une expression basicFormula quelconque) :

- par "if sur vrai" : $if\ 1\ x\ x = x$
- par "arguments égaux" : $equal\ \left(if\ 1\ x\ x\right) = equal\ x$
- par "fonctions égales" : $equal\ \left(if\ 1\ x\ x\right)\ x = equal\ x\ x$
- par "égalité et vérité" : $equal\ x\ x$

Chapitre 9

Vision globale de Lazi

9.1 Les 6 premières règles

Nous verrons rapidement que les 6 premières règles permettent d'utiliser des fonctions et des sortes de propriétés sur des **choses** (en Lazi, il n'y a pas d'ensemble et d'élément mais des "choses"). "sortes de propriété" car nous n'avons pas la forme de tiers exclue classique.

Nous définirons des fonctions pour créer, modifier et tester des structures telles que des listes, des représentations de formules ou de preuves.

9.2 Niveau des représentations

Nous allons représenter les mots et les formules à l'intérieur de Lazi. Pour distinguer les formules déjà définies des représentations des formules dans Lazi, nous utiliserons un système de niveau : le niveau 0 sont pour les mots et formules déjà définies. Le niveau 1 pour les représentations dans Lazi. Pour faire court nous pourrons aussi les dénommer "n-formules" pour n un entier naturel. On pourra aussi, à l'intérieur d'une 1-formule , représenter une formule, ce sera alors une 2-formule etc.

Nous utiliserons cette même notion de niveau pour les preuves. Par exemple «Par la règle "vrai" nous déduisons "1 = (1 = 1)"» est une 0-preuve.

Se promener dans les niveaux des preuves et des formules est autant une question d'habitude que pour les imbrications d'ensembles ou de fonctions.

Lazi ne définit pas de quantificateurs ou calcul de prédicat fort comme les mathématiques classiques, il les remplace par les représentations de preuves, qui comme pour les quantificateurs, auront besoin d'être imbriquées.

9.3 En Lazi, choses = (prédicats ∪ ensembles)

En mathématique classique, si P est un prédicat, f une fonction et x un élément du domaine de f, la formule suivante est syntaxiquement incorrecte "$P \Rightarrow f(x)$". En Lazi, "$if\ if$" est une chose (sans propriété ni intérêt spécial), et l'on peut prouver "*equal equal equal*".

9.4 La règle de la descente

Cette règle permet de "faire des mathématiques" dans une représentation de preuve (construire une 1-preuve pour prouver une 1-formule dont la traduction en une 0-formule sera une vérité). La propriété "isSimpleProof" définit ce qu'est une 1-preuve. Une 1-preuve aura les 6 mêmes premières règles déjà formulées, mais à la place de la règle de la descente simple il y aura la règle de la descente pleine (plus sophistiquée que la simple) ajoutant une forme primitive de tiers exclu et de généralisation.

Pourquoi avoir "cacher" des règles (un peu) plus complexes ? La description des fondations au niveau-0 doit rester la plus simple possible, que ce soit pour éviter les ambiguïtés comme pour pouvoir être interprété par un logiciel avec le moins de bug possible. Il aurait été lourd de vouloir décrire les règles de niveau-1 dans le langage courant puis de recommencer la description en Lazi, alors que Lazi permet de les décrire avec concision et clarté.

Nous allons maintenant présenter la règle de la descente (pleine).

9.5 La puissance sans la prétention

On entend ici par "puissance" le fait de fournir des outils permettant de faciliter la construction de preuves, et par "sans prétention" le fait de ne rien prouver de plus que les autres règles.

La règle de descente apporte de la puissance sans avoir de prétention. Cette affirmation n'est pas encore démontrée mais me parait intuitive au regard des règles données ci-dessous

9.6 Règle de la descente

En plus d'affirmer que toute 1-preuve prouve une 0-formule, elle contient aussi deux autres affirmations :

- Si on a prouvé if x y z c'est que l'on peut prouver $x = 1$ ou $x = 0$.
- Si une preuve contient une variable v alors en remplaçant v par une valeur la preuve reste valide.

La première affirmation est une forme de tiers exclu qui nous permettra d'explorer les cas $x = 1$ et $x = 0$.

En regardant les 6 règles de base il est clair qu'une formule "if ..." ne peut être prouvée que par les deux dernière règle. Si ce n'était pas le cas alors, comme "if" n'a pas de sens pour les autres règles, on pourrait remplacer "if" par n'importe quelle valeur et arriver à une absurdité. (Ceci n'est pas une démonstration mais une idée de démonstration). Nous "voyons" donc que la règle du tiers exclu ne permet pas de déduire plus de vérité.

La deuxième affirmation est équivalente à la quantification universelle. Elle s'appuie sur le fait que v n'ayant aucune propriété particulière, le remplacer par une valeur dans la preuve garde la preuve valide. Il me parait donc intuitif (et il reste à démontrer) que cette règle ne permet pas de déduire plus de vérité.

On peut donc "voir" que la règle de la descente n'ajoute aucune vérité à celles déductibles des règles de base.

Chapitre 10

Les outils de définition

10.1 Logiciel Lazi

À partir de ce point les définitions et les notations simples sont lues (directement du code source LaTeX de ce texte) par le logiciel Lazi-soft pour les incorporer en tant que définition de Lazi. Ce logiciel permet de construire et vérifier des preuves Lazi. Le lecteur apercevra quelques très rares symboles " ?" (par exemple $?x$) qui est une indication d'ordonnancement de calcul pour Lazi-soft.

10.2 Définition de notations pour des fonctions de placement

Nous avons l'habitude en mathématique de définir des fonctions avec des variables, par exemple $f : x \to 1 + x \times 3$ définit un objet mathématique f vérifiant par exemple "$f(2) = 1 + 2 \times 3$".

Nous allons utiliser une forme similaire de définition de fonction. Pour cela nous allons partir de deux fonctions (if et distribute) et voir que par calcul elles permettent de placer n'importe quel argument où l'on veut dans une formule quelconque.

Contrairement à l'habitude en mathématique, nous allons donc ici non pas définir des fonctions par un procédé de remplacement mais par un procédé de placement.

/*
Fonction 'identité'
*/
$**Def identityF** = if 0 0

/*
Fonction constante (le premier argument est la constante), on aura constantF x y = x
*/
$**Def constantF** = if 1

Notation n°10 : variable(s): x
Condition: x est un "word"
" $F $x \to x$ " → " #1 ".

Notation n°11 : variable(s): x, y
Condition: x est un word inutilisé dans y.
" $F $x \to y$ " → " #1 y ".

La notation ci-dessous est optionnelle. Elle permet de simplifier les formules. Comme elles sont directement utilisées pour faire des calculs et l'objet de preuves, cela a un intérêt.

Notation n°12 : variable(s): x, y
Condition: x est un "word" inutilisé par y (qui n'apparait pas dans y).
" $F $x \to y\ x$ " → " y ".

Notation n°13 : variable(s): x, y, z
Condition: x est un "word" qui est utilisé dans y z et qui est utilisé dans y s'il est égal à z (pour ne pas interférer avec les deux notations précédentes).
" $F $x \to y\ z$ " → " $distribute$ \($F $x \to y$\) \($F $x \to z$\) ".

Exemple : L'exemple ci-dessous élude beaucoup de petits raisonnements, il peut servir à s'exercer avec les règles Lazi. On peut déjà chercher à vérifier les 3 propriétés classiques de l'égalité : réflexivité, symétrie, transitivité.
$\$F\ x \to x\ a$
est la notation pour *distribute* $\left(\$F\ x \to x\right)\ \left(\$F\ x \to a\right)$
qui se traduit en *distribute* #1 (#1 a).
Donc $\left(\$F\ x \to x\ a\right)\ b$
se traduit en *distribute* #1 (#1 a) b
qui est égal à (#1 b) (#1 $a\ b$)
Comme (#1 b) = b, (#1 b) (#1 $a\ b$) = b (#1 $a\ b$).
Comme (#1 $a\ b$) = a, b (#1 $a\ b$) = $b\ a$.
Comme l'égalité est transitive, on déduit $\left(\$F\ x \to x\ a\right)\ b = b\ a$.

Notation n°14 : variable(s): $x_1, \ldots, x_n, f$
Condition: $x_1, \ldots, x_n$ sont des "word", $n \geqslant 2$
" $\$F\ x_1, \ldots, x_n \to f$ " → " $\$F\ x_1 \to \$F\ x_2 \ldots, x_n \to f$ ".

Nous ajoutons une notation pour définir un nom à l'intérieur d'une formule :

Notation n°15 : variable(s): x, y, f
Condition: x est un "word".
" $\left[\textbf{let}|\ x\textbf{=}y\textbf{;}\ f\right]$ " → " $\left(\$F\ x \to f\right)\ y$ ".

Remarque : Nous avons défini des sortes de fonctions, mais il ne faudrait pas croire qu'il s'agit de la forme habituelle. Celle utilisée ici est purement syntaxique et la seule difficulté de compréhension est d'oublier les habitudes (domaine, nombre d'argument etc).

Remarque : Si #1 est défini comme une fonction à deux arguments, et que l'on veut définir g comme égal à f, on peut définir #1 = $\$F\ x, y \to$ #1 $x\ y$ mais aussi #1 = $\$F\ x \to$ #1 x ou encore #1 = #1. J'ai essayé de mettre le plus souvent tous les arguments pour rendre la lecture plus didactique, mais pour éviter des lourdeurs il arrive aussi que certains arguments ne soit pas indiqués.

10.3 Définition d'une fonction de placement récursive

Soit h, nous cherchons à définir f de telle sorte que l'on ait f = h f (dans la pratique h sera presque toujours défini par une fonction). Cherchons la valeur de f en supposant f = h f. On a alors f = h (h f) = h (h (h f)) ... = h (h (h (h ...)))). On voit donc que l'argument de h doit être une fonction qui a pour argument elle-même. Soit selfx = $F f,x -> x x. On a (selfx h)(selfx h) = h . (selfx h) (selfx h) = h . h . (selfx h) (selfx h) = h . h . h . (selfx h) (selfx h) ...

Cela correspond donc à ce que l'on cherche, et en posant f = (selfx h) (selfx h) on a f = h f.

10.3.1 Composition fonctionnelle

```
/*
Composition de deux fonctions f et g.
*/
$Def functionCompose = $F f, g, x → f (g x)
```

Notation n°0 : variable(s): x, y , associativité: left, priorité: 10
"x .o y" → "#1 $x\ y$".

10.3.2 Définition

```
/*
Applique l'argument à lui-même.
*/
$Def self = $F x → x x
```

```
/*
Double le premier argument d'une function.
```

*/
$**Def selfx** = $F\ f, x \rightarrow f\ (x\ x)$

Notation n°1 : variable(s): f est un "word", n est un entier naturel.;
" $FR\ f, x1, ..., xn \rightarrow def$ " → " #1 $\Big(\#1\ F\ f, x1, ..., xn \rightarrow def\Big)$ ".

Notation n°2 : variable(s) : func,def,comment ;
Condition : func est un word qui est un mot utilisé dans def

```
/*
comment
*/
$Def func = def
```

qui signifie :

```
/*
comment
*/
$Def func = $FR func → def
```

Remarque : Si on traduit $FR\ func \rightarrow def$ en formula le mot func est inutilisé car c'est la variable d'une notation de fonction. Ces notations font disparaître à leur traduction les mots qui sont des variables.

Chapitre 11

Les outils, niveau 1

11.1 Défintions des outils de niveau 1

11.2 Les listes et le lambda calcul

Cette partie n'a d'intérêt que pour les personnes ayant l'habitude du lambda calcul ou de ses langages dérivés comme Haskell. L'habitude est de construire les listes par une structure représentant l'élément de tête et le reste. J'ai suivi cette convention, jusqu'à arriver à des inversions de sens de plus en plus nombreux (du type "commencer par la fin"), pour finir par m'apercevoir que tout rentrait dans l'ordre si je construisais les listes par une structure représentant le début de la liste et l'élément de queue. Ces deux structures sont similaires, la différence est juste dans le vocabulaire et les notations, mais il est très soulageant de ne pas avoir à passer son temps à "commencer par la fin pour finir au début".

Voici un résumé des problèmes rencontrés dans la construction à la Haskell :

- Dans la pratique (par exemple écrire une liste de course) on construit une liste petit à petit en ajoutant des éléments en queue, le constructeur naturel est donc "ajouter un élément en queue", alors qu'en Haskell c'est l'inverse.
- Quand l'ordre des éléments de la liste compte (comme une preuve mathématique), comment construire une liste m dérivée (comme un map) d'une liste l et ayant le même ordre ? Seul foldr (qui commence par la fin de la liste) permet de construire m dans le même ordre. Mais si l'ordre des élément est important, pour que foldr puisse être utilisé il faut alors que le premier élément soit en fin de liste !

```
///
///                                   Notations de base
///
/* Notation pour les boolean */
$Nota 0 nonOp zero

$Nota 1 nonOp one

/*
"." est un pseudo-opérateur : il a le même rôle que l'application mais avec la plus basse priorité
et associatif à droite. Ainsi on peut alléger l'écriture en replaçant f (x y z) par f . x y z, ou
encore f ( g ( x y z ) ) par f . g . x y z
$_apply_ n'est pas une formule, juste un symbole utilisé dans cette notation pour représenter
l'application.
*/
$Nota . infix right 0 $_apply_

$Nota = infix left 10 equal

///
///                                   Fonctions de base
///

/*
Fonction `identité'
*/
$Def identityF = if 0 0

/*
Fonction constante (le premier argument est la constante), on aura constantF x y=x
*/
$Def constantF = if 1

/*
Composition de deux fonctions f et g.
*/
$Def functionCompose = $F f,g,x -> f . g x

$Nota .o infix left 10 functionCompose

/*
Compose la fonction sur elle-même.
*/
$Def autoCompose = $F f -> f .o f

/*
Modifie une fonction en appliquant l'argument à lui-même en premier.
*/
$Def selfx = $F f,x -> f . x x

$Def recurse = $F f -> selfx f . selfx f

///
///                                   Logique de base
///

/*
{"et" logique sur des booléens
*/
$Def boolAnd = $F x,y -> if x y 0

$Nota &b infix left 6 boolAnd


/*
"ou" logique sur des booléens
*/
$Def boolOr = $F x,y -> if x 1 y

$Nota |b infix left 6 boolOr

/*
```

```
"non" logique sur un booléen
*/
$Def boolNeg = $F x -> if x 0 1

$Nota !b prefix 6 boolNeg

/*
x est-il un booléen ?
*/
$Def isBoolean = $F x -> if x 1 1

// Remarque : Nous aurons une forme de tiers exclu qui nous permettra de déduire des vérités à
partir d'une vérité "if x y z".

// Remarque : "isBoolean equal" n'a aucunne raison d'être un booléen.

/*
Deux booléens sont-ils égaux.
*/
$Def boolEqual = $F x,y -> if x y !b y

// Remarque : "boolEqual" retourne un booléen si x et y le sont, alors que $0 = 0$  n'a aucunne
raison d'être un booléen.

/*
Deux booléens sont-ils inégaux.
*/
$Def boolNotEqual = $F x,y -> !b boolEqual x y

///
///                                          Appartenance
///

/*
Cette définition n'a pas de valeur mathématique mais seulement pédagogique. "in" ne sert qu'à
informer le lecteur que l'on s'attend à ce que "$\co{y}{x}$" retourne un booléen.
*/
$Def in = $F x,y -> y x

$Nota ∈ infix left 10 in

/*
Négation de l'appartenance.
*/
$Def notIn = $F x,y -> !b y x

$Nota ∉ infix left 10 notIn


///
///                                          Les couples
///

/*
La fonction "couple de x et y".
*/
$Def pair = $F x,y,c -> if c x y

/*
Retourne la premiere composante d'un couple.
*/
$Def pairFirst = $F x -> x 1

/*
Retourne la seconde composante d'un couple.
*/
$Def pairSecond = $F x -> x 0

// Remarque : Tout chose x peut-être vue comme une paire, ses deux éléments sont x 1 et x 0.

/*
La chose est-elle une paire . Tout est une paire donc retourne toujours vrai.
```

```
*/
$Def isPair = constantF 1

///
///                                        Maybe
///

// Nous allons définir une représentation de soit "rien" soit "la chose x". Nous nous servirons de
ces représentations pour définir les listes. "just x" est représenté par toute chose f telle que "f
1 = 1$" et "f 0=x$".  "nothing" est représenté par toute chose f telle que "f 1=0".

/*
représente "pas de chose"
*/
$Def nothing = constantF 0

$Nota 0m nonOp nothing

/*
Prend en argument x et signifie "la chose x"
*/
$Def just = $F x -> $P[ 1 , x ]

/*
x est-il quelque chose ?
*/
$Def isThing = $F x -> x 1

/*
x est-il nothing ?
*/
$Def isNothing = $F x -> !b isThing x

/*
x est un un "Just x" ou un "Nothing" ?
*/
$Def isMaybe = $F x -> isBoolean . x 1

/*
Récupère la chose d'un "just x"
*/
$Def getThing = $F jx -> jx 0

/*
Retourne x à partir de just x et y à partir de nothing.
*/
$Def maybeTo = $F y, m -> if ( isThing m ,  getThing m, y )

/*
Gère l'application d'une fonction à un maybe, retourne un maybe qui est 0m (nothing) si l'argument
est nothing, sinon retourne just de l'application de la fonction à l'argument du just.
*/
$Def applyJust = $F f,x -> if ( isThing x , just . f(getThing x) , 0m )

/*
Comme applyJust mais pour deux arguments.
*/
$Def apply2Just = $F f,x,y ->
  if ( isThing x &b isThing y , just . f(getThing x, getThing y) , 0m )

///
///                                        Right / Left
///

// Nous allons définir une représentation de ``une chose marquée left'' et de ``une chose marquée
right''. Pour cela nous utilisons une paire pair b x où x est la chose et b une valeur égal à 1 ou
0.

/*
Le LeftRight est-il un left.
*/
$Def isLeft = $F x -> pairFirst x
```

```
/*
Le LeftRight est-il un right.
*/
$Def isRight = $F x -> !b isLeft x

/*
Retourne la chose contenue dans le leftRight.
*/
$Def leftRightToThing = pairSecond

/*
Construction d'un left.
*/
$Def left = $F x -> $P[ 1 , x ]

/*
Construction d'un right.
*/
$Def right = $F x -> $P[ 0 , x ]

/*
L'argument est-il un leftRight ?
*/
$Def isLeftRight = $F x -> isPair x &b isBoolean ( pairFirst x )

///
///                                          Divers
///

/*
L'application.
*/
$Def apply = $F f,a -> f a

/*
Applique un argument à une fonction
*/
$Def applyArg = $F a,f -> f a

/*
Fonction doublement constante (ignore deux arguments).
*/
$Def constantF2 = $F c -> constantF . constantF c

///
///                                        Les listes
///

// Nous allons définir une représentation primitive des ensembles: cette construction permettra de
déduire une version simple et calculable de ``pour tout'' et ``il existe''.

///                                       Représentation


/*
Nous allons représenter une liste par une fonction `f' de la manière suivante:
  * nothing représente la liste vide
  * just . pair x l représente la liste de dernier élément `x' et commençant par la liste `l'.
*/

/*
Une liste vide.
*/
$Def emptyList = 0m

$Nota 0l nonOp emptyList

// Exemple : La liste constituée du seul élément 1 peut se représenter par: just . pair 1 0l
```

```
/*
Ajoute x en queue de la liste l.
*/
$Def addToList = $F x,l -> just $P[ x , l ]

/*
À la liste l, ajoute x en queue.
*/
$Def toListAdd = $F l,x -> addToList x l

$Nota :l infix left 10 toListAdd

/* Notation : variable :
 $L[] -> 0l
*/


/* Notation : variables : x1 , ..., xn
   condition : n>0
 $L[x1,...,xn] -> $L[x1,...,xn-1] :l xn
*/


// Remarque : $L[x2,...,x1]=$L[]

/*
Retourne la liste contenant l'unique élément x.
*/
$Def singletonList = $F x -> 0l :l x

///                                        Liste vide

/*
La liste `l' est-elle vide.
*/
$Def isEmptyList = isNothing

/*
La liste `l' est-elle non vide.
*/
$Def notEmptyList = $F l -> !b isEmptyList l


///                                        Tête et queue


/*
Dernier élément d'une liste.
*/
$Def listLast = pairFirst .o getThing

/*
Retourne nothing si la liste est vide, sinon just le premier élément.
*/
$Def listLastMaybe = $F l -> if ( isEmptyList l , 0m , just . listLast l )

/*
La liste privée de son dernier élément.
*/
$Def listInit = pairSecond .o getThing

/*
Retourne si la liste est un singleton.
*/
$Def isSingletonList = $F l -> notEmptyList l &b isEmptyList ( listInit l )

/*
Retourne si la liste comporte 2 éléments.
*/
$Def is2ElementsList = $F l -> notEmptyList l &b isSingletonList ( listInit l )
```

```
///                                Les fonctions de parcour

// Le texte explicatif se trouve à la fin de cette partie de code source, sous le titre "Étude sur
les fonctions de parcours des listes".

/*
Fonction de calcul récursive sur une liste. Retourne la valeur d'accumulation finale (voir ci-
dessous et l'étude ci-dessus). Ses arguments:
- f: la fonction de calcul de la valeur d'accumulation. Ses arguments sont:
    - h: l'élément de la liste
    - r: la valeur d'accumulation
- r: la valeur d'accumulation de départ
- l: la liste à parcourir.
- it : la fonction d'itération, elle détermine l'ordre des calculs. Ses arguments sont:
    - lff: la fonction de parcour déjà appliquée à f, il reste donc à l'appliquer à r et lt.
    - fh: f appliquée à l'élément de tête de la liste
    - r: la valeur d'accumulation
    - lt: la queue de la liste
L'argument lf n'est pas à fournir puisqu'il est utilisé par la récurrence pour représenter la
fonction récursive (on peut le voir comme par exemple "listRevFold".
*/
$Def listFoldByIter = $F it ->
  recurse $F lf,f,r,l ->
    if ( isEmptyList l , r , it ( lf f , f . listLast l , r , listInit l ) )

/*
Itérateur de parcours où l'on commence par le fin de la liste.
*/
$Def listRevIterator = $F lff,fh,r,lt -> lff ( fh r ) lt

/*
Itérateur de parcours où l'on commence par la tête de la liste.
*/
$Def listIterator = $F lff,fh,r,lt -> fh . lff r lt

/*
Parcour une liste en partant de la queue tout en calculant récursivement un résultat r. Le résultat
en cours est calculé par une fonction f sur le résultat précédent et l'élément de la liste. f, r et
l sont éludés.
*/
$Def listRevFold = listFoldByIter listRevIterator

/*
Pareil que listRevFold mais en partant de la tête
*/
$Def listFold = listFoldByIter listIterator


/*
En reprenant notre exemple de liste de course, nous allons maintenant imaginer qu'en plus de
calculer le nombre d'articles alimentaires nous gérons aussi les erreurs, c'est à dire que si un
nom d'article est illisible nous arrêtons les calculs en déclarants que la liste n'est pas
conforme. La fonction que nous assurons produit donc soit un état "erreur", soit l'information que
le calcul s'est effectué avec le résultat. Nous allons pour cela utiliser $\defi{just}$ et
$\defi{nothing}$ pour l'accumulateur et une fonction d'itération s'arrêtant éventuellement si un
$\defi{nothing}$ (une erreur) est rencontrée. La fonction de calcul f devra retourner un
$\cno{just}{r}$ ou un $\defi{nothing}$ et prendra en argument un accumulateur "maybe".
*/

/*
Itérateur de parcours où l'on commence par le début de la liste, tout en gérant les erreurs. r doit
être un maybe, c'est le cas au départ et par la suite car f retourne un maybe.
*/
$Def listRevMaybeIterator = $F lff,fh,r,lt ->
  if ( isNothing r , 0m , lff ( fh . getThing r ) lt )

/*
Itérateur de parcours où l'on commence par le début de la liste. r n'est pas un maybe, mais f
retourne un maybe, donc r2 est un maybe.
*/
```

```
$Def listMaybeIterator = $F lff,fh,r,lt ->
  $Let r2 = lff r lt ,
  if ( isNothing r2 , 0m , fh . getThing r2 )

/*
Comme listRevFold mais avec une gestion des erreurs, voir ci-dessus.
*/
$Def listRevFoldMaybe = $F f,r -> listFoldByIter listRevMaybeIterator f . just r

/*
Comme listFold mais avec une gestion des erreurs.
*/
$Def listFoldMaybe = $F f,r ->  listFoldByIter listMaybeIterator f . just r

/*
Comme listRevIterator mais un critère p arrête le parcour s'il est vrai sur r.
*/
$Def listRevStopIterator = $F p,lff,fh,r,lt -> if ( p r , r , lff ( fh r ) lt )

/*
Comme listRevFold mais un critère p arrête le parcour s'il est vrai sur r.
*/
$Def listRevFoldStop = $F p -> listFoldByIter . listRevStopIterator p

/*
Retourne une liste : le début devient la fin
*/
$Def listReverse = $F l -> listRevFold addToList 0l l

/*
La liste constituée de l'application de `f' sur chaque élément de `l'.
*/
$Def listMap = $F f,l -> listFold ( $F x,r -> r :l f x ) 0l l

/*
l1 et l2 sont deux listes. Si la longueur de l1 est plus grande que l2 alors réduit la première
liste à la taille de la seconde, sinon ne fait rien.
*/
$Def listReduceFirst = $F l1,l2 ->
  // Réduit l1 d'autant qu'il reste d'élément dans resteL1
  listFold ($F x,l -> listInit l) l1 .
  // Réduit l1 de la longueur de l2, s'arrête s'il ne reste plus rien à réduire.
  listRevFoldStop ($F resteL1 -> isEmptyList resteL1) ( $F x,resteL1 -> listInit resteL1 ) l1 l2


/*
Comme listMap mais la fonction f prend deux arguments provenant de deux listes l et m devant être
de la même longueur.
*/
$Def list2Map = $F f,l,m ->
  if
  (
    isEmptyList l
    ,
    0l
    ,
    list2Map ( f , listInit l , listInit m ) :l f ( listLast l , listLast m )
  )

/*
Assemble deux listes en une liste de paires. Les deux listes doivent être de la même longueur.
*/
$Def list2Pair = list2Map pair


/*
Retourne l2 réduite (par la fin) d'autant que la longueur de l1. Retourne just du résultat ou
nothing si l1 est strictement plus longue que l2.
*/
$Def reduceFromListMaybe = $F l1,l2 ->
  listRevFoldMaybe
  ( $F x,r -> if ( isEmptyList r , 0m , just . listInit r ) , l2 , l1 )
```

```
/*
l1 et l2 sont-elles de la même longueur. En premier on réduit l1 d'autant que la longueur de l2.
*/
$Def listEqualLength = $F l1,l2 ->
  $Let res = reduceFromListMaybe l1 l2
  ,
  isThing res &b isEmptyList ( getThing res )

/*
l1 a-t-elle une longueur égale ou plus petite que l2 ?
*/
$Def listShorterOrEqual = $F l1,l2 -> isThing . reduceFromListMaybe l1 l2

/*
La fonction suivante map une liste mais en même temps calcul un paramètre r récursivement qui sert
à mapper la liste. On aura:
 listFoldMapF f r $L[a,b,c] =
  $L[ pairFirst . f a r, pairFirst . f b (pairSecond . f a r), pairFirst . f c (pairSecond . f b
(pairSecond . f a r)) ].
f a les même argument que pour listFold, elle retourne la paire (valeur du maping,accumulateur)
*/
$Def listFoldMapF = $F f,r,l ->
  pairFirst . listFold (
    $F x, $P[r1, r2] ->
    $Let fxr = f x r2,
    $P[ r1 :l pairFirst fxr , pairSecond fxr ]
  )
  $P[0l,r] l

/*
listFoldMapFG est un cas simplifié de listFoldMapF quand le calcul de la valeur et de
l'accumulateur sont séparés. On peut donc fournir deux fonctions : f pour la valeur et g pour
l'accumulateur.
La fonction suivante map une liste (en utilisant f) mais en même temps calcul un paramètre r
récursivement qui sert à mapper la liste (en utilisant g). On aura:
 f g r $L[a,b,c] = $L[ f a r, f b (g a r), f c (g b (g a r)) ].
*/
$Def listFoldMapFG = $F f,g -> listFoldMapF $F x,r -> $P[ f x r, g x r ]



///                                          Concaténation

/*
Retourne la liste `x' à laquelle on a ajouté `y' en queue.
*/
$Def concatList = $F x,y -> listFold addToList x y

/*
Retourne la liste `y' à laquelle on a ajouté `x' en queue.
*/
$Def concatToList = $F x,y -> concatList y x

$Nota +l infix left 10 concatList

/*
La concaténation de tous les éléments (des listes) d'une liste.
*/
$Def concatListList = $F l -> listFold concatToList 0l l

/*
Combinaison de concatListList appliqué après un listMap.
*/
$Def concatListMap = $F f,l -> concatListList . listMap f l

///                                      "Pour tout" et "il existe"

/*
Nous allons définir un ``pour tout'', ``il existe'' et ``appartient'' sur les listes.

Définition informelle de "une propriété p" :
```

```
Nous utilisons ici l'expression ``une propriété p'' pour indiquer que les cas qui nous intéressent
sont ceux où p retourne un booléen sur les arguments qui nous intéressent.

Remarque : Nous pourrons définir plus tard des notions de domaine de fonction etc. Mais Lazi ne
donne aucunne garantie sur les choses existantes (on n'a pas une vérité universelle forte de la
forme "(x ∈ y) |b (x ∉ y)"), mais en général avoir peu d'apprioris sur la réalité est une force.
*/


/*
Un "pour tout" sur une liste prend en argument une liste `l' et une propriété `p' et exprime que
tout élément de l est vérifié par `p'. La condition d'arrêt permet en général de diminuer le temps
de calcul.
*/
$Def forAllOnList = $F l,p ->
  listRevFoldStop boolNeg ( $F e,r -> p e ) 1 l

/*
Notation : variables : p,x,m
Conditions : x est un word
∀l x/m; p -> forAllOnList m $F x -> p

Remarque : un "L" minuscul est accolé au quantificateur.
*/

/*
Applique "and" sur tous les éléments de la liste.
*/
$Def listAnd = $F l -> ∀l x/l; x

/*
Un "il existe" sur une liste prend en argument une liste `l' et une propriété `p' et exprime qu'un
élément de l est vérifié par `p'. La condition d'arrêt permet en général de diminuer le temps de
calcul.
*/
$Def existsOnList = $F l,p ->
  listRevFoldStop identityF ( $F e,r -> p e ) 0 l

/*
Notation : variables : p,x,m
Conditions : x est un word
∃l x/m; p -> existsOnList m $F x -> p
*/

/*
Applique "or" sur tous les éléments de la liste.
*/
$Def listOr = $F l -> ∃l x/l; x

///                                          Égalité

/*
Retourne si les listes `l1' et `l2' sont égales suivant la fonction de comparaison `eq'.
*/
$Def listEqual = $F eq,l1,l2 -> listEqualLength l1 l2 &b listAnd ( list2Map eq l1 l2 )


///                                 Appartenance à une liste

/*
Pour une comparaison `eq', la liste `l' contient-elle un élément comparable à x.
*/
$Def inList = $F eq,l,x -> existsOnList l ( eq x )

///                                          Unicité

/*
La liste ne contient pas de doublon ?
*/
$Def noDoubleInList = $F eq,l -> isEmptyList l |b
  (
    isThing . listRevFoldMaybe
```

```
      (
        $F x,l -> if ( inList eq l x , 0m , just . listInit l ) ,
        listInit l ,
        l
      )
  )

///                                    listFilter & co

/*
D'une liste de Maybe , retourne les choses dans les just, dans le même ordre.
*/
$Def listGetThings = $F l ->
  listFold ($F m,result -> if( isThing m, result :l getThing m, result) ) 0l l

/*
Applique un maping sur une liste puis un listGetThings
*/
$Def getThingsListMap = $F f,l -> listGetThings . listMap f l

/*
Retourne just le dernier élément  nothing si la liste est vide.
*/
$Def getMaybeOneThingList = $F l ->
  if( isEmptyList l, 0m, just . listLast l )

/*
Applique un maping sur une liste puis retourne just le dernier élément étant un just ou nothing si
la liste ne contient que des 0m.
*/
$Def getMaybeOneThingListMap = $F f,l -> getMaybeOneThingList . getThingsListMap f l


/*
Sélectionner les éléments de la liste `l' vérifiant la propriété p.
*/
$Def listFilter = $F p,l -> listFold ($F x,res-> if ( p x , res :l x , res )) 0l l

/*
Comme listFilter mais en plus map les éléments sélectionnés.
*/
$Def listFilterMap = $F p,f,l -> listFold ($F x,res-> if ( p x , res :l f x , res )) 0l l

///                                    listRemoveList


/*
La liste l1 moins les éléments se trouvant dans l2 suivant l'égalité eq.
*/
$Def listRemoveList = $F eq,l1,l2 -> listFilter ( $F x -> !b inList eq l2 x ) l1

///                Fonction définie par une liste (fonction par liste)

// Une liste de paires (argument,image) peut servir pour définir une fonction.


/*
Retourne la liste des images de x.
*/
$Def listFuncToImages = $F eq,l,x -> listFilterMap ( $F e -> eq ( pairFirst e ) x ) pairSecond l

/*
Utilise une liste l comme une fonction définie en extension.
*/
$Def listFuncToFunc = $F eq,l,x -> listLast . listFuncToImages eq l x

/*
Utilise une liste l comme une fonction définie en extension et retournant 0m si l'argument est hors
domaine et sinon "just image".
*/
$Def listFuncToFuncMaybe = $F eq,l,x ->
  $Let r = listFuncToImages eq l x , if ( isEmptyList r , 0m , just . listLast r )
```

```
/*
Domaine d'une fonction par liste, retourné sous forme d'une liste.
*/
$Def listFuncDomain = $F l -> listMap pairFirst l

/*
Teste si une chose est dans le domaine.
*/
$Def listFuncInDomain = $F eq,l,x -> ∃l a/listFuncDomain l; eq x a

/*
Codomaine d'une fonction par liste.
*/
$Def listFuncCodomain = $F l -> listMap pairSecond l

/*
Teste si une chose est dans le codomaine.
*/
$Def listFuncInCodomain = $F eq,l,x -> ∃l a/listFuncCodomain l; eq x a

/*
Réduit une fonction par un filtre p sur son domaine.
*/
$Def listFuncReduce = $F l,p -> listFilter ( $F x -> !b p ( pairFirst x ) ) l


/*
Ajoute le couple (a,i) à la liste par fonction.
*/
$Def listFuncAdd = $F f,a,i -> f :l $P[ a , i ]

/*
Renvoie une fonction similaire à f mais où a est envoyé sur i.
*/
$Def listFuncSet = $F eq,f,a,i -> listFuncAdd ( listFuncReduce f ( eq a ) ) a i

/*
Unions des fonctions f et g, si x est dans le domaine de f et g, c'est la valeur de g qui est
utilisée.
*/
$Def listFuncMerge = $F eq,f,g -> listRevFold ( $F p@$P[x,y], g -> if (listFuncInDomain eq g x, g,
g :l p) ) g f

/*
Renvoie une fonction similaire à f mais où la valeur x associée à a est remplacée par g x
*/
$Def listFuncMod = $F eq,f,a,g ->
  listMap
    ( $F p@$P[pa,px] -> if( eq pa a , $P[ pa , g px ] , p ) )
    f

/*
Comme map sur les listes, mais là ce sont les images de la fonction qui sont mappées.
*/
$Def listFuncMap = $F f,l -> listMap ( $F $P[x,y] -> $P[ x , f y ] ) l

/*
Définition informelle : super fonction :
Une super fonction est une fonction où les images sont considérées comme des fonctions.
*/

///                                          Validité

/*
`l' est-elle une liste.
*/
$Def isList = $F l -> isMaybe l &b ( isEmptyList l |b isList ( listInit l ) )

// Remarque : Nous serons assurés que si l est une liste finie alors isList l =1 et que si isList l
=1 alors l est une liste finie. Mais si l n'est pas une liste isList l ne vaut pas 0.
```

```
///                                  Liste de booléens

/*
l est-elle une liste binaire.
*/
$Def isBoolList = $F l -> isList l &b ∀l x/l; isBoolean x

///                                     listFirst

/*
Pour une liste non vide, le premier élément de la liste.
*/
$Def listFirst = $F l -> listRevFold ($F x,r -> x) nothing l


///
///                                  Les formules
///

///                                Définition des mots

// Nous utiliserons des listes de booléens pour représenter les mots.

/*
Être un mot.
*/
$Def isWord = isBoolList

/*
Égalité entre deux mots.
*/
$Def wordEqual = listEqual boolEqual

$Nota =w infix left 10 wordEqual

/*
Inégalité entre deux mots.
*/
$Def wordNotEqual = $F x,y -> !b x =w y

///                        Notation pour la représentation des mots

/*
Nous pouvons utiliser n'importe quelle liste de 0 et 1 pour représenter un mot. Un système simple
est d'avoir une correspondance avec les mots de notre langage.

Nous utiliserons le codage informatique ``UTF-8'' qui permet de traduire chaque caractère en un
nombre binaire entre 8 et 32 bits. Nous traduirons un mot par la concaténation des listes
représentant les caractères de ce mot, le premier caractère étant en tête de liste.

Notation : l
condition : l est un mot
`l' -> $L[ LB ]

où LB est la liste booléenne représentant le mot en UTF-8. Par exemple `AB' se traduit en
$L [ 0,1,0,0,0,0,0,1,0,1,0,0,0,0,1,0 ] car en UTF-8 'A' a le code 65 et 'B' 66.
*/

///                               Représentation des formules

/*
Maintenant que nous avons défini une représentation pour les mots, il ne nous reste plus qu'à
représenter l'application d'une fonction sur un argument pour pouvoir représenter n'importe quelle
formule Lazi.
Une formule d'un seul mot sera représentée par left x.
Une application de x sur y sera représentée par right . pair x y
*/

/*
La formule est-elle un mot.
*/
$Def formulaIsWord = $F f -> isLeft f
```

```
/*
Fonction retournant la formule composé du mot en argument.
*/

$Def wordToFormula = $F w -> left w

/*
Quand la formule est composée d'un seul mot, retourne ce mot.
*/
$Def formulaToWord = leftRightToThing

/*
La formule est-elle une application.
*/
$Def formulaIsApply = $F f -> isRight f

/*
Applique (assemblage) une formule à un autre.
*/
$Def formulaApply = $F x,y -> right $P[ x , y ]

$Nota .f infix left 10 formulaApply

// Nous allons définir des fonctions pour exprimer ``la fonction dans une application de formules''
et ``l'argument dans une application de formule''.

/*
Retourne la partie gauche (fonction) d'une formule composée.
*/
$Def formulaFunc = $F x -> pairFirst . leftRightToThing x

/*
Retourne la partie droite (l'argument) d'une formule composée.
*/
$Def formulaArg = $F x -> pairSecond . leftRightToThing x

/*
La chose est-elle une formule.
*/
$Def isFormula = $F t ->
  isLeftRight t &b
  if
  (
    formulaIsWord t
    ,
    isWord . formulaToWord t
    ,
    isFormula ( formulaFunc t ) &b isFormula ( formulaArg t )
  )

// Remarque : Comme les autres fonctions booléennes, nous pourrons faire des déductions sur
"isFormula x" quand x est une formule, mais dans le cas contraire isFormula x ne sera pas un
boolée, nous ne pourrons rien en déduire mais cela ne nous généra en rien.


/*
Deux formules sont-elles égales.
*/
$Def formulaEqual = $F x,y ->
  (
    formulaIsWord x &b
    formulaIsWord y &b
    formulaToWord x =w formulaToWord y
  )
  |b
  (
    formulaIsApply x &b
    formulaIsApply y &b
    formulaEqual ( formulaFunc x , formulaFunc y ) &b
    formulaEqual ( formulaArg x , formulaArg y )
  )
```

```
$Nota =f infix left 10 formulaEqual

/*
Négation de formulaEqual.
*/
$Def formulaNotEqual = $F x,y -> !b x =f y
```

11.3 Étude sur les fonctions de parcours des listes

Nous allons utiliser des fonctions récursives pour parcourir les listes.

Si nous voulons compter le nombre d'articles de nourriture sur une liste de course nous parcourons la liste élément par élément et pour chaque article de nourriture nous ajoutons 1 à un compteur. C'est le principe de base des parcours de liste ici : Un "compteur" r, une liste l, une fonction f s'occupant de la mise à jour du compteur.

Nous allons définir la fonctions de parcours de liste listFoldLeft et d'autres apparentées.
Soit une liste l de 3 éléments a b c, f une fonction et r une valeur de départ. On aura
#1 $f\ r\ l = f\ c\ \Big(f\ b\ \big(f\ a\ r\big)\Big)$ et #1 $f\ r\ l = f\ a\ \Big(f\ b\ \big(f\ c\ r\big)\Big)$.
On peut donc définir listFoldLeft et listFoldRight ainsi :

Définition récursive n°1 : *Calcul sur une liste avec un accumulateur, on commence par le début de la liste.*
#1 = $\$F\ f, r, l \rightarrow \mathbf{if}\Big(\#1\ l\ \mathbf{,}\ r\ \mathbf{,}\ listFoldLeft\ f\ \big(f\ (\#1\ l)\ r\big)\ (\#1\ l)\Big)$

Définition récursive n°2 : *Calcul sur une liste avec un accumulateur, on commence par la fin de la liste.*
#1 = $\$F\ f, r, l \rightarrow \mathbf{if}\Big(\#1\ l\ \mathbf{,}\ r\ \mathbf{,}\ f\ (\#1\ l)\ \big(listFoldRight\ f\ r\ (\#1\ l)\big)\Big)$

Nous aurons à définir d'autres fonctions de parcours. Nous voyons que ces fonctions ont des points communs. De plus, comme elles sont récursives, les preuves sur ces fonctions devrons utiliser la récurrence. Il parait donc utile de factoriser ces fonctions, c'est à dire de les définir à partir d'une fonction regroupant les parties communes. Elles sont :

- $if\ (\#1\ l)\ r$
- $f\ (\#1\ l)$
- $\#1\ l$
- r
- $\#1\ f$

Nous rassemblerons les parties différentes dans une fonction *i* dite d'itération car elle détermine la manière dont la liste est parcourue.

On obtient : #1 =
$\$FR\ listFoldLeft, f, r, l \rightarrow \mathbf{if}\Big(\#1\ l\ \mathbf{,}\ r\ \mathbf{,}\ \#1\ \big(listFoldLeft\ f\big)\ \big(f\ (\#1\ l)\big)\ r\ (\#1\ l)\Big)$

où #1 = $\$F\ lff, fh, r, lt \rightarrow lff\ \big(fh\ r\big)\ lt$

Donc en définissant
#1 = $\$F\ i \rightarrow \$FR\ lf, f, r, l \rightarrow \mathbf{if}\Big(\#1\ l\ \mathbf{,}\ r\ \mathbf{,}\ i\ \big(lf\ f\big)\ \big(f\ (\#1\ l)\big)\ r\ (\#1\ l)\Big)$

On a #1 #1 = #1

Chapitre 12

Les outils, niveau 2

Nous allons définir quelques outils plus sophistiqués, principalement en combinant les outils de niveau 1.

```
///
///                              Liste en tant qu'ensemble
///

/*
Tous les éléments de la liste x sont des éléments de la liste y.
*/
$Def listInclude = $F eq,x,y -> ∀l a/x; ∃l b/y; eq a b

/*
Les deux listes ont les mêmes éléments, les doublons sont possibles.
*/
$Def listSetEqual = $F eq,x,y -> listInclude eq x y &b listInclude eq y x

/*
Intersection de deux listes.
*/
$Def listIntersect = $F eq,l,m -> listFilter ( $F x -> ∃l y/m; eq x y ) l

/*
Intersection d'une liste de listes, la liste ne doit pas être vide.
*/
$Def listsIntersect = $F eq,ll -> listRevFold (listIntersect eq , listLast ll , listInit ll)

/*
Union de deux listes. S'il n'y a pas de doublon dans la première liste alors le résultat est sans
doublon.
*/
$Def listUnion = $F eq,l,m -> listRevFold ( $F xm,j -> if ( inList eq j xm , j , j :l xm ) ) l m

/*
Union d'une liste de listes.
*/
$Def listsUnion = $F eq,ll -> listRevFold (listUnion eq) 0l ll

/*
Différence de deux listes. On retourne l moins les éléments de m.
*/
$Def listDifference = $F eq,l,m -> listFold ( $F x,r -> if ( ∃l y/m; eq x y , r , r :l x ) ) 0l l


/*
La liste est-elle sans doublon.
*/
$Def listNoDup = $F eq,l ->
  isThing .
  listRevFoldMaybe
  (
    $F x,r -> if ( inList eq r x , 0m , just . listInit r )
    ,
    listInit l
    ,
    l
  )

/*
Enlève les doublons, ne conserve pas l'ordre.
*/
$Def listRemoveDup = $F eq,l ->
  listRevFold
  (
    $F x,r -> if ( inList eq r x , r , r :l x )
    ,
    0l
    ,
    l
  )

// Remarque : Quand il n'y a pas de doublon listRevFoldMaybe retourne $\cno{just}{\cnob{listInit}
{\emptyList}}$, mais on ne s'intéresse qu'à tester si le résultat est un just ou un nothing.
```

```
///
///                                  Liste de mots
///

/*
Être égal à un des éléments d'une liste de mots.
*/
$Def inWordList = $F x,l -> inList wordEqual l x

$Nota Ew infix left 10 inWordList

/*
La liste l contient-elle un mot. Nous définissons une fonction avec les arguments inversés car
l'ordre humainement normal des arguments n'est pas l'ordre normal du point de vue de l'utilisation
en tant que prédicat (les arguments les plus fixes doivent être en premier).
*/
$Def listHasWord = $F l,x -> x Ew l

/*
Contraire de `inWordList'.
*/
$Def notInWordList = $F x,l -> !b x Ew l

$Nota ∉w infix left 10 notInWordList

/*
Égalité entre deux listes de mots.
*/
$Def wordListEqual = listSetEqual wordEqual

/*
La liste de mots l moins le mot n.
*/
$Def wordListRemove = $F n,l -> listFilter ( wordNotEqual n ) l

/*
Intersection de deux listes de mots.
*/
$Def wordListIntersect = listIntersect wordEqual


///
///                                Liste de formules
///


/*
La liste de formule l moins la formule t.
*/
$Def formulaListRemove = $F t,l -> listFilter ( formulaNotEqual t ) l

/*
"listRemoveList" pour les formules.
*/
$Def formulaListRemoveList = listRemoveList formulaEqual

/*
Retourne si deux listes de formules sont égales.
*/
$Def formulaListEqual = $F l,m -> listEqual formulaEqual l m

/*
Retourne si une formule est dans une liste de formules.
*/
$Def inFormulaList = $F x,l -> inList formulaEqual l x

$Nota Ef infix left 10 inWordList


///
///                                  Dictionnaires
///
```

```
///                                  Définition de base

/*
Définition informelle : Dictionnaire :
Un dictionnaire est une fonction par liste dont le domaine est une liste de noms.
*/

/*
Pour un dictionnaire donné, retourne la valeur associée au mot.
*/
$Def dictApply = $F d,w -> listFuncToFunc wordEqual d w

$Nota .di infix left 10 dictApply


/*
Pour un mot donné, retourne la valeur correspondante sur un dictionnaire.
*/
$Def wordProj = $F w,d -> d .di w

/*
Pour un dictionnaire donné, retourne $\defi{nothing}$ si w est hors domaine, si non retourne just
la valeur associée au mot.
*/
$Def dictApplyMaybe = $F d,w -> listFuncToFuncMaybe wordEqual d w

/*
Pour un dictionnaire d dont l'entrée w retourne une fonction f, retourne f d. Cela servira quand
une fonction du dictionnaire a besoin d'accéder à d'autres données dans le dictionnaire.
*/
$Def dictApplyFunc = $F d,w -> d .di w . d

$Nota .df infix left 10 dictApplyFunc

/*
Pour un mot donné, retourne la valeur correspondante à l'application fonctionnelle (.df) sur ce
mot.
*/
$Def wordFuncProj = $F w,d -> d .df w


/*
À partir d'un dictionnaire dict, d'un mot w et d'une image i, retourne dict où les éventuelles
entrées pour w sont supprimé et où l'entrée w retourne i.
*/
$Def dictSet = $F dict,w,i -> listFuncSet wordEqual dict w i

/*
Union de deux dictionnaires, le second a la priorité en cas de conflit (quand les deux on une même
entrée).
*/
$Def dictMerge = $F d1,d2 -> listFuncMerge wordEqual d1 d2

$Nota ∪di infix left 10 dictMerge

/*
Modifie la valeur x d'une entrée w d'un dictionnare d, la nouvelle valeur est f x.
*/
$Def dictMod = $F d,w,f -> listFuncMod wordEqual d w f

/*
Réduit un dictionnaire en supprimant les entrées fournies par la liste de mots l
*/
$Def dictReduce = $F d,l -> listFuncReduce d ($F w -> inWordList w l)

/*
À partir d'un dictionnaire dict, d'un mot w et d'une image i, retourne dict où le couple (i,w) est
ajouté (sans supprimer l'éventuelle ancien image de w).
*/
$Def dictAdd = listFuncAdd
```

```
/*
Le mot w est-il une entrée du dictionnaire.
*/
$Def isDictEntry = $F w,d -> listFuncInDomain wordEqual d w

$Nota ∈di infix left 10 isDictEntry

/*
Le mot w n'est-il pas une entrée du dictionnaire.
*/
$Def notDictEntry = $F w,d -> !b isDictEntry w d

$Nota ∉di infix left 10 notDictEntry

/*
Le domaine du dictionnaire.
*/
$Def dictEntries = listFuncDomain

/*
Les valeurs du dictionnaire.
*/
$Def dictValues = listFuncCodomain

/*
La chose est-elle un dictionnaire.
*/
$Def isDict = $F d ->
  isList d &b
  listAnd ( listMap ( $F x -> isPair x &b isWord ( pairFirst x ) ) d ) &b
  noDoubleInList wordEqual . dictEntries d

///      Notation pour définir des fonctions ayant pour variables des dictionnaires
/*
Nous allons étendre la notation de définition de fonction pour faciliter l'utilisation des
dictionnaires en variables.

Nous introduisons une notation pour définir une fonction prenant en argument un dictionnaire pour
passer les arguments.

Définition informelle de notation: Un vecteur de noms nommé est une syntaxe de la forme
"x=$V[x1,...,xn]" où x est un word et x1,...,xn sont des word ou vecteur de nom nommés.

Définition informelle de notation: Un vecteur de noms anonyme est une syntaxe de la forme
"$V[x1,...,xn] où x1,...,xn  sont des word ou vecteur de noms nommés.

Définition informelle de notation:  Un vecteur de noms est un vecteur de noms nommé ou anonyme.

Notation :
  Variables : x1,f
  Condition : x1 est un word
   "$F $V[x1] -> f"  ->   "( $F g,v -> g . v .di `x1' ) $F x1 -> f "

Remarque : Si on avait utiliser la définition suivante (où on applique la fonction pour remplacer
g),  "$F $V[x1] -> f"  ->   " $F v -> ( $F x1 -> f , v .di `x1' ) "
on aurait alors introduit la variable "v" dans f, et donc le sens de f serait changé s'il utilisait
le mot v.

Notation :
  Variables : x1 ... xn,f
  Conditions : n>1, x1 est un word, x2...xn sont des word ou vecteurs de noms nommés.
  "$F $V[x1,...,xn] -> f" -> "( $F g,v -> g (v .di `x1') v ) $F x1,$V[x2,...,xn] -> f "


La notation précédente permet d'accéder simplement aux valeurs du dictionnaire dans une fonction,
mais la fonction peut avoir besoin du dictionnaire entier. Pour cela nous ajoutons la notation
suivante qui permet d'utiliser dans le corps de la fonction (f) le dictionnaire entier (x) ainsi
que la valeur de ses entrées par la notation précédante:

Notation :
  Variables : x,v,f
  Conditions : x est un word, v est un vecteur de noms anonyme.
```

```
  "$F x=v -> f" -> "$F x -> ($F v -> f) x"

Nous ajoutons une notation pour mixer les types de variables, cette notation étend celle réduite à
la condition où les x_i$ sont des noms.

Notation :
  Variables : x1 ... xn,f
  Conditions : x1...xn sont des word ou vecteurs de noms, n>1
  " $F x1,...,xn -> f " -> "$F x1 -> $F x2,...,xn -> f"

La notation suivante permet d'imbriquer les dictonnaires.

Notation :
  Variables : x1,y1 ... xn,f
  Conditions : n>1, x1 est un word, y1 est un vecteur de nom anonyme, x2 ... xn sont des word ou
vecteurs de noms nommés.
  "$F[x1=y1,x2...xn] -> f" -> "( $F g,v -> g (v .di `x1') v ) $F x1=y1,$V[x2,...,xn] -> f "

Nous étendons la notation pour définir un vecteur de noms en tant que variable.

Notation :
  Variables : x,y,f
  Conditions : x est un vecteur de noms.
  "$Let x=y, f" -> "$F x -> f"
 */

///                                    Fonctions avancées


/*
map les valeurs d'un dictionnaire. L'argument de la fonction et du dictionnaire sont éludés.}
listFuncMap.
*/
$Def dictMap = listFuncMap

/*
Retourne un dictionnaire avec les mêmes entrées que d1 et où les valeurs sont f appliquée à deux
arguments: les valeurs dans d1 et d2 correspondantes à l'entrée.
*/
$Def dict2Map = $F f,d1,d2 ->
  listMap
  ( $F p -> $P[ pairFirst p , f ( pairSecond p , d2 .di pairFirst p ) ] )
  d1

///                                    Sur les formules

/*
Parcourt une formule e en appliquant fw sur les mots et fa sur le résultat de la récursion sur les
parties gauche et droite de la formule (la fonction et l'argument).
*/
$Def foldFormula = $F fw,fa,e ->
  if
  (
    formulaIsWord e
  ,
    fw . formulaToWord e
  ,
    fa ( foldFormula fw fa ( formulaFunc e ) , foldFormula fw fa ( formulaArg e ) )
  )

/*
Remplace un ensemble de mots par leur valeur associé (par le dictionnaire d) dans une formule.
*/
$Def replaceWords = $F d,f ->
  foldFormula
  ( $F e -> maybeTo ( wordToFormula e , dictApplyMaybe d e ) )
  formulaApply
  f

/*
Traduit une formule f à partir d'un dictionnaire d donnant le sens des mots.
*/
```

```
$Def translateFormula = $F d,f -> foldFormula ( listFuncToFunc d ) apply f

/*
Tous les mots de f sont-ils dans la liste de mots l.
*/
$Def formulaUseWords = $F l,f -> foldFormula ( inList wordEqual l ) boolAnd f

/*
Les mots aparaissant dans la formule f, sans doublon.
*/
$Def usedWords = $F f -> foldFormula singletonList (listUnion wordEqual) f

/*
Tous les mots de f sont-ils des entrées de d.
*/
$Def canTranslateFormula = $F d,f -> formulaUseWords f . dictEntries d

/* Fait correspondre une formule model avec variables à une autre formule obj pour assigner les
variables. Retourne just le dictionnaire des valeurs des variables (lv est la liste des variables).
model est la formule à variables, obj la formule à faire correspondre (contenant les valeurs). En
cas d'erreur retourne 0m (nothing). ass est le dictionnaire des assignations déjà trouvées, si une
variable déjà trouvée réapparait avec une valeur différente alors la fonction retourne nothing.
*/
$Def matchVars = $F lv,ass,model,obj ->
  if ( formulaIsWord model // Si le modèle est un mot
  ,
    $Let w = formulaToWord model,
    if ( w ∈w lv  // Si model est une variable
    ,
      if ( isDictEntry w ass // Si la variable est déjà trouvée.
      ,
        if ( ass .di w =f obj // Si la valeur correspond
        ,
          just . ass // On retourne just le dictionnaire
        ,
          nothing // Sinon il y a échec.
        )
      ,
        just . dictAdd ass w obj // On retourne just les assignations avec la nouvelle
      )
    ,
      if ( model =f obj , just ass , 0m ) // Comparaison sans variable
    )
  ,
    if ( formulaIsWord obj   // si obj et un word et non model
    ,
      0m // matching impossible
    ,
      $Let mass2 = matchVars lv ass ( formulaFunc model , formulaFunc obj ),
      if ( isThing mass2
      ,
        // Prend les assignations trouvées dans la  partie fonction et continue dans la partie
argument.
        matchVars lv (getThing mass2 , formulaArg model , formulaArg obj )
      ,
        nothing
      )
    )
  )
```

12.1 Notations pour les variables des fonctions

12.1.1 Présentation

Jusqu'à présent la notation pour définir les variables des fonctions est très simple : c'est une liste de noms. Nous allons étendre ces notations, voyons d'abord un exemple pour en comprendre l'intérêt : Si une fonction f a en premier argument une paire p et qu'elle utilise dans son corps et à plusieurs reprise "#1 p", si dans la déclaration des variables de la fonction nous pouvons déclarer une variable f valant "#1 p" alors le corps de la fonction est plus concis et donc plus clair. Cet exemple serait aussi valable si l'argument est un dictionnaire, on pourrait préférer avoir une variable "*conclusion*" plutôt que d'écrire à plusieurs reprise "(*deduction .di 'conclusion'*)". Par exemple la notation pour une paire sera :
$\$F\ p = \$P[f, s] \rightarrow$ corps pouvant utiliser les variables p, f et s
Et si nous n'avons pas besoin d'utiliser p dans le corps nous pouvons écrire :
$\$F\ \$P[f, s] \rightarrow$ corps pouvant utiliser les variables f et s

Nous voyons que nous allons donc développer un véritable petit langage pour les variables des fonctions, ses expressions seront les expressions de variables, **varexp** en condensé. De même que le langage des notations Lazi évolue par l'ajout de notation, le langage des expressions de variables évoluera. Nous alors définir ici ses éléments de base.

12.1.2 noms de variable

Un word est une varexp.

12.1.3 Fonctions et listes de varexp

Nous avons défini les listes de variables ordinaires pour la définition des fonctions, nous allons l'étendre aux varexp.

Notation n°3 : variable(s): $x_1, \ldots, x_n, b$
Condition: $x_1, \ldots, x_n$ sont des varexp, $n \geqslant 2$
" $\$F\ x_1, \ldots, x_n \rightarrow b$ " → " $\$F\ x_1 \rightarrow \$F\ x_2 \ldots, x_n \rightarrow b$ ".

12.1.4 paires

Notation n°4 : variable(s): $x1, x2, b$
Condition: x1 et x2 sont des varexp
" $\$F\ \$P[x1, x2] \rightarrow b$ " → " $\Big(\$F\ f, p \rightarrow f\ \big(\#1\ p\big)\ \big(\#1\ p\big)\Big)\ \$F\ x1, x2 \rightarrow b$ ".

Remarque : On pourrait penser à une définition plus simple comme :

Notation n°5 : variable(s): $x1, x2, b$
Condition: x1 et x2 sont des varexp
" $\$F\ \$P[x1, x2] \rightarrow b$ " → " $\Big(\$F\ p \rightarrow \big(\$F\ x1, x2 \rightarrow b\big)\ \big(\#1\ p\big)\ \big(\#1\ p\big)\Big)$ ".

Mais dans cette notation p est une variable valide du corps de la fonction (b) à définir.

12.1.5 Assignation

Nous avons vu dans l'exemple de la présentation que nous pouvons accéder à la fois à la fois au contenu de la paire tout en accédant à la paire en entier. Cette fonctionnalité est obtenue par les assignations.

Notation n°6 : variable(s): v, x, b
Condition: v est un nom et x est une varexp
" $\$F\ v = x \rightarrow b$ " → " $\big(\$F\ f, a \rightarrow f\ a\ a\big)\ \$F\ v, x \rightarrow b$ ".

12.1.6 Dictionnaires

Notation n°7 : variable(s): $v1, ..., vn, x1, ..., xn, b$
Condition: v1,...,vn sont des noms et x1 ... xn sont des varexp
" $\$F\ \$D[v1 = x1, ..., vn = xn] \rightarrow b$ " → " $\big(\$F\ f, d \rightarrow f\ (d\ .di\ 'v1')\ ...\ (d\ .di\ 'vn')\big)\ \$F\ x1, ..., xn \rightarrow b$ ".

Appelons "assignation d'entrée de dictionnaire" les sous-varexp "vk=xk". Comme nous voudrons souvent utiliser le même nom de variable que le nom de l'entrée, nous allons définir une notation pour les assignation d'entrée de dictionnaire.

Notation n°8 : variable(s): x
Condition: x est une assignation d'entrée de dictionnaire qui est un nom
" x " → " $x = x$ ".

Ainsi, par exemple $\$F\ \$D[v1 = x1, v2] \to b$ sera équivalent à $\$F\ \$D[v1 = x1, v2 = v2] \to b$

12.1.7 Listes

Notation n°9 : variable(s): x, y, b
Condition: x et y sont des varexp
" $\$F\ x\ :l\ y \to b$ " → " $\Big(\$F\ f, l \to f\ (\#1\ l)\ (\#1\ l)\Big)\ \$F\ x, y \to b$ ".

Remarque : Une fonction définissant des variables pour le dernier élément de la liste peut prendre en argument une liste vide, tant que, dans ce cas, elle ne cherche pas à calculer la variable correspondant au pseudo dernier élément.

Chapitre 13

Les types

Pour la règle de la descente nous aurons besoin de représenter des déductions ainsi que les règles de déduction.

13.1 Principes

Il existe de nombreuses expressions de ces deux notions :

1. • langage courant : chose (Le mot "chose" du langage courant ne porte pas en soi l'idée que la chose possède une structure et se range dans des catégories. Mais comme c'est le cas pour tout ce qui nous entoure et donc cela va sans dire.)
   • logique mathématique : structure, modèle
   • algèbre : structure algébrique
   • informatique : objet, instance de classe
2. • langage courant : catégorie, classe
   • logique mathématique : théorie
   • algèbre : les structures de ... (par exemple "les structures d'anneaux")
   • informatique : type, classe

Cette partie de Lazi décrit ce qu'est une preuve, nous utiliserons le champs sémantique informatique car il est plus riche, mêmes si les structures décrites sont plus inspirées par les mathématiques.

13.2 Pour les mathématiciens

Si les mathématiques standards ne sont pas présentées sous un angle informatique c'est parce que l'on ne détaille pas avec la rigueur (scientifique) de Lazi ce qu'elles sont. Par exemple on se contente d'évoquer le remplacement de symboles sans vouloir détailler ce qu'est une liste ou comment représenter un symbole parmi une infinité d'autres.

Si on a besoin ici d'expliciter dans les moindres détails les fondations, et si on a besoin de le faire à partir d'outils extrêmement rudimentaires, c'est que la puissance de Lazi ne vient pas de règles de hauts niveaux mais au contraire de sa simplicité et de son auto-représentation.

13.3 Fonctionnement des types

Type Un type t est un dictionnaire ayant au moins les deux entrées suivantes :

domain : En mathématiques on dirait que domain définit l'ensemble de la structure t. Si $\left(t\ ._{df}\ \text{`}domain\text{'}\right)\ x = 1$ alors x est une **instance** de t. Les instances de t sont les choses dont "s'occupe" le type t, c'est à dire que la pluspart des fonctions de t prennent aux moins une instance en argument. Propriété à vérifier : aucune.

equal A pour sens : si x, y sont deux instances alors $\left(t\ ._{df}\ \text{`}equal\text{'}\right)\ x\ y$ définit si x et y sont égaux. Propriété à vérifier : les 3 propriétés de l'égalité sur les instances de t, ainsi que la propriété #1 $\left(\left(t\ ._{df}\ \text{`}equal\text{'}\right)\ x\ y\right)$

Remarque : Rappelons nous que $._{df}$ (voir #1) fournit le dictionnaire en premier argument (la variable "this" en informatique). Cela permet donc aux fonctions d'un dictionnaire d'utiliser les autres fonctions (et les données) de celui-ci.

Remarque : L'entrée "equal" n'est pas complètement indispensable, nous l'incluons ici car nous l'utiliserons un peu partout.

Remarque : x peut être n'importe quoi, comme un dictionnaire, une liste etc.

Remarque : Nous parlons d'ensemble mais ce n'est pas au sens de la théorie des ensembles, la fonction "domain" ne retourne pas forcément un booléen. Toutefois la forme faible de tiers exclu de Lazi nous permettra de raisonner sur des hypothèses du style $\left(t\ ._{df}\ \text{`}domain\text{'}\right)\ x = 1$.

Afin de familiariser le lecteur avec les types nous allons déjà définir quelques types simples. La structure récursive des preuves nous obligera vite à aborder d'autres notions.

Chapitre 14

Les types primitifs

Définissons les types pour les données de base. Deux entrées supplémentaires nous serons nécessaires :

14.1 Les entrées supplémentaires

replaceWord this w r i : Remplace le mot w par la formule r dans l'instance i. Cette fonction est utilisée pour remplacer une mot dans une preuve (par exemple pour remplacer une variable).

toFormula this i Retourne une 1-formule f telle que sa traduction en 0-formule soit égale à i. Voir la section "Niveau des représentations" pour en comprendre l'intérêt.

Chapitre 15

Outils pour les types évolués

15.1 Représentation

Si nous réfléchissons aux sens notre esprit aura tendance à utiliser des mots comme "odeur". Ce mot se réfère à nos souvenirs d'odeurs ainsi qu'à notre conception de notre sens de l'olfaction. Notre pensée utilise une représentation symbolique (le mot "odeur", ou quelque chose se rapprochant de ce mot) plutôt que directement l'ensemble des connaissances se référant au concept d'odeur.

Si nous voulons définir un type t ayant pour instances d'autres types, nous serons obligés de pratiquer une indirection de la même forme que notre esprit, car il est impossible de définir une fonction "domain" pouvant calculer si une chose est bien le type visé. Les instances seront donc des mots et nous utiliserons un dictionnaire pour relier le mot au type. t aura une entrée "object" qui retourne le type à partir de sa représentation (un mot).

Pour tous les types dont les instances représentent une chose nous définirons la fonction **object**.

15.2 incha

Notre esprit utilise une notion proche de celle des types :

- "odeur", ses instances sont les odeurs perçues par notre système olfactif.
- "sens" : ses instances sont nos sens (la vue, l'olfaction etc).
- "sensation" : ses instances sont par exemple la perception de la table par le coude, ou encore l'image perçue par la vue.

Nous voyons donc que les instances de "sensation" sont l'ensemble des instances des différents sens. Pour décrire une sensation, on commence par donner le sens concerné (par exemple "c'est une odeur de soupe").

Nous aurons les mêmes besoins. Nous dirons qu'un type est une union s'il sert à regrouper différents types en un seul. Par exemple les déductions auront différents types et nous aurons besoin de représenter une déduction quelconque. L'équivalent de "c'est une odeur de soupe" sera une liste d'instances, l'élément de queue sera l'instance du type de plus haut niveau jusqu'au plus bas (en tête). Une telle liste, une chaîne d'instances, sera désignée par "incha" (pour **in**stances **cha**in).

Définition informelle n° 2, incha : Pour t un type, on dit que l est une incha de t si l est une liste où le dernier élément est une instance de t et chaque élément suivant est une instance de l'objet correspondant à l'instance précédente (on parcourt donc la liste de la fin vers le début).

Pour manipuler les types dans les incha nous aurons besoin qu'ils définissent les entrées

object this i Même si les instances ne sont pas des représentations.

isTerminal this i qui vaut 1 si les objets de t ne sont pas des types et sinon 0.

Chapitre 16

Les déductions

16.1 maths

Pour définir ce qu'est une preuve valide nous avons besoin d'un ensemble de 4 données regroupées dans un dictionnaire et que nous appellerons "**maths**". Nous donnons ici un sens flou de ces quatre données, nous verrons plus bas le sens exacte :

topDeductions : Fourni les types de base des déductions. C'est un dictionnaire où les entrées sont les noms servant à représenter les types et les valeurs sont les types.

truths : Liste des vérités déjà établies (au départ ce sont les axiomes). Une vérité est ici une incha de #1 ($maths\ ._{di}\ `topDeductions'$). Nous avons l'habitude de ne considérer que les formules en vérités, mais ici nous étendons cette notion. Nous pouvons retrouver les formules en calculant les objets des inchas terminaux.

qualities : Fourni le dictionnaire des qualités des mots du langage (voir ci-dessous).

translations : Fourni la traduction entre la représentation des mots du langage et leur sens respectif.

```
///                                   Le type "pair"

/*
Type ``pair'',tl et tr sont les types des éléments de gauche et droite.
*/
$Def pairT = $F tl,tr ->
  $D[
    domain      = $F this,i ->
      isPair i &b
      ( tl .df `domain' , pairFirst i ) &b
      ( tr .df `domain' , pairSecond i )
    ,
    equal       = $F this,i,j ->
      ( tl .df `equal' , pairFirst i , pairFirst j ) &b
      ( tr .df `equal' , pairSecond i , pairSecond j )
    ,
    replaceWords = $F this,d,i ->
      $P[
        ( tl .df `replaceWords' , d , pairFirst i )
        ,
        ( tr .df `replaceWords' , d , pairSecond i )
      ]
    ,
    matchVars = $F this,i,lv,ass,model ->
      $Let mass2 = ( tl .df `matchVars' , pairFirst i, lv , ass , pairFirst model ),
      if ( isThing mass2
      ,
        // Prend les assignations trouvées dans la première partie et continue dans la seconde.
        ( tr .df `matchVars' , pairSecond i, lv, getThing mass2 , pairSecond model )
      ,
        nothing
      )
  ]


///                                   Le type list anonyme

// Nous allons définir un type ``anonymListT t'',  ses instances seront les listes où tous les
éléments ont sont des instances de t.

/*
Type de liste d'éléments d'un certain type t
*/
$Def anonymListT = $F t ->
  $D[
    domain       = $F this,i -> isList i &b ∀l x/i; ( t .df `domain' ) x
    ,
    equal        = $F this -> listEqual . t .df `equal'
    ,
    replaceWords = $F this,d,i -> listMap ( t .df `replaceWords' . d) i
    ,
    matchVars = $F this,i,lv,ass,model -> listEqualLength i model &b
      listFoldMaybe ($F $P[xi,xm],ass -> ( t .df `matchVars' ) xi lv ass xm) ass (list2Pair i
      model)
  ]

///                                   Le type list

/*
Pour certaines listes, le type des éléments dépend de l'émément précédant dans la liste. Dans ce
cas nous devons calculer récursivement les types des éléments à partir d'une valeur (appelé
contexte) de départ (r0), d'une fonction ft calculant le type à partir du contexte, et enfin d'une
fonction (next) calculant le contexte suivant.

Pour une liste $L[a,b,c] les types des éléments seront $L[ ft r0 , ft . next a r0 , ft . next b .
next a r0 ]

*/
$Def listT = $F ft,next,r0 ->
  $D[
    domain = $F this,i -> isList i &b
      isThing . listFoldMaybe
```

```
      ( $F x,r -> if(ft r .df  `domain' . x, just . next x r, 0m) )
      r0
      i
  ,
  equal = $F this,l,m -> listEqualLength l m &b
    isThing . listFoldMaybe
      ( $F $P[x,y],r -> if ( ( ft r .df `equal' ) x y , just . next x r , 0m ) )
      r0
      (list2Pair l m)
  ,
  // l est-elle le début de m (on peut avoir l=m).
  isStart = $F this,l,m -> listShorterOrEqual l m &b
    ( this .df `equal' , l , listReduceFirst m l )
  ,
  replaceWords = $F this,d,i -> listFoldMapFG ( $F x,r -> ( ft r .df `replaceWords' ) d x ) next
r0 i
  ,
  matchVars = $F this,i,lv,ass,model ->
    if
    (
      listEqualLength i model
    ,
      $Let res =
        listFoldMaybe
          (
            $F $P[xi,xm],$P[r,ass] ->
              $Let newAss = ( ft r .df `matchVars' ) xi lv ass xm ,
              if ( isThing newAss, just $P[ next xi r , getThing newAss ], 0m )
          )
          $P[r0,ass] (list2Pair i model)
      ,
      if ( isThing res , just . pairSecond . getThing res , 0m )
    ,
      0m
    )
  ,
  // Map la liste. La fonction de mapage prend en premier argument le type de l'élément et en
second l'élément.
  map = $F this,f,i -> listFoldMapFG ( $F x,r -> f (ft r) x ) next r0 i
  ,
  // Le type de départ.
  startType    = ft r0
  ,
  // Calcul le contexte du dernier élément. La liste ne doit pas être vide pour que le résultat
ait un sens.
  finalContext = $F this,i -> listFold next r0 . listInit i
  ,
  // Calcul le type de listLast i (pour $L[a,b,c] : ft . next b . next a r0. La liste ne doit pas
être vide pour que le résultat ait un sens.
  finalType = $F this,i -> ft . this .df `finalContext' . i
]



///                                          wordT


/*
Description correspondant au type ``mot''. ``replaceWords'' ne remplace pas le mot car replaceWords
sert à remplacer un mot à l'intérieur des formules.
*/
$Def wordT = $D[
  domain       = $F this -> isWord
  ,
  equal        = $F this -> wordEqual
  ,
  replaceWords = $F this,d,i -> i
  ,
  matchVars = $F this,i,lv,ass,model -> if ( i =w model , just ass , 0m )
]

///                                          formulaT
```

```
/*
Type ``formula''.
*/
$Def formulaT = $D[
  domain       = $F this -> isFormula
  ,
  equal        = $F this -> formulaEqual
  ,
  replaceWords = $F this -> replaceWords
  ,
  matchVars = $F this,i,lv,ass,model -> matchVars lv ass model i
]

///                                          dictT

// Nous allons définir un type ``dictT instanceType'' où  instanceType est le dictionnaire des
types des différentes valeurs.


/*
Type des dictionnaires, chaque entrée a son propre type, dt est le dictionnaire des types.
*/
$Def dictT = $F dt ->$D[
    domain      = $F this,i ->
      isDict i &b
      wordListEqual ( dictEntries dt , dictEntries i ) &b
      listAnd ( dictValues . dict2Map ( $F t,v -> t .df `domain' . v ) dt i )
    ,
    equal       = $F this,i,j ->
      listAnd .
      dictValues .
      dict2Map apply ( dict2Map ( $F t -> t .df `equal' ) dt i ) j
    ,
    replaceWords = $F this,d,i ->
      dict2Map ( $F t -> ( t .df `replaceWords' ) d ) dt i
    ,
    matchVars = $F this,i,lv,ass,model ->
      listFoldMaybe
        (
          $F $P[xie,xiv],ass ->
          ( dt .di xie .df `matchVars' ) xiv lv ass (model .di xie)
        )
        ass i
  ]

///                                          anonymDictT

// Nous allons définir un type ``anonymDictT t'' où  toutes les valeurs on le même type t .

/*
Type des dictionnaires, t est le type des valeurs.
*/
$Def anonymDictT = $F t ->
  $D[
    domain      = $F this,i -> isDict i &b
      // les valeurs doivent avoir le type demandé
      ∀l x/dictValues i; ( t .df `domain' ) x
    ,
    equal       = $F this,i,j ->
      wordListEqual ( dictEntries i , dictEntries j ) &b
      listAnd ( dictValues . dict2Map ( t .df `equal' ) i j )
    ,
    replaceWords = $F this,d,i -> dictMap ( ( t .df `replaceWords' ) d ) i
    ,
    matchVars = $F this,i,lv,ass,model ->
      if
      (
        wordListEqual ( dictEntries i , dictEntries model )
      ,
```

```
        listFoldMaybe
          (
            $F $P[xie,xiv],ass ->
            ( t .df `matchVars' ) xiv lv ass (model .di xie)
          )
          ass i
      ,
        0m
      )
  ]

///                                    Représentation

/*
Type dont les instances sont des mots (les entrées de l'argument dr). La fonction object retourne
la chose représentée.
*/
$Def symbolsSetT = $F dr ->
  $D[
    domain = $F this,i -> isWord i &b i ∈di dr
    ,
    equal  = $F this -> wordEqual
    ,
    object = $F this,i -> dr .di i
    ,
    replaceWords = $F this,d,i -> i
    ,
    matchVars = $F this,i,lv,ass,model -> if ( i =w model , just ass , 0m )
  ]

///                                    incha


// Le type d'une incha de t. Pour une incha vue en tant que listT, le contexte est directement le
type de l'élément.
$Def inchaT = $F t ->
  $Let parent = listT identityF ($F i,t -> t .df `object' . i) t ,
  // Type hérité de listT
  parent +l
  $D[
    // Retourne l'objet final
    object = $F this,i -> (parent .df `finalType') i .df `object' . listLast i
    ,
    // Est-ce que l'incha de t représente un objet ?
    isTerminal = $F this,i -> (parent .df `finalType') i .df `isTerminal' . listLast i
    ,
    // Retourne l'instance de plus haut niveau.
    topInstance = $F this,i -> listFirst i
  ]
```

16.1.1 La liste des vérités

Nous partons d'un maths où "truths" contient les inchas des règles de déductions (donc les 7 règles de bases déjà vues).

L'activité mathématique de base en Lazi consiste à ajouter une vérité validée à maths. Ainsi on procède par accumulation de vérités, ce qui est le processus déjà décrit dans la définition du système de production de vérités. Nous aurons donc une fonction de validation. Une chose est une vérité valide par rapport à un certain maths si c'est une incha de #1 ($maths\ ._{di}$ '$topDeductions$'). Cela ne veut rien dire d'autre que "une vérité consiste à compléter une règle valide".

16.1.2 Les qualités de mots

"maths" servira par la suite à étendre Lazi. Ces extensions introduiront des contraintes sur les mots. Par exemple les quantificateurs ne pourront pas être utilisés dans la variable "z" de la règle "distribute" (car dupliquer le quantificateur empêche de le tracer ce qui permet de le traduire) et pour cela nous avons besoin de l'information des contraintes sur les différents mots.

Pour cela, nous utilisons un dictionnaire où chaque mot du langage (même quand nous utiliserons plus tard des variables) sera une entrée et où la valeur sera une liste de ses qualité. Une qualité est représentée par un mot.

Nous introduisons tout de suite les qualités de mots car cela n'ajoute que quelques lignes et nous évite des redéfinitions plus tard. Les règles n'ont pas à connaître toutes les qualités possibles mais juste celles pour lesquelles elles imposent des obligations. Pour ne pas changer les règles de base par la suite nous devons définir les qualités suivantes :

copyable : Le mot peut être être dédoublés dans les déductions (s'il n'est pas copiable on peut le tracer dans les preuves, ce sera le cas des quantificateurs)

replaceable : le mot peut être remplacé (par exemple si x n'a pas cette qualité alors il ne doit pas être remplacé par $if\ 1\ x\ 1$).

16.1.3 Traduction des représentations des mots

Nous représenterons des déductions, nous aurons besoin par exemple de traduire la 1-formule conclusion de la déduction. Il nous faudra par exemple traduire «if» par if. Nous allons définir le dictionnaire de ces traduction.

Les mots du langage ne sont pas fixes, par exemple quand nous utiliserons une variable nous devrons ajouter le mot au dictionnaire. Le dictionnaire étant utilisé dans les n-preuves, nous devons définir sa formule correspondante.
Si par exemple le dictionnaire fait, à l'entrée 'v', correspondre $if\ 1$, cela signifie que dans les représentations de preuve il est valide d'utiliser le mot v, et que si l'on traduit une formule de la preuve (par exemple sa conclusion), alors v devra être traduit par $if\ 1$. Si dans la représentation de la preuve est construite une représentation de preuve, cette 2-preuve pourra utiliser v, en quoi sera-t-il traduit dans la 1-preuve ? Il suffit qu'il soit traduit par v, nous n'avons pas besoin (ni les moyens) de fournir le sens premier de v aux >0-preuves.

```
///                                     topAssertionT

/*
Le type des types de plus haut niveau des règles de déduction.
*/
$Def topAssertionT = $F maths ->
  $Let parent = symbolsSetT  ( maths .di `topAssertions' ),
  parent ∪di
  $D[
    // Toute instance est valide, c'est à dire qu'une vérité ne peut pas rendre un type d'assertion
    invalide.
    isValid      = $F this,i,truths -> 1
    ,
    object       = $F this,i -> (parent .df `object') i maths
    ,
    isTerminal   = $F this,i -> 0
    // Remarque : cela interdit qu'une instance aboutisse directement à une conclusion.
  ]


///                                     assertionT

/*
Le type d'une assertion. Une assertion est une chose qui a toutes les caractéristiques d'une vérité
mis à part la vérification de la cohérence par rapport aux autres vérités.
*/
$Def assertionT = $F maths ->
  inchaT (topAssertionT maths)

///                                     deductionT

// Une déduction est une assertion terminale.

// Pour un maths et une liste de vérités données, si la déduction est valide alors son objet est
une assertion  vraie (que l'on peut donc ajouter aux vérités).


/*
Le type "déduction" pour un maths donné.
*/
$Def deductionT = $F maths ->
  $Let it = assertionT maths ,
  it ∪di
  $D[
    domain = $F this,i -> (it .df `domain') i &b
      (it .df `isTerminal') i
    ,
    isValid = $F this,i,truths ->
      isThing . listFoldMaybe (
        $F x,t -> if( (t .df  `isValid') x truths , just . t .df `object' . x , 0m) ,
        topAssertionT maths ,
        i
      )
  ]

///                                     Les vérités

// On utilisera des listes pour stocker les vérités. On construit ces listes en y ajoutant les
objets des déductions.

/*
i est-il un des éléments de la liste des vérités de maths ?
*/
$Def isInTruths = $F maths,truths,i -> inList (assertionT maths .df `equal' ) truths i


/*
l est-elle une liste de vérités pour maths.
*/
$Def isListOfTruths = $F maths,truths,l -> listInclude (assertionT maths .df `equal') l truths
```

```
///                                Les types des assertions

/*
Ajoute à maths un nouveau type d'assertion.
*/
$Def addTopAssertion = $F maths,name,type -> dictMod maths `topAssertions' . addToList (pair name
type)


///                                Le type proofT maths



/*
Retourne le type d'une preuve non vide pour un maths donné. C'est une liste (non vide) d'instances
de deductionT.
*/
$Def proofT = $F maths ->
  $Let parentT = anonymListT (deductionT maths),
  parentT ∪di
  $D[
    // We add the condition of being not empty
    domain = $F this,i -> ( parentT .df `domain' ) i &b notEmptyList i
    ,
    // f est-elle la conclusion de la preuve (la conclusion de la dernière déduction de la preuve).
    isProofOf = $F this,i,f ->
      ( assertionT maths .df `equal' , this .df `object' . i , f )
    ,
    // Vérifie que la preuve est valide par rappport à une certaine liste de vérités.
    isValid = $F this,i,truths ->
      isThing . listFoldMaybe ( $F d, truths ->
        if
        (
          ( deductionT maths .df `isValid' ) d truths
        ,
          just . truths :l ( deductionT maths .df `object' ) d
        ,
          nothing
        )
      ) truths i
    ,
    // Applique la preuve sur une liste de vérités et retourne la liste où les vérités sont
    ajoutées.
    truthsAfter = $F this,i,truths -> listFold
      ( $F d,truths -> truths :l (deductionT maths .df `object' . d) ) truths i
    ,
    isTerminal  = $F this,i -> 1
    ,
    // L'assertion déduite par la dernière déduction.
    object      = $F this,i -> deductionT maths .df `object' . listLast i
  ]

/*
Vérifie que p est une preuve , que f est une formule qui est la conclusion de p. Cette fonction est
utilisée dans la règle de la descente.
*/
$Def checkProofOfFormula =  $F maths, truths, p, f ->
  (proofT maths .df `domain') p &b
  (proofT maths .df `isValid') p truths &b
  isFormula f &b
  (proofT maths .df `isProofOf') p $L[`formula', f]


///                                Les qualités des mots
/*
Les qualités des mots de base de Lazi.
*/
$Def laziAllQualities = $L[ `copyable', `replaceable' ]

/*
Le dictionnaire des qualités des mots de base. L'entrée est le mot, la valeur est la liste des
qualités.
```

```
*/
$Def laziWordsQualities =
  listMap ( $F w -> pair w laziAllQualities ) $L[ `equal', `distribute' , `one' , `zero' , `if' ]

/*
La formule f a-t-elle les qualités requises de la liste ql et tous ses mots sont-ils connus du
dictionnaire des qualités de maths.
*/
$Def formulaHasQualities = $F maths,ql,f ->
  foldFormula (
    $F w -> $Let r = applyJust ( listInclude wordEqual ql , dictApplyMaybe ( maths .di
    `wordsQualities') w  ) ,
    isThing r &b getThing r // r doit valoir just 1, sinon la réponse est 0.
  ) boolAnd f

/*
Ajoute des mots et leurs qualités à un maths.
*/
$Def addWordsToMaths = $F maths,ql ->
  dictMod maths `wordsQualities' . concatToList ql

/*
Retourne toutes les qualités possibles pour les mots.
*/
$Def allQualities = $F maths ->
  listsUnion wordEqual . dictValues . maths .di `wordsQualities'

///                    Traduction des représentations des mots

/*
Dictionnaire donnant le sens des mots basic Lazi.
*/
$Def basicLaziDict = $D[ equal = equal , distribute = distribute , one = one , zero = zero , if =
if ]

/*
Traduit une 1-formule en 0-formule à l'aide du dictionnaire "translations" de maths.
*/
$Def translateFormulaMaths = $F maths,f -> translateFormula ( maths .di `translations' ) f

/*
Ajoute un mot v ayant les qualités ql dans le dictionnaire des qualités de maths.
*/
$Def addVarToMaths = $F maths,v,ql ->
  dictSet maths `wordsQualities' ( dictSet ( maths .di `wordsQualities' ) v ql )

/*
Ajoute un mot v ayant a traduction f dans le dictionnaire des traductions de maths.
*/
$Def addTransToMaths = $F maths,v,f ->
  dictSet maths `translations' ( dictSet ( maths .di `translations' ) v f )


///                                    formulaAssertionT

/*
Le type formulaAssertionT est le type de la forme la plus simple d'assertion. C'est aussi une
déduction qui déduit la formule elle-même. Un exemple en français serait "Et que peut-on déduire du
fait que Paul a un chat ? Que Paul a un chat.".
*/
$Def formulaAssertionT = $F maths ->
  formulaT ∪di
  $D[
    domain       = $F this,i ->
      isFormula i &b
      formulaUseWords ( dictEntries ( maths .di `wordsQualities' ) ) i
    ,
    // On ne peut utiliser une formule comme déduction que si la formule est une vérité.
    isValid     = $F this,i,truths -> ∃l t/truths; (assertionT maths .df `equal') $L[ `formula' , i
    ] t
    ,
    isTerminal  = $F this,i -> 1
```

```
    ,
    object      = $F this,i -> $L[ `formula' , i ]
  ]


///                                     applyRuleT

/*
Le type des applications des règles. Les instances sont les valeurs des variables d'une règle.
C'est un type terminal, il a donc pour objet l'assertion déduite (la conclusion). Ses arguments
sont:
 - maths
 - conditions : la liste des assertions à variables conditions de la règle de déduction.
 - conclusion : l'assertion à variables conclusion de la règle.
 - wordsQualities : dictionnaire des variables, en valeur la liste des qualités nécessaires.

Une instance est un dictionnaire "nom de variable" -> valeur.
*/
$Def applyRuleT = $F maths,$D[wordsQualities,conditions,conclusion] ->
  $Let parent = dictT ( dictMap ( constantF formulaT ) wordsQualities ),
  $Let newMaths = addWordsToMaths maths wordsQualities ,
  parent ∪di
  $D[
    domain      = $F this,i ->
      // i est-il un dictionnaire de formules de mêmes entrées que "wordsQualities"
      ( parent .df `domain' ) i &b
      // Les valeurs ont-elles leurs mots connus et vérifiant les qualités requises.
      listAnd ( dictValues . dict2Map ( formulaHasQualities maths ) wordsQualities i )
    ,
    isValid     = $F this,i,truths ->
      // Les conditions sont-elles dans les vérités de maths
      isListOfTruths maths truths ( listMap ( assertionT newMaths .df `replaceWords' . i )
      conditions )
    ,
    isTerminal  = $F this,i -> 1
    ,
    object      = $F this,i -> ( assertionT newMaths .df `replaceWords' ) i conclusion
  ]

///                                     ruleT

/*
Le types des règles (comme les règles de Lazi déjà exprimées). Ses instances seront les paramètres
définissant une règle, ses objets seront les applyRuleT.
*/
$Def ruleT = $F maths ->
  // On calcul le type de l'instance à partir de l'instance! Mais si on regarde précisément le
  calcul on voit que l'on ne présuppose rien sur l'argument de 'domain'.
  $Let instanceType = $F $D[wordsQualities] -> dictT
    $Let newMaths = addWordsToMaths maths wordsQualities ,
    $D[
      wordsQualities = anonymDictT . anonymListT wordT
      ,
      conditions = anonymListT . assertionT newMaths
      ,
      conclusion = assertionT newMaths
    ]
  ,
  $D[
    domain      = $F this,i -> ( instanceType i .df `domain' ) i
    ,
    // On ne peut utiliser une règle comme déduction que si la règle est une vérité.
    isValid     = $F this,i,truths -> ∃l t/truths; (assertionT maths .df `equal') $L[ `rule' , i ]
    t
    ,
    equal       = $F this,i,j -> (instanceType i .df `equal') i j
    ,
    object      = $F this,i -> applyRuleT maths i
    ,
    isTerminal  = $F this,i -> 0
    ,
    // Les variables ne sont pas des mots ordinaires et ne sont pas à remplacer, c'est pourquoi on
```

```
    les enlève du dictionnaire des remplacements.
    replaceWords = $F this,d,i@$D[wordsQualities] ->
      (instanceType i .df `replaceWords' , dictReduce d (dictEntries wordsQualities), i)
    ,
    matchVars = $F this,i -> (instanceType i .df `matchVars') i
  ]

///                                    provedRuleT

// provedRuleT est une règle pour ajouter une règle. Son instance sera composée d'une preuve (de la
règle) en plus de la règle (une instance de ruleT). La preuve devra se faire dans le contexte de
maths et de la règle: on ajoute les conditions en hypothèses et les variables en mots.

/*
À partir d'une instance d'une règle, ajoute les variables dans les mots et les conditions comme
hypothèses. L'entrée translations de maths n'est pas touchée car il n'y a pas de traduction des
variables.
*/
$Def addRuleContext = $F maths, $D[wordsQualities] -> addWordsToMaths maths wordsQualities

/*
Une instance de ce type sera une preuve d'une certaine règle. Cela permetra donc d'ajouter en
vérité des règles. Pour prouver une règle il faut fournir la preuve de la conclusion dans le maths
où les hypothèses sont ajoutées ainsi que les variables (qui doivent être nouvelles) et leurs
contraintes.
*/
$Def provedRuleT = $F maths ->
  // Nous avons besoin de l'instance pour calculer son type! La fonction "domain" a quand même un
  sens car nous n'avons besoin que de l'entrée 'rule' et elle est vérifiée en premier (c'est un
  listRevFold qui vérifie le domaine des dictionnaires.
  $Let instanceType = $F $D[rule] -> dictT
    $D[
      rule  = ruleT maths
      ,
      proof = proofT . addRuleContext maths rule
    ]
  ,
  $D[
    domain       = $F this,i@$D[rule = rule@$D[wordsQualities,conclusion],proof] ->
      // On n'utilise pas instanceType pour vérifier le domaine car avant de vérifier proof il faut
      s'assurer que les variables de rule ne sont pas des mots de maths.
      $Let pt = proofT . addRuleContext maths rule , // Le type de la preuve.
      ( ruleT maths .df `domain' ) rule &b
      // Les variables de la règle ne doivent pas être des mots du langage de maths
      isEmptyList (
        listIntersect (wordEqual , dictEntries wordsQualities , dictEntries ( maths .di
        `wordsQualities' ) )
      ) &b
      // proof est-il dans le domaine ?
      ( pt .df `domain' ) proof &b
      // proof doit prouver la conclusion de la règle.
      ( pt .df `isProofOf') proof conclusion
    ,
    isValid      = $F this,i@$D[rule = rule@$D[conditions],proof],truths ->
      ( proofT ( addRuleContext maths rule ) .df `isValid' , proof , truths +l conditions )
    ,
    equal        = $F this,i1,i2 -> (instanceType i1 .df `equal') i1 i2
    ,
    object       = $F this,$D[rule] -> $L[ `rule' , rule ]
    ,
    isTerminal   = $F this,i -> 1
    ,
    // Les variables ne sont pas des mots ordinaires et ne sont pas à remplacer, c'est pourquoi on
    les enlève du dictionnaire des remplacements. La fonction de remplacement de la règle le fait,
    mais on doit aussi le faire pour la preuve.
    replaceWords = $F this,d,i@$D[wordsQualities] ->
      ( instanceType i .df `replaceWords' , dictReduce d (dictEntries wordsQualities) , i )
    ,
    matchVars = $F this,i -> (instanceType i .df `matchVars') i
  ]
```

```
///                                  laziTopAssertions

// Nous avons maintenant toutes les définitions pour définir l'entrée "topAssertions" de maths.

/*
Dictionnaire des types d'assertions de Lazi.
*/
$Def laziTopAssertions =
  $D[
    formula = formulaAssertionT,
    rule = ruleT,
    provedRule = provedRuleT,
    // On ajoute proofT juste pour pouvoir faire des sous-preuves, ce n'est pas une nécessite et
    c'est un cas particulier de proovedRule. Faire des sous-preuves évite d'encombrer la liste des
    vérités.
    proof = proofT
  ]
```

Chapitre 17

maths

17.1 Présentation

Pour définir maths il nous faut ses 3 entrées : les contraintes sur les mots, les types de règles de déduction et "truths", la liste des inchas autorisés. Il nous reste cette dernière à définir.

Les règles de généralisation et de tiers exclu sont récursives : c'est à dire qu'elle expriment ce qu'est une 1-preuve, et pour exprimer ce qu'est une 1-preuve il nous faut "truths". Ainsi dans une 1 preuve on pourra utiliser les règles de déductions et raisonner avec des 2-preuves etc.

Par l'introduction d'une hypothèse ou d'une variable les règles sur les preuves modifient la définition de ce qu'est une preuve (par la modification de truths). Il ne sera donc pas suffisant de transformer un "truths" prédéfini en formule.

La récursivité n'est pas celle habituelle car elle se produit aussi sur le n des "n-preuves". Pour arriver à cela nous utiliserons la combinaisons de la récursivité habituelle plus la transformation en formule.

Nous avons deux manières pour trouver la formule correspondant à (c'est à dire "dont la traduction est") une chose.

Par les notations : par exemple la formule correspondant à #1 est «#1». Cela ne s'applique qu'à des choses entièrement fixées.

par toFormula : toFormula est défini pour des choses au type définis (comme des listes).

```
///
///                                    Les règles
///


///                                 Égalité et vérité

/*
Instance de ruleT pour la règle.
*/
$Def equalityAndTruthRule = $L[ `rule' , $D[
    wordsQualities = $D[ x = 0l , y = 0l ]
    ,
    conditions  = $L[ $AF[x] , $AF[x = y] ]
    ,
    conclusion  = $AF[y]
  ]
]

///                                 Arguments égaux

/*
Instance de ruleT pour la règle.
*/
$Def equalArgumentsRule = $L[ `rule' , $D[
    wordsQualities = $D[ x = 0l , y = 0l , z = 0l ]
    ,
    conditions  = $L[ $AF[x = y] ]
    ,
    conclusion  = $AF[z x = z y]
  ]
]

///                                 Fonctions égales

/*
Instance de ruleT pour la règle.
*/
$Def equalFunctionsRule = $L[ `rule' , $D[
    wordsQualities = $D[ x = 0l , y = 0l , z = 0l ]
    ,
    conditions  = $L[ $AF[x = y] ]
    ,
    conclusion  = $AF[x z = y z]
  ]
]

///                                   Distribute

/*
Instance de ruleT pour la règle.
*/
$Def distributeRule = $L[ `rule' , $D[
    wordsQualities = $D[ x = 0l , y = 0l , z = $L[ `copyable' ] ]
    ,
    conditions  = 0l
    ,
    conclusion  = $AF[distribute x y z = x z ( y z )]
  ]
]

///                                   if sur 1

/*
Instance de ruleT pour la règle.
*/
$Def ifOnTruthRule = $L[ `rule' , $D[
    wordsQualities = $D[ x = $L[ `replaceable' ] , y = 0l ]
    ,
    conditions  = 0l
    ,
```

```
    conclusion  = $AF[if 1 x y = x]
  ]
]

///                                    if sur 0

/*
Instance de ruleT pour la règle.
*/
$Def ifOnFalseRule = $L[ `rule' , $D[
    wordsQualities = $D[ x = 0l , y = $L[ `replaceable' ] ]
    ,
    conditions  = 0l
    ,
    conclusion  = $AF[if 0 x y = y]
  ]
]

///                                    Tiers exclu

/*
Instance de ruleT pour la règle.
La règle du tiers exclu. Si on a if x y z = 1 , si t est prouvé sous l'hypothèse x = 1 ainsi que
sous l'hypothèse x = 0, alors on déduit t.
*/
$Def excludedMiddleRule = $L[ `rule' , $D[
    wordsQualities = $D[ x = 0l , y = 0l , z = 0l , t = 0l ]
    ,
    conditions  = $L[
      $AF[ isBoolean . if x y z ] ,
      $L[ `rule' , $D[ wordsQualities=0l, conditions = $L [ $AF[ x = 1 ] ], conclusion = $AF[ t ] ]
      ],
      $L[ `rule' , $D[ wordsQualities=0l, conditions = $L [ $AF[ x = 0 ] ], conclusion = $AF[ t ] ]
      ]
    ]
    ,
    conclusion  = $AF[t]
  ]
]



///                                    induction

/*
Instance de ruleT pour la règle.
La règle de l'induction. Si p est vrai sur la liste vide et si quand il est vrai sur une liste
alors il est vrai sur la liste ayant un élément en plus, alors p est vrai pour toute liste.
*/
$Def inductionRule =
$L[ `rule' , $D[
    wordsQualities = $D[ l = 0l , p = 0l ]
    ,
    conditions  = $L[
      $AF[ isList l ] ,
      $AF[ p 0l ] ,
      $L[ `rule' ,
        $D[
          wordsQualities = $D[ m = 0l , x = 0l ],
          conditions = $L [ $AF[ isList m ], $AF[ p m ] ],
          conclusion = $AF[ p . m :l x ]
        ]
      ]
    ]
    ,
    conclusion  = $AF[ p l ]
  ]
]
```

```
///
///                                        laziMaths
///

/*
La chose "maths" pour Lazi.
*/
$Def laziMaths =
  $D[
    topAssertions = laziTopAssertions
    ,
    wordsQualities   = laziWordsQualities
    ,
    translations  = basicLaziDict
  ]

///
///                                        laziRules
///

/*
Les règles de déduction Lazi.
*/
$Def laziRules =
  $L[
    equalityAndTruthRule
  ,
    equalArgumentsRule
  ,
    equalFunctionsRule
  ,
    distributeRule
  ,
    ifOnTruthRule
  ,
    ifOnFalseRule
  ,
    excludedMiddleRule
  ,
    inductionRule
  ]
```