Outils de preuve et vérification

Pierre Courtieu

5 octobre 2011

Table des matières

1 Préambule **5**
1.1 Rappels de logique 5

I Preuve de programmes fonctionnels (NFP119) **7**

2 Introduction **9**
2.1 Motivations 9
2.2 Rappels de programmation fonctionnelle 10
2.2.1 Ordre supérieure : « Les fonctions sont des valeurs comme les autres » 11
2.2.2 Polymorphisme : « Ne pas restreindre les type des fonctions inutilement » 12
2.3 Propriétés sur les programmes fonctionnels 13
2.4 Valeurs définies, valeurs indéfinies 13
2.5 Équation de programme 14
2.6 Premières preuves de programmes 14

3 Ensembles, relations, fonctions, ordres **17**
3.1 Notations logiques 17
3.2 Ensembles 17
3.3 Relations 18
3.4 Fonctions 18
3.5 Ordres 19
3.5.1 Définitions 19
3.5.2 Ordres bien fondés 20

4 Récurrence et induction **23**
4.1 Récurrence «faible» 23
4.2 Récurrence «forte» 23
4.3 Induction 24
4.3.1 Ensemble défini par induction - intuition 24
4.3.2 Ensemble défini par induction - définition 25
4.3.3 Raisonnement par induction 26
4.3.4 Raisonnement par induction bien fondée 27

5 Preuves de programmes fonctionnels **29**
5.1 Exemple 29
5.2 Exercices 29

II Preuve de programmes impératifs (NFP209) 35

6 Introduction 37

7 Systèmes logiques 39
7.1 Les règles d'inférence . . . 39
7.2 Arbre de déduction . . . 40

8 La logique de Hoare 43
8.1 Les triplets de Hoare . . . 43
8.2 Correction d'un triplet de Hoare . . . 44
8.3 Les instructions . . . 44
8.4 Les assertions . . . 45
8.5 Les règles de Hoare . . . 45
8.5.1 Affectation . . . 46
8.5.2 Séquence . . . 47
8.5.3 Conséquence . . . 48
8.5.4 Conditionnelle . . . 49
8.5.5 While . . . 49
8.5.6 While avec variant (correction totale) . . . 50
8.5.7 Les tableaux . . . 51

9 Conclusion 53
9.1 Les difficultés . . . 53
9.2 Programme annoté . . . 53
9.3 exercices . . . 54

10 Jessie 55
10.1 max . . . 55
10.2 swap . . . 55
10.3 divide . . . 56
10.4 Autres exemples . . . 56

Bibliographie 59

III Solutions des exercices 61

Chapitre 1

Préambule

Ce polycopié concerne les deux unités d'enseignement (UE) suivantes du CNAM : `NFP119` (programmation fonctionnelle, niveau Licence 2) et `NFP209` (programmation rigoureuse, niveau master 1). Plus exactement il traite de la partie de ces deux cours qui concerne la *preuve de programme*. Le polycopié est divisé en deux parties correspondant à ces deux UE : preuve de programmes fonctionnels et preuve de programmes impératifs.

Il nécessite, pour une lecture sereine, d'avoir des notions de base en logique qui peuvent être acquises en suivant au préalable l'UE `NFP108` (logique niveau Licence 2). En particulier dans la deuxième partie le chapitre 1.1 « Rappels de logique » est un rapide survol du contenu de `NFP108`. Bien entendu il est également indispensable d'avoir des notions solides en programmation pour suivre ces deux cours.

Prière d'envoyer vos remarques à l'adresse suivante : `Pierre(pt)Courtieu(at)cnam(pt)fr`.

1.1 Rappels de logique

Rappels de logique

- true, false
- $\neg p$ « non p »
- $p \wedge q$ « p et q »
- $p \vee q$ « p ou q »
- $p \Rightarrow q$ « p implique q ».
- $\forall x, P$ « pour tout x, $P(x)$ est vrai ».
- $\exists x, P$ signifie « il existe au moins un x tel que $P(x)$ est vrai ».

$p \Rightarrow q$ Attention !

$p \Rightarrow q$ « p implique q ».

Attention piège !

- **soit p est fausse,**
- soit p et q sont toutes les deux vraies.

Nous ferons appel à des notations logiques simples et usuelles, dont voici un récapitulatif :

- true est la propriété vraie
- false est la propriété fausse
- $\neg p$ signifie « non p »
- $p \wedge q$ signifie « p et q »
- $p \vee q$ signifie « p ou q »

- $p \Rightarrow q$ signifie « p implique q ». Attention piège ! c'est-à-dire que *soit p est fausse*, soit p et q sont vraies aussi.
- $\forall x, P$ signifie « pour tout x, P est vrai ». En général x apparaît dans P et on note $P(x)$. Il est fréquent d'écrire $\forall x \in E, P(x)$ pour $\forall x, (x \in E \rightarrow P(x))$
- $\exists x, P$ signifie « il existe au moins un x tel que P est vraie ». En général x apparaît dans P et on note $P(x)$. On écrit souvent $\exists x \in E, P(x)$ pour $\exists x, (x \in E \wedge P(x))$.

Cette liste respecte l'ordre standard de *précédence*, du symbole le plus fort ($\neg$) au symbole le plus faible ($\exists$ et $\forall$). Par conséquent l'expression suivante :

$$\forall x \in \mathbb{N}, x > 3 \wedge x < 4 \vee \neg x > 12 \Rightarrow x = 3$$

se parenthèse implicitement comme ceci :

$$\forall x \in \mathbb{N}, \overbrace{(\overbrace{(\overbrace{(x > 3 \wedge x \leq 6)} \vee \overbrace{(\neg(x > 12))})} \Rightarrow \overbrace{x = 3})}$$

Remarque : $x > 3$, $x = 3$... sont des propriétés atomiques.

Dans ce cours, nous effectuerons implicitement les simplifications suivantes :

- $p \vee false \ \equiv \ p$,
- $p \wedge true \ \equiv \ p$,
- $true \Rightarrow p \ \equiv \ p$,
- $false \Rightarrow p \ \equiv \ true$,
- $p \Rightarrow true \ \equiv \ true$, c'est-à-dire que $p \Rightarrow true$ ne contient aucune information sur p,
- $p \Rightarrow false \ \equiv \ \neg p$.

Dans la deuxième partie (chapitre 7) de ce polycopié nous introduirons la notions de *système logique* et de *règle d'inférence*.

Première partie

Preuve de programmes fonctionnels (`NFP119`)

Chapitre 2

Introduction

2.1 Motivations

Cette partie est une introduction à la programmation raisonnée. La programmation de qualité passe par la compréhension rigoureuse (mathématique) des objets informatiques. Le cours comporte deux parties : La programmation fonctionnelle et les mathématiques pour l'informatique.

Buts du cours

But : Approche mathématique de la programmation

- Pourquoi ? Bon programmeur comprends que :
  - Programme = objet mathématique
  - Bon programme = "bon" objet mathématique
    (s'arrête, calcul le bon résultat etc)
- Comment ? À travers deux thèmes intéressants par ailleurs :
  - Programmation Fonctionnelle
  - Mathématique pour l'informatique

Le cours commence par aborder le paradigme de la programmation fonctionnelle. Ceci présente deux intérêts.

Premièrement la programmation fonctionnelle possède des bases mathématiques plus rigoureuses et plus simples que la programmation impérative classique.

Deuxièmement c'est un paradigme de programmation très puissant lorsqu'il est utilisé avec un langage adapté (nous utiliserons `Objective Caml` (`ocaml`)). Il permet de développer des algorithmes complexes très aisément. Le transparent ci-dessous donne un aperçu (en `ocaml`) d'une des particularités des langages fonctionnels : les fonctions sont des données comme les autres. Elles peuvent être passées en paramètre d'autres fonctions, être retournées comme résultat d'autres fonctions etc.

Programmation Fonctionnelle

- Origine : meilleure compréhension de la programmation
- $\neq$ programmation impérative
- Possible avec les langages classiques (objets, `void*`)
- Langage dédié(`ocaml`) + simple + puissant (que C ou Java)
- Fonction = fonction mathématique = objet du langage

```
let f = fun x -> x+1;;
let g = fun y -> y 8;;

g f;;                           ⟶ 9
g (fun x -> x*2);;              ⟶ 16
g (fun x -> (fun z -> x/z));;   ⟶ fun z -> 8/z
```

La deuxième partie du cours aborde les outils de base pour l'étude des objets informatiques. En effet, pour de raisonner sur les structures de données et sur les programmes, il est nécessaire d'utiliser des techniques mathématiques spécifiques : la récurrence, le raisonnement par induction etc. Cette partie sera illustrée par des exemples de programmes fonctionnels. Le présent document aborde cette deuxième partie du cours : la preuve de programmes fonctionnels. Il contient des extraits du polycopié de mathématiques pour l'informatique d'Arnaud Lallouet [4]. Il n'est pas inutile d'avoir des notions de logique (`NFP108`) pour comprendre le contenu de cette première partie.

L'étude des programmes impératifs, plus difficile, fait l'objet d'autres cours (`NFP209`) dont celui-ci est un pré-requis et est présentée dans la deuxième partie de ce document.

Mathématique pour l'informatique

Raisonner sur les objets informatiques :

- Structures de données (dans ce cours)
- Programmes fonctionnels (dans ce cours)
- Programmes impératifs (+ difficile : NFP108, NFP209...)

Outils utiles :

- Notions mathématiques
  ensembles définis par induction, ordre bien fondé etc
- Techniques de démonstration
  récurrence, induction etc

2.2 Rappels de programmation fonctionnelle

Programmation fonctionnelle

- Pas d'effets de bord : programme *p* retourne *toujours* le même résultat.
- Pas de variable *mutable* (modifiable)
- Stocker des valeurs *immutables* en mémoire pour les utiliser plusieurs fois

Programmation fonctionnelle : syntaxe

Java syntax

```
public static int f(int i) {
  if (i<=0) { return 1; }
  else { return (i*f(i-1)); }
}
```

Ocaml syntax

```
let f i =
  if i<0 then 1
  else i + f(i-1);;
```

La programmation fonctionnelle désigne *un style de programmation* dans lequel on n'utilise pas *d'effet de bord*, c'est-à-dire dans lequel *un programme* `p` *appliqué à un paramètre* `a` *retourne toujours le même résultat*. En d'autre termes *l'état* de la machine sur lequel `p(a)` est exécuté n'a pas d'influence sur le résultat de `p(a)`. En particulier la caractéristique principale d'un programme fonctionnel est qu'*il n'utilise pas de variable (ni de tableau) mutable (c'est-à-dire modifiable)*. Une valeur peut être stockée en mémoire afin d'être utilisée plusieurs fois, mais cette valeur n'est pas modifiable. Nous verrons dans la suite que ceci rend l'analyse des programmes fonctionnels beaucoup plus simple que les programmes à effets de bord car une programme peut être considéré *indépendemment du contexte dans lequel il est appelé*, seule la valeur de ses arguments détermine son comportement.

La programmation fonctionnelle étant un style de programmation, elle peut être mise en œuvre avec tout langage de programmation. Néanmoins pour être suffisamment puissante il est préférable de l'utiliser avec un langage permettant *l'ordre supérieur* et le *polymorphisme*. Ces deux caractéristiques, décrites plus bas, sont en général considérées comme indispensables à la programmation fonctionnelle. À tel point qu'on les inclue souvent dans la définition même de la programmation fonctionnelle.

Les deux sections suivantes, tout en présentant succinctement l'ordre supérieur et le polymorphisme, sont une présentation de la syntaxe d'un langage qui possède ces caractéristiques : `Objective Caml` (`ocaml`). Nous en recommandons donc la lecture car les programmes que nous analyserons dans la suite seront écrits dans cette syntaxe, même si ils ne seront ni polymorphes ni d'ordre supérieur.

Pour plus de détail sur ces notions, se reporter à la partie du cours de `NFP119` concernant la programmation fonctionnelle et `ocaml`.

2.2.1 Ordre supérieure : « Les fonctions sont des valeurs comme les autres »

Dans les langages d'ordre supérieur, les fonctions sont considérées comme des valeurs normales, au même titre que les entiers ou les booléens, et peuvent donc être mises dans des variables, passées en paramètres d'une fonction, être retournées comme résultat d'une fonction etc.

Par exemple voici la définition de la valeur `1`, de la chaîne de caractère `"toto"` *et de la fonction* `plusdeux` qui retourne la valeur de son argument plus deux. Chacune de ces valeurs est stockée dans une variable (non mutable) :

```
let un = 1
let s = "toto"
let plusdeux = fun x -> x + 2
```

Notez que le type de l'argument de `plusdeux` (ici `int`) est *inféré automatiquement* par `ocaml`. On peut également utiliser la syntaxe suivante, plus pratique dans certains cas, où l'argument n'est pas introduit par le mot-clé **`fun`** mais déclaré après le nom de la fonction :

```
let plusdeux x = x + 2
```

Pour utiliser une fonction, il faut *l'appliquer* à (tout ou partie de) ces arguments. Pour cela, on écrit les arguments après le nom de la fonction, sans virgule et sans parenthèse [1] (dans la suite on indique la valeur des expressions à l'aide du symbole ⇝, il s'agit du résultat du calcul des expressions) :

```
plusdeux 3      ⇝ 5
plusdeux (plusdeux 2)      ⇝ 6
```

Mais il est possible d'utiliser la fonction `plusdeux` en tant que valeur, par exemple comme valeur pour une autre variable, auquel cas cette nouvelle variable peut être utilisée comme `plusdeux` :

1. En `ocaml`, les parenthèses ne servent qu'à résoudre d'éventuelles ambiguïtés. Par exemple l'expression `f g y` signifie « `f` appliquée à `g` et à `y` ». Si au contraire on désire écrire « `f` appliquée à `g y` » il faut écrire `f (g y)`.

```
let mafonction = plusdeux
mafonction 3      ⤳ 5
mafonction (plusdeux 2)      ⤳ 6
mafonction (mafonction 2)      ⤳ 6
```

Les fonctions peuvent aussi être retournées comme résultat d'une autre fonction, ici les deux fonctions `plusdeux` et `fun a -> a+3` sont les deux résultats possibles de `f` :

```
let f x = if x=0 then plusdeux else (fun a -> a+3)
```

Dans ce cas le résultat de `f` peut donc être utilisé comme une fonction, c'est-à-dire appliqué à un entier :

```
(f 0) 7      ⤳ 9
(f 3) 7      ⤳ 10
```

Enfin on peut écrire une fonction qui prend une autre fonction en argument :

```
let g h = h 2 + h 3
g mafonction      ⤳ 9
g (fun a -> a+3)      ⤳ 11
```

2.2.2 Polymorphisme : « Ne pas restreindre les type des fonctions inutilement »

Soit `f1` la fonction suivante qui prend un argument booléen et deux arguments entiers (notez au passage comment spécifier explicitement le type d'un argument et le type de retour de la fonction) :

```
let f1 (b:bool) (n:int) (m:int) : int = if b then n else m
```

Cette fonction retourne la valeur de son premier argument si `b` est vrai, et la valeur du second sinon.

```
f1 true 3 4      ⤳ 3
f1 false 3 4      ⤳ 4
```

Supposons maintenant qu'on désire écrire une fonction `f2` ayant le même comportement mais où `n` et `m` sont des `float` :

```
let f2 (b:bool) (n:float) (m:float) : float = if b then n else m
```

On voit bien que les deux fonctions sont identiques, aux informations de type près. Il semble naturel de n'écrire qu'une seule fonction dans laquelle *on ne restreint pas le type de* `n` *et* `m`. La seule restriction est que `n` et `m` doivent avoir le même type, et que ce type sera aussi celui du résultat.

Le polymorphisme des langages comme `ocaml` permet cela très facilement car d'une part il est permis de ne pas donner de type particulier à `n` et `m`, et d'autre part le compilateur calcule dans tous les cas lui-même les restrictions minimales, même lorsque les types sont beaucoup plus complexes.

```
let f (b:bool) n m = if b then n else m
```

On peut ensuite appliquer au type que l'on veut :

```
f true 3 5      ⤳ 3
f true 3.2 5.6      ⤳ 3.2
f false "titi" "toto"      ⤳ "toto"
f false "titi" 5  Ne compile pas
```

On voit que le compilateur permet l'utilisation de `f` sur plusieurs types différents, mais vérifie que les types des arguments sont cohérents entre eux. Pour être plus précis, les types inférés par `ocaml` pour `n`, `m` et la valeur de retour de `f` sont les suivants :

```
let f (b:bool) (n:'a) (m:'a) : 'a = if b then n else m
```

Autrement dit la fonction `f` est de type : `bool -> 'a -> 'a -> 'a`. Où `'a` est une variable de type qui peut être remplacée par n'importe quel autre type. À chaque utilisation de `f` tous les `'a` doivent être remplacés par le *même* type. On voit donc que la cohérence des types de `n`, `m` et du résultat est maintenue.

Pour plus de simplicité, dans la suite du cours les démonstrations se feront systématiquement sur des fonction non polymorphes.

2.3 Propriétés sur les programmes fonctionnels

Les programmes fonctionnels sont des fonctions, ce qui rend simple l'expression de propriétés sur ces programmes. Une propriété de fonction s'écrit en général de la manière suivante : pour une fonction de type $F_1 \to \cdots \to F_n \to E$ (on suppose donc les fonctions non polymorphes), on veut démontrer une propriété de la forme :

$$\forall x_1 \in F_1, \ldots, x_n \in F_n, Q(x_1, \ldots, x_n, (f\ x_1\ \ldots\ x_n))$$

pour toute propriété Q reliant les *paramètres* $(x_1\ \ldots\ x_n)$ de la fonctions à la *valeur de retour* $(f\ x_1\ \ldots\ x_n)$ de cette fonction. Soit par exemple le programme fonctionnel suivant :

Programme 2.1 – premier programme

```
let f x = 1  1
```

Il est possible de démontrer la propriété suivante : $\forall x \in \mathbb{N}, f\ x = 1$.

Il est souvent nécessaire de restreindre le domaine sur lequel la fonction vérifie Q. Pour cela on décompose Q en deux : la *pré-condition* P sur les arguments, et la *post-condition* Q qui relie le résultat aux paramètres.

$$\forall x_1 \in F_1, \ldots, x_n \in F_n, P(x_1, \ldots, x_n) \Rightarrow Q(x_1, \ldots, x_n, (f\ x_1\ \ldots\ x_n))$$

Par exemple, si on veut exprimer qu'une fonction retourne une valeur plus grande ou égale à son argument *quand ce dernier est lui-même plus grand ou égal à* 1, on écrira la propriété de la façon suivante : $\forall x, x \geq 1 \Rightarrow h\ x \geq x$.

2.4 Valeurs définies, valeurs indéfinies

Il semble aisé de démontrer que $\forall x \in \mathbb{N}, f\ x = 1$. C'est en effet trivial, mais il faut néanmoins relativiser cette propriété. En effet que se passe-t-il par exemple dans l'appel suivant ?

```
let rec g x = g (x+1);;
f (g 1);;
```

L'appel à `g` ne termine pas, donc `f (g 1)` ne rend pas de résultat, alors que le type de `g 1` est bien `int`. Donc la propriété $\forall x \in \mathbb{N}, fx = 1$ semble fausse. La raison de ce paradoxe apparent est que être de type `int` et appartenir à $\mathbb{N}$ ne signifient pas la même chose. En particulier il existe des valeurs de type `int` qui sont *indéfinies*, comme `g 1` ci-dessus. Si on suppose que x est une *valeur définie*, la propriété $\forall x \in \mathbb{N}, fx = 1$ est vraie. Dans la suite, quand on écrira $\forall x \in E, \ldots$ cela signifiera que x est une valeur *définie* appartenant à E.

2.5 Équation de programme

Pour démontrer un programme écrit en CAML, la première chose à faire est d'écrire son « équation ». Par exemple l'équation du programme 2.1 ci-dessus est la suivante :

$$f\ x = 1 \tag{2.1}$$

Prenons un programme plus complexe :

Programme 2.2 – une conditionnelle

```
let rec h n = if n<=0 then 0 else n + h (n-1)
```

l'équation est la suivante :

$$\mathtt{h}\ n = \begin{cases} 0 & \text{si } n \leq 0, \\ n + \mathtt{h}\ (n-1) & \text{sinon.} \end{cases} \tag{2.2}$$

Remarquez que l'équation d'une fonction récursive contient des appels à la fonction des deux côtés de l'équation.

Remarquez également qu'on ne peut pas vraiment démontrer cette équation, en effet il faudrait pour cela démontrer entre autre que l'expression CAML `n<=0` est bien compilé vers le test $n \leq 0$, mais aussi que le `if` est bien compilé vers une structure de test etc. Il existe des recherches visant à démontrer qu'un *compilateur* est correct, c'est-à-dire qu'il produit un code assembleur ayant bien la même *sémantique* que le code source. Nous ne nous intéressons pas ici à ce sujet et nous admettrons dorénavant que le compilateur est correct. L'équation d'un programme fonctionnel est donc la traduction « mot à mot » du code source du programme en écriture mathématique. Nous ne nous soucierons pas non plus des éventuels dépassements de capacité de la machine sur laquelle s'exécutent nos programmes.

Nous utiliserons l'équation (2.2) plus tard dans la section 5 lorsque nous utiliserons les démonstration par récurrence et induction. Pour la propriété sur la fonction triviale `f` : $\forall x \in \mathbb{N}, f x = 1$, la preuve de cette propriété est immédiate car il s'agit de l'équation de la fonction.

2.6 Premières preuves de programmes

Lorsque les programmes sont suffisamment simples, en particulier si ils ne sont pas récursifs et n'utilisent pas de types autres que les types de base, les démonstrations ne nécessitent pas d'outils mathématique particulier, comme le montre cet exemple :

Exemple 2.6.1. *Soit la fonction caml suivante :*

```
let g x = if (x/2)*2 = x then x else x + 1
```

On désire montrer la propriété suivante : $\forall x \in \mathbb{N}, g\ x$ est paire. Pour cela, on commence par exhiber l'équation de la fonction g :

$$g\ x = \begin{cases} x & si\ (x/2)*2 = x \\ x+1 & sinon \end{cases} \tag{2.3}$$

Or $(x/2)*2 = x$ est vrai ssi x est paire, donc finalement on a l'équation suivante :

$$g\ x = \begin{cases} x & si\ x\ est\ paire \\ x+1 & sinon \end{cases}$$

La démonstration se fait par cas :

- *Soit x est paire, et alors* $f\ x = x$ *est paire aussi. OK.*
- *Soit x est impaire, et alors* $f\ x = x + 1$ *est paire. OK.*

En revanche l'étude de programmes plus complexes nécessite un outil mathématique simple : la récurrence et ses généralisations à l'induction et aux ordres bien fondés. Les chapitres suivants (3 et 4) définissent ces notions.

Exercice 2.1 *Démontrez que le programme suivant retourne toujours un nombre positif ou nul :*

```
let pos x = if x >= 0 then x else (-x)
```

□

Chapitre 3

Ensembles, relations, fonctions, ordres

Traditionnellement l'objet mathématique de base est l'*ensemble*. Par définition un ensemble est la *collection* de ces *éléments*. Le but de ce chapitre est d'effectuer un rapide survol de la notion d'ensemble et des notions immédiatement associées : les relations entre ensembles et les fonction des ensemble vers les ensembles. On commence par un rapide rappel des notation logiques usuelles.

3.1 Notations logiques

On note les opérations logiques selon le tableau suivant :

$p \wedge q$	p et q
$p \vee q$	p ou q
$\neg p$	non p
$a \rightarrow b$	a implique b
$\forall x, p$	pour tout x, p
$\exists x, p$	il existe un x tel que p

3.2 Ensembles

Définition 3.2.1 (Ensemble)**.** *Soit E un ensemble, on note $e \in E$ la propriété ≪e appartient à E≫, et $e \notin E$ la propriété ≪e n'appartient pas à E≫.*

Il existe différente façon de définir un ensemble à partir de ces éléments :

- *Par extension*, c'est-à-dire en donnant la liste complète de ces éléments. On note alors l'ensemble entre accolade. Par exemple : $E = \{1,2,3\}$.
- *Par intention*, c'est-à-dire à partir d'un autre ensemble U en se restreignant aux éléments vérifiant une propriété P. On note également entre accolade. Par exemple : $F = \{x \in E | x < 3\}$.
- *Par induction*, c'est-à-dire à partir d'un ensemble d'éléments de base et de règles permettant de construire les autres éléments. Cette méthode de définition des ensembles fera l'objet d'un chapitre plus loin dans ce document.
- *Par des opérations ensemblistes*, comme l'union ou l'intersection, permettant de construire des ensembles à partir d'autres ensembles. Nous définissons certaines de ces opérations ci-dessous.

Définition 3.2.2 (Union)**.** *$A \cup B$ est l'ensemble contenant les éléments de A et les éléments de B. Plus formellement :*

$$A \cup B = \{x | x \in A \vee x \in B\}$$

Définition 3.2.3 (Intersection)**.** *$A \cap B$ est l'ensemble contenant les éléments appartenant à la fois à A et à B.*

$$A \cap B = \{x | x \in A \wedge x \in B\}$$

Définition 3.2.4 (Soustraction)**.** *$A \backslash B$ est l'ensemble contenant les éléments de A qui ne sont pas dans B.*

$$A \backslash B = \{x | x \in A \wedge x \notin B\}$$

Définition 3.2.5 (Produit cartésien)**.** *$A \times B$ est l'ensemble des* paires d'éléments *de A et B.*

$$A \times B = \{(x,y) | x \in A \wedge y \in B\}$$

On généralise le produit cartésien à n ensembles : $A_1 \times \ldots \times A_n = \{(x_1, \ldots, x_n) | \forall 1 \leq i \leq n, x_i \in A_i\}$. On note $A^n = \underbrace{A \times \ldots \times A}_{n\ fois}$.

3.3 Relations

Une relation est un ensemble de couple (a,b) où $a \in A$ et $b \in B$. Plus formellement :

Définition 3.3.1. *Soit A et B deux ensembles, une relation R de A dans B est un sous-ensemble de $A \times B$. Lorsque $(a,b) \in A \times B$, on note $R(a,b)$ et parfois aRb.*

Exemple : `est_double`$(4,2)$, $x = y\ldots$

Définition 3.3.2 (Réflexivité)**.** *Une relation R est* réflexive *si pour tout a, $R(a,a)$.*

Définition 3.3.3 (Irréflexivité)**.** *Une relation R est* irréflexive *si pour tout a, $\neg R(a,a)$.*

Définition 3.3.4 (Symétrie)**.** *Une relation R est* symétrique *si pour tout a, b, si $R(a,b)$ alors $R(b,a)$.*

Définition 3.3.5 (Antisymétrie)**.** *Une relation R est* antisymétrique *si pour tout a, b, si $R(a,b)$ et $R(b,a)$ alors $a = b$..*

Définition 3.3.6 (Transitivité)**.** *Une relation R est* transitive *si pour tout a, b, c, si $R(a,b)$ et $R(b,c)$ alors $R(a,c)$.*

Définition 3.3.7 (irreflexivité)**.** *Une relation R est* irreflexive *si pour tout a, $R(a,a)$ est faux. Autrement dit si $\forall a, \neg R(a,a)$.*

3.4 Fonctions

Une fonction de A dans B est une relation qui a une propriété supplémentaire : à chaque $a \in A$ correspond au plus un $b \in B$. Plus formellement :

Définition 3.4.1 (Fonction)**.** *Une fonction f de A vers B (on note $f : A \to B$) est une relation de A dans B dans laquelle pour tout $a \in A$, il existe au plus un $b \in B$ tel que $(a,b) \in f$. On note $f(a) = b$ la propriété $(a,b) \in f$.*

Définition 3.4.2 (Fonction totale ou application)**.** *Une fonction $f : A \to B$ est* totale *si pour tout $a \in A$, il existe un $b \in B$ tel que $f(a) = b$. Dans le cas contraire elle est dite* partielle.

Définition 3.4.3 (Injection)**.** *Une fonction* $f : A \to B$ *est une* injection *si pour tout* a *et* $a' \in A$*, si* $f(a) = f(a')$ *alors* $a = a'$*. On parle également de* fonction injective.

Définition 3.4.4 (Surjection)**.** *Une fonction* $f : A \to B$ *est une* surjection *si pour tout* $b \in B$ *il existe un* $a \in A$ *tel que* $f(a) = b$*. On parle également de* fonction surjective.

Définition 3.4.5 (Bijection)**.** *Une fonction est une* bijection *si elle est totale, bijective et surjective. On parle également de* fonction bijective.

Notez que ces propriétés dépendent des ensembles A et B. Par exemple une fonction $f : A \to B$ peut être totale alors que la "même" fonction $f : A' \to B'$ est partielle. En toute rigueur il faut toujours citer les ensembles A et B lorsqu'on énonce une propriété de ce type. En pratique on omettra cette précision si elle peut être déduite du contexte sans ambiguïté.

3.5 Ordres

3.5.1 Définitions

Une relation d'ordre, souvent notée $\leq$, est un type particulier de relation :

Définition 3.5.1 (Relation d'ordre (large))**.** *Une relation d'ordre sur un ensemble* E *est une relation binaire sur* E *réflexive, transitive et antisymétrique.*

Définition 3.5.2 (Relation d'ordre stricte)**.** *Une relation d'ordre strict est une relation binaire irréflexive, et transitive.*

Définition 3.5.3 (Relation d'ordre totale)**.** *Une relation d'ordre est* totale *ssi* $\forall x, y \in E, x \leq y \vee y \leq x$*. Elle est* partielle *sinon.*

Propriété 3.5.4. *Une relation d'ordre stricte* $<$ *est fortement antisymétrique, c'est-à-dire* $\forall x, y \in E,\ x < y \to \neg(y < x)$.

Notez qu'une relation d'ordre totale est forcément réflexive, donc un ordre strict ne peut pas être total. Par abus de langage on dira cependant qu'un ordre strict $>$ est total si son extension $(< \cup =)$ est un ordre totale.

Une relation d'ordre stricte peut-être construite à partir d'une relation d'ordre large en restreignant la relation aux couples d'éléments *distincts* et *non ordonnés pour l'ordre large dans l'autre sens*.

Propriété 3.5.5. *Soit* $\leq$ *une relation d'ordre large. On peut définir la relation d'ordre stricte associée à* $\leq$*, notée* $<$*, comme suit* $\forall x, y \in E,\ x < y \leftrightarrow x \leq y \wedge x \neq y \wedge \neg x \geq y$*. Si* $\leq$ *est un ordre total, alors la condition* $x \neq y$ *est inutile.*

Inversement :

Propriété 3.5.6. *Soit* $<$ *une relation d'ordre stricte. On peut définir la relation d'ordre large associée à* $<$*, notée* $\leq$ *comme suit* $\forall x, y \in E,\ x \leq y \leftrightarrow x < y \vee x = y$.

3.5.2 Ordres bien fondés

Un ordre est bien fondé si il n'existe pas de suite infinie décroissante. Cette notion est centrale pour les démonstrations de *terminaison* de programme.

Définition 3.5.7 (Ordre bien-fondé)**.** *Soit $<$ une ordre. On dit que $<$ est bien fondé si il n'existe pas de suite infinie décroissante (x_n) d'éléments de E (c'est-à-dire telle qu'on ait $\forall n,\ x_{n+1} < x_n$).*

Remarquez qu'un ordre bien-fondé est nécessairement strict car si $a \leq a$, alors la suite $(x_n = a)$ est infinie et décroissante.

Exemple 3.5.8. *L'ordre strict standard sur $\mathbb{N}$ est bien fondé.*

Exemple 3.5.9. *L'ordre strict standard sur $\mathbb{Z}$ n'est pas bien fondé. En effet, la suite suivante est infinie décroissante : $(x_0 = 0, x_{n+1} = -x_n)$.*

Exemple 3.5.10. *L'ordre strict standard sur $\mathbb{Z}^+$ est bien fondé.*

Exemple 3.5.11. *L'ordre strict standard sur $\mathbb{R}$ n'est pas bien fondé. En effet, la suite suivante est infinie décroissante : $(x_0 = 0, x_{n+1} = -x_n)$.*

Exemple 3.5.12. *L'ordre standard sur $\mathbb{R}^+$ n'est pas bien fondé. En effet, la suite suivante est infinie décroissante : $(x_0 = 1, x_{n+1} = \frac{x_n}{2})$.*

Théorème 3.5.13 (Ordre bien fondé par image inverse)**.** *Soit E et F deux ensembles et $f : E \to F$ une fonction de E vers F. Soient $>_E$ et $>_F$ deux ordres tels que $\forall x,y \in E, x >_E y$ ssi $f(x) >_F f(y)$. Si $>_F$ est bien fondé alors $>_E$ est bien fondé. (Notez la direction de l'implication, qui va de F vers E).*

Définition 3.5.14 (Composition lexicographique d'ordre)**.** *Soit $<_1$ et $<_2$ deux relations d'ordre. On appelle* composition lexicographique *de $<_1$ et $<_2$, noté $<_{Lex(<_1,<_2)}$ la relation suivante :*

$$x <_{Lex(<_1,<_2)} y \ ssi \begin{cases} x <_1 y \ \vee & (lex_1) \\ x = y \wedge x <_2 y & (lex_2) \end{cases}$$

Théorème 3.5.15. *La composition lexicographique de deux ordres bien-fondés est bien-fondée.*

Démonstration. Nous utilisons une démonstration par l'absurde : Soit $<_1$ et $<_2$ deux relations d'ordre bien fondées et $<$ sa composition lexicographique ($<_{Lex(<_1,<_2)}$). Supposons que $<$ n'est pas bien-fondée. Il existe alors une suite infinie décroissante (x_n) sur cet ordre :

$$x_0 > x_1 > x_2 > \ldots > x_i > \ldots \infty$$

On distingue 2 cas :

1. Il existe un k à partir duquel toutes les étapes sont de type (lex_2). Alors cela signifie que la suite $(x_{n \geq k})$ est infinie décroissante pour $<_2$. Or $<_2$ est bien-fondée donc c'est une contradiction.
2. Un tel k n'existe pas. Alors (x_n) contient une infinité d'étapes de type (lex_1). Comme $<_1$ est bien-fondé, la suite (x_n) contient une alternance infinie d'étapes (lex_1) et d'étapes (lex_2) :

   $$x_0 \quad >_2 \ldots >_2 \quad x_{i_1} \quad >_1 \quad x_{i_1+1} \quad >_2 \quad x_{i_1+2} \quad >_2 \ldots >_2 \quad x_{i_2} \quad >_1 \quad x_{i_2+1} \quad \ldots \infty$$

   Or les x_i sont de la forme (a,b) et les étapes de type (lex_2) ne modifient pas les a_i, donc la suite est de la forme :

   $$(a_0,b_0) >_2 \ldots >_2 (\mathbf{a_0}, b_{i_1}) >_1 (\mathbf{a_{i_1+1}}, b_{i_1+1}) >_2 (a_{i_1+1}, b_{i_1+2}) >_2 \ldots >_2 (\mathbf{a_{i_1+1}}, b_{i_2}) >_1 (\mathbf{a_{i_2+1}}, b_{i_2+1}) \ldots \infty$$

Donc on peut construire la suite infinie suivante, composée des a_i et décroissante pour $<_1$:

$$\mathbf{a_0} >_1 \mathbf{a_{i_1+1}} >_1 \mathbf{a_{i_2+1}} \ \ldots \ \infty$$

Ce qui est une contradiction car $<_1$ est bien-fondé.

□

Exemple 3.5.16. *L'ordre $<_{Lex(<_{\mathbb{N}},<_{\mathbb{Z}^+})}$ est bien fondé car $<_{\mathbb{N}}$ et $<_{\mathbb{Z}}{}^+$ sont bien fondés.*

Exemple 3.5.17. *Soit* `f` *la fonction suivante sur $\mathbb{N}$:*

```
let f n = if n=0 then 0 else n + f (n-1)
```

l'appel récursif `f (n-1)` *se fait sur* `n-1`*, qui est plus petit que l'argument* `n` *pour l'ordre bien fondé $<_{\mathbb{N}}$. Donc il ne peut pas y avoir une suite infinie d'appels récursifs car cela signifierait qu'il existe une suite infinie décroissante pour $<_{\mathbb{N}}$. Donc la fonction* `f` *termine sur $\mathbb{N}$.*

En revanche, si `f` *est définie sur $\mathbb{Z}$, il n'existe pas d'ordre bien fondé $<$ sur $\mathbb{Z}$ tel que $n-1<n$. Donc on ne peut pas démontrer que la fonction termine (et de fait elle ne termine pas si on lui passe un argument négatif).*

Exercice 3.2 *Soit* `ack` *la fonction suivante :*

```
let rec ack x y =
  if x = 0 then y + 1
  else if y = 0 then (ack (x-1) 1)
  else (ack (x-1) (ack x (y-1)))
```

Cette fonction termine pour tout $x,y \in \mathbb{N}$, démontrez le. □

Exercice 3.3 *Démontrez que la fonction f suivante termine.*

$$\begin{aligned} f(x,0) &= x &(3.1)\\ f(0,s(y)) &= f(s(y)*10^{1000}\ ,\ y) &(3.2)\\ f(s(x),s(y)) &= 10^{1000}*f(x,s(y)) &(3.3) \end{aligned}$$

□

Théorème 3.5.18. *Soit $>_E$ et $>_F$ deux ordres sur les ensembles E et F. Soit $f : E \rightarrow F$ telle que $\forall e,e' \in E, e >_E e' \rightarrow f(e) >_F f(e')$. Alors $>_E$ est bien fondé si $>_F$ l'est.*

Chapitre 4

Récurrence et induction

4.1 Récurrence ≪faible≫

La récurrence classique, dite faible, est celle qu'on voit à l'école, elle permet de démontrer une propriété de la forme $\forall n \in \mathbb{N}, P(n)$ en montrant qu'elle est vraie pour 0 et qu'elle est *héréditaire*, c'est-à-dire que si $P(n)$ est vraie pour un n donné (éventuellement $n = 0$), alors elle est aussi vraie pour $n+1$. Plus formellement, on exprime le princie de raisonnement par récurrence de la façon suivante :

$$\boxed{\forall P, \left(\begin{array}{l} P(0) \ \wedge \\ \forall n \in \mathbb{N}, P(n) \to P(n+1) \end{array} \right) \to \forall n \in \mathbb{N}, P(n)} \tag{4.1}$$

$P(0)$ est appelé le *cas de base* et $\forall n \in \mathbb{N}, P(n) \to P(n+1)$ est appelé le cas de récurrence (faible).

4.2 Récurrence ≪forte≫

La récurrence forte est en fait équivalente à la version faible, mais s'avère plus agréable dans certains cas. la notion d'hérédité y est différente : si on suppose $P(k)$ est vraie pour tout k strictement plus petit qu'un n donné (éventuellement $n = 0$), alors elle est vraie pour n. Plus formellement :

$$\boxed{\forall P, \left(\ \forall n \in \mathbb{N}, \left(\ \left(\ \forall k < n, P(k) \ \right) \to P(n) \ \right) \ \right) \to \forall n \in \mathbb{N}, P(n)} \tag{4.2}$$

Remarquez que le cas de base a disparu. En fait il est contenu dans l'autre cas (le cas de récurrence (forte)). Nous montrons ce résultat maintenant :

Propriété 4.2.1. *Si la propriété* $\left(\ \forall n \in \mathbb{N}, ((\forall k < n, P(k)) \to P(n)) \ \right)$ *est vraie, alors* $P(0)$ *aussi.*

Démonstration. Si cette propriété est vraie alors en particulier elle est vraie pour $n = 0$:

$$(\forall k < 0, P(k)) \to P(0).$$

Comme il n'y a pas de $k < 0$ dans $\mathbb{N}$, $\forall k < 0, P(k)$ est équivalent à *vrai*, on a donc :

$$vrai \to P(0).$$

qui est équivalent à $P(0)$. □

4.3 Induction

Le raisonnement par induction est une généralisation de la récurrence faible à d'autres ensemble que $\mathbb{N}$. Les ensembles sur lesquels s'applique se principe (ce « schéma ») de raisonnement sont les *ensembles définis par induction*.

4.3.1 Ensemble défini par induction - intuition

La définition d'ensemble E par induction comporte 3 parties :

- La *base*, qui est un ensemble d'éléments appartenant à E par définition.
- L'*induction*, qui est un ensemble de règles (d'*opérateurs*) permettant de construire (récursivement) de nouveaux éléments de E à partir d'éléments de E. Donc en partant de la base, puis par itération.
- La *fermeture* qui signifie que E est l'ensemble de tous les éléments que l'on peut construire à partir de la base en appliquant les opérateurs du point précédent. La fermeture signifie également que E *ne contient aucun autre éléments*.

Par exemple voici une définition de l'ensemble $\mathbb{N}$:

Exemple 4.3.1 (Définition inductive de $\mathbb{N}$)**.**

- Base : $B = \{0\}$ *;*
- Induction : $\Omega = \{S\}$ *où S est d'arité* 1 *et prend son argument dans* $\mathbb{N}$*(noté* $S : \mathbb{N} \to \mathbb{N}$*) ;*
- fermeture : $\mathbb{N}$ *est la fermeture de B par* Ω*.*

On peut faire plusieurs remarques :

- On voit que les termes de la forme $0,\ S(0),\ S(S(0))\ldots$ appartiennent à $\mathbb{N}$.
- $\mathbb{N}$ apparaît dans sa propre définition (ici dans le type de S). Il s'agit d'une définition *récursive* : en fait dans la définition $\mathbb{N}$ représente n'importe quel ensemble E vérifiant $B \subset E$ et $\forall x \in E, S(x) \in E$. C'est l'opération de fermeture qui permet de préciser ensuite que parmi les E possibles, on en prend un en particulier (voir section suivante).
- L'opérateur S est un *constructeur*, pas une fonction. Ce qui signifie que $S(0)$ est un élément de $\mathbb{N}$, et qu'il ne se calcule pas vers une autre valeur (il n'est pas égal à 1, qui n'est d'ailleurs pas défini ici). Cette notion de constructeur est bien évidemment à rapprocher de celle de constructeur de type dans les langages fonctionnels. En fait dans les langages fonctionnels, on appelle constructeurs les éléments de la base et les opérateurs. Par exemple on peut définir le type `nat` en `Caml` comme suit :

```
type nat =
 | Zero
 | S of nat;;
```

Cependant on peut parfois associer un *calcul* aux opérateurs dans une définition inductive.

Suivant cette dernière remarque, pour $\mathbb{N}$ on peut par exemple remplacer S par la fonction *succ*, définie par exemple sur $\mathbb{R}$, qui ajoute 1 à son argument. Dans ce cas $\mathbb{N}$ est défini comme un sous-ensemble de $\mathbb{R}$:

Exemple 4.3.2 (Définition inductive de $\mathbb{N}$ à l'aide d'un opérateur fonction)**.**

- Base *:* $B = \{0\} \subset \mathbb{R}$ *;*
- Induction *:* $\Omega = \{succ\}$ *où* $succ : \mathbb{R} \to \mathbb{R}$ *telle que* $succ : n \mapsto n+1$
- Fermeture *:* $\mathbb{N}$ *est la fermeture de B par* Ω*.*

On voit que les réels $0,\ 1,\ 2,\ldots$ appartiennent à $\mathbb{N}$.

On retiendra donc que lorsqu'on définit un ensemble par induction, si on associe des fonctions aux opérateurs, alors l'ensemble est défini comme un *sous-ensemble* d'un ensemble défini précédemment ($\mathbb{N}$

comme sous-ensemble de $\mathbb{R}$). En revanche lorsque les opérateurs sont des constructeurs, les éléments de l'ensemble défini inductivement sont *nouveaux* (S(0) est une nouvelle valeur, égale seulement à elle-même).

4.3.2 Ensemble défini par induction - définition

Nous pouvons maintenant définir la notion de définition inductive d'un ensemble E. Avant cela il est nécessaire de préciser la notion d'opérateur et de *stabilité* d'un ensemble par rapport à un opérateur :

Définition 4.3.3 (opérateur, arité, application définie)**.**
- *Un* opérateur f *est un symbole muni d'une signature* $F_1 \times \ldots \times F_n$*, ou les* F_i *sont des ensembles quelconques.*
- *On appelle* n *l'arité de* f*.*
- *On dit que l'application de* f *à des arguments* $x_1, \ldots, x_n$*, notée* $f(x_1, \ldots, x_m)$ *est* défini *si* $x_1 \in F_1, \ldots, x_n \in F_n$*.*

Reprenons l'exemple 4.3.1, on voit que S a bien pour signature $F_1 \times ... \times F_n$ avec $n = 1$ et $F = \mathbb{N}$.

La stabilité d'un ensemble E par rapport à un opérateur permet de définir formellemement la notion de fermeture. La définition en toute généralité étant complexe, nous donnons 3 définitions successives de plus en plus générales : la stabilité pour un opérateur d'arité 1 dont l'argument appartient à E, puis pour un opérateur d'arité quelconque prenant tous ses arguments dans E, puis pour un opérateur d'arité quelconque ne prenant pas tous ses arguments dans E.

Définition 4.3.4 (Stabilité d'un ensemble par rapport à un opérateur d'arité 1)**.**
Un ensemble E *est dit* stable par rapport à l'opérateur f *de signature* E *si* $\forall x \in E, f(x) \in E$*. On écrit aussi* $f(E) \subseteq E$*.*

On généralise à $f : E^n \to U$:

Définition 4.3.5 (Stabilité d'un ensemble par rapport à un opérateur sur E^n)**.**
Un ensemble E *est dit* stable par rapport à l'opérateur f *de signature* E^n *si* $\forall e_1 \ldots e_n \in E, f(e_1, \ldots, e_n) \in E$*. On écrit aussi* $f(E^n) \subseteq E$*.*

On généralise à la signature $F_1 \times \ldots \times F_m \times E^n$, qui couvre tous les opérateurs possibles modulo permutation des arguments :

Définition 4.3.6 (Stabilité d'un ensemble par rapport à un opérateur)**.**
Étant donné les ensembles $F_1 \ldots F_m$*, un ensemble* E *est dit* stable *par rapport à un opérateur* f *de signature* $F_1 \times \ldots \times F_m \times E^n$ et par rapport aux ensembles $F_1 \ldots F_m$*, si*

$$\forall x_1 \in F_1, \ldots, \forall x_m \in F_m, \forall e_1, \ldots, e_n \in E,\ f(x_1, \ldots, x_m, e_1, \ldots, e_n) \in E.$$

On écrit aussi $f(F_1, \ldots, F_n, E^n) \subseteq E$*.*

Définition 4.3.7 (Définition inductive)**.**
Soient :
- B *un ensemble appelé* base *;*
- Ω *un ensemble d'opérateurs* $\{\overrightarrow{f_i}\}$*.*

On appelle fermeture inductive de B par Ω *ou* ensemble inductivement défini par $\langle B, \Omega \rangle$ *l'ensemble* E *défini par le schéma suivant :*

E *est* la plus petite partie de U *telle que :*

1. *$B \subseteq E$;*
2. *E est stable par rapport aux opérateurs de Ω.*

On voit que la définition de E fait appel à E lui-même dans le point 2 ci-dessus. Pour cette raison, la fermeture inductive est parfois appelée *fermeture récursive*.

Exemple 4.3.8 (Définition inductive des listes d'entiers $\mathscr{L}_{\mathbb{N}}$)**.**
- *$B = \{Nil\}$;*
- *$\Omega = \{Cons\}$ où Cons a pour signature $\mathbb{N} \times \mathscr{L}_{\mathbb{N}}$.*

Exemples d'élément de $\mathscr{L}$: Nil, $Cons(S(0), Nil)$, $Cons(0, Cons(S(0), Nil))$... On notera $Cons(x,l)$ parfois dans la suite $x :: l$.

Exemple 4.3.9 (Définition inductive des arbres binaires d'entiers $\mathscr{AB}_{\mathbb{N}}$)**.**
- *$B = \{Vide\}$;*
- *$\Omega = \{Noeud\}$ où Noeud a pour signature $\mathscr{AB}_{\mathbb{N}} \times \mathbb{N} \times \mathscr{AB}_{\mathbb{N}}$.*

Exemples d'élément de $\mathscr{AB}$: $Vide$, $Noeud(Vide, S(0), Vide)$, $Noeud(Vide, S(0), Noeud(Vide, 0, Vide))$...

4.3.3 Raisonnement par induction

Le raisonnement d'induction permet de montrer des propriétés de le forme $\forall x \in E, P(x)$ pour un ensemble E défini par induction. Par exemple sur l'ensemble $\mathbb{N}$ défini dans l'exemple 4.3.1, on retrouve un principe d'induction semblable au principe de récurrence faible :

$$\forall P, \left(\begin{array}{l} P(0) \wedge \\ \forall x \in \mathbb{N}, P(x) \rightarrow P(S(x)) \end{array} \right) \rightarrow \forall x \in \mathbb{N}, P(x)$$

Le principe est le suivant : pour prouver qu'une propriété est vraie pour tout élément d'un ensemble E défini par induction, il faut démontrer :
- Que P est vrai pour chaque élément de la base B_E (ici : $P(0)$) ;
- Les opérateurs préservent P (ici S préserve P : $\forall x \in \mathbb{N}, P(x) \rightarrow P(S(x))$). On dit aussi que P est une propriété *héréditaire* pour les opérateurs.

Plus généralement voici la forme du principe de raisonnement par induction sur un ensemble quelconque défini par induction :

Théorème 4.3.10 (Principe d'induction structurelle)**.**
Soit $E = < B, \Omega >$ un ensemble défini par induction et P une propriété. Pour montrer $\forall x \in E, P(x)$ il suffit de démontrer :

1. *base : $\forall x \in B, P(x)$;*
2. *règles : pour chaque opérateur $f \in \Omega$ de signature $F_1 \times \ldots \times F_m \times E^n$, f préserve P, c'est-à-dire :*

$$\forall e_1, \ldots, e_n \in E, P(e_1) \wedge \ldots \wedge P(e_n) \rightarrow \overrightarrow{\forall x_i \in F_i},\ P(f(x_1, \ldots, x_m, e_1, \ldots, e_n)).$$

Exemple 4.3.11 (Principe de raisonnement par induction sur les listes d'entiers)**.** *D'après Le théorème 4.3.10, le principe d'induction sur les listes d'entiers $\mathscr{L}$ de l'exemple 4.3.8 est le suivant :*

$$\forall P, \left(\begin{array}{l} P(Nil) \wedge \\ \forall l \in \mathscr{L}_{\mathbb{N}}, P(l) \rightarrow \forall n \in \mathbb{N}, P(Cons(n,l)) \end{array} \right) \rightarrow \forall l \in \mathscr{L}_{\mathbb{N}}, P(l)$$

Exemple 4.3.12 (Principe de raisonnement par induction sur les arbres binaires d'entiers)**.** *D'après Le théorème 4.3.10, le principe d'induction sur les arbres binaires d'entiers $\mathscr{AB}$ de l'exemple 4.3.9 est le suivant :*

$$\forall P, \left(\begin{array}{l} P(Vide) \wedge \\ \forall a_1, a_2 \in \mathscr{AB}_{\mathbb{N}}, P(a_1) \wedge P(a_2) \rightarrow \forall n \in \mathbb{N}, P(Noeud(a_1, n, a_2)) \end{array} \right) \rightarrow \forall a \in \mathscr{AB}_{\mathbb{N}}, P(a)$$

Avant d'utiliser ces principes de raisonnements, il est nécessaire de pouvoir exprimer des propriétés sur les ensembles inductif. Ceci nécessite de pouvoir définir des *fonctions* sur les type inductifs, c'est-à-dire des *programmes* fonctionnels. Nous utiliserons ici la syntaxe des langage fonctionnels pour exprimer ces fonctions. Nous ne détaillerons pas les bases mathématiques associées à cette syntaxe, ceci n'étant pas l'objet de ce cours. Nous nous attacherons en revanche à démontrer des propriétés sur ces programmes.

4.3.4 Raisonnement par induction bien fondée

On aura noté que le principe d'induction correspond à la récurrence faible transposée aux ensembles définis par induction. Il nous reste à définir l'équivalent de la récurrence forte transposée aux mêmes ensembles. En fait l'induction ne se généralise pas uniquement sur les ensembles définis par induction mais sur *les ensembles munis d'un ordre bien fondé*. Sans entrer dans les détails ces deux notions sont en réalité assez proches.

Théorème 4.3.13 (Principe d'induction bien fondée)**.**
Soit E un ensemble et $<$ un ordre bien fondé total sur E. Soit P une propriété sur les éléments de E. Pour montrer $\forall x \in E, P(x)$ il suffit de démontrer que la propriété est héréditaire *pour $<$, plus précisément :*

$$\forall P, \left(\forall e \in E, \left(\left(\forall x < e,\ P(x) \right) \rightarrow P(e) \right) \right) \rightarrow \forall x \in E, P(x)$$

Ce principe se révèle bien adapté au raisonnement sur les fonctions récursives « non structurelles », c'est-à-dire dans lesquelles les appels récursifs se font sur des arguments plus petits *selon un ordre ne correspondant pas à l'ordre inductif.*

Exemple 4.3.14. *D'après le théorème 4.3.13 ci-dessus, on peut définir un principe d'induction sur les listes d'entiers naturels à partir de l'ordre suivant bien fondé sur les listes :*

$$l < l' \ \ ssi \ \ sum(l) <_{\mathbb{N}} sum(l').$$

où $sum(l)$ est la somme des éléments de la liste l. On obtient le principe suivant sur les listes d'entiers :

$$\forall P, \left(\forall l \in \mathscr{L}_{\mathbb{N}}, \left(\left(\forall l' \in \mathscr{L}_{\mathbb{N}},\ t.q.\ sum(l') < sum(l),\ P(l') \right) \rightarrow P(l) \right) \right) \rightarrow \forall l \in \mathscr{L}_{\mathbb{N}}, P(l)$$

Chapitre 5

Preuves de programmes fonctionnels

5.1 Exemple

Maintenant que nous avons les outils théoriques adéquats, nous allons *démontrer* des propriétés sur des programmes fonctionnels. Reprenons le programme 2.2 de la section 2.5 :

```
let f n = if n<=0 then 0 else n + f (n-1)
```

l'équation est la suivante :

$$\texttt{f}\ n = \begin{cases} 0 & \text{si } n \leq 0, \\ n + \texttt{f}(n-1) & \text{sinon.} \end{cases} \tag{5.1}$$

La première propriété que nous allons démontrer sur le programme `f` est que son résultat est toujours positif ou nul :

Propriété 5.1.1. $\forall n \in \mathbb{Z}, \texttt{f}\ n \geq 0$

Démonstration. Soit n un entier relatif quelconque, démontrons que $\texttt{f}\ n \geq 0$.
Pour démontrer cela nous procédons premièrement par cas :

- Soit n est négatif et d'après l'équation 5.1 : $\texttt{f}\ n = 0$ et la propriété est vraie ;
- Soit $n \in \mathbb{N}$ et on procède par *récurrence sur n* :
  Base : Par 5.1 : $\texttt{f}\ 0 = 0$ et la propriété est vraie ;
  Réc : Soit $i \in \mathbb{N}$, supposons que $\texttt{f}\ i \geq 0$, alors $i+1 > 0$ donc d'après l'équation 5.1, $\texttt{f}\ (i+1) = (i+1) + \texttt{f}(i+1-1) = (i+1) + \texttt{f}(i)$. Comme par hypothèse de récurrence $\texttt{f}\ i \geq 0$. $\texttt{f}\ (i+1)$ est donc positif. La propriété est donc vraie.
  Par récurrence, $\forall n \in \mathbb{N}, \texttt{f}\ n \geq 0$.
  Conclusion la propriété est vraie pour tout $n \in \mathbb{Z}$.

□

5.2 Exercices

Exercice 5.4 *Prouvez la propriété suivante :* $\forall n \in \mathbb{Z}, \texttt{f}\ n \leq \texttt{f}\ (n+1)$ □

Exercice 5.5 *Soit* `filtrez` *la fonction récursive suivante :*

```
let rec filtrez l =
  match l with
    | [] -> []
    | e::l' -> if e >= 0 then e:: filtrez l' else filtrez l'
```

1. *Démontrez la propriété suivante :* $\forall l, \forall x \in$ `filtrez` $l, x \geq 0$.
2. *Quelle(s) propriété(s) faut-il prouver pour spécifier complètement* `filtrez` *? Démontrez les.*

□

Exercice 5.6 *Soit* `filtren` *la fonction récursive suivante :*

```
let rec filtren l n =
  match l with
    | [] -> []
    | e::l' -> if e >= n then e:: filtren l' else filtren l'
```

Démontrez la propriété suivante : $\forall n, \forall l, \forall x \in$ `filtren` $l\ n, x \geq n$. □

Exercice 5.7 *Soit* `maxlist` *la fonction récursive suivante :*

```
let rec maxlist l =
  match l with
    | [] -> 0
    | e::l' ->
        let mx = maxlist l' in
        if mx>=e then mx else e
```

Démontrez la propriété suivante : $\forall l, \forall x \in l, x \leq$ `maxlist` $(x :: l)$. □

Exercice 5.8 *Soit la fonction f suivante définie par l'équation :*

$$f(x :: y :: l) = x :: f\ l \tag{5.2}$$

$$f(x :: Nil) = Nil \tag{5.3}$$

$$f(Nil) = Nil \tag{5.4}$$

1. *Écrivez un programme* `ocaml` *correspondant à cette équation.*
2. *Démontrez que pour toute liste l,* $|f(l)| \leq |l|/2$.

(rappel : $|l|$ *et* $|f(l)|$ *désignent les* longueurs *des listes l et* $f(l)$*).*

□

Exercice 5.9 *Soit les fonctions fst, snd, f, g et h définies par :*

$$fst(x,y) = x \quad (5.5)$$
$$snd(x,y) = y \quad (5.6)$$
$$g(Nil) = (Nil,Nil) \quad (5.7)$$
$$g((x,y) :: l) = ((x :: fst(g\ l))\ ,\ (y :: snd(g\ l))) \quad (5.8)$$
$$h(Nil,Nil) = Nil \quad (5.9)$$
$$h(x :: l\ ,\ y :: l') = (x,y) :: h(l,l') \quad (5.10)$$

1. *Démontrez que pour tout couple c,* $(fst(c), snd(c)) = c$ *(A)*
2. *Démontrez que pour toute liste de couples l,* $h(g\ l) = l$. *(B)*

□

Exercice 5.10 *Soit* `g` *la fonction suivante :*

```
let rec g l =
  match l with
    | [] -> []
    | n::l' -> (n+1) :: (g l')
```

Démontrez la propriété suivante : $\forall l, |g\ l| = |l|$.
(Rappel : $|g\ l|$ *et* $|l|$ *représentent respectivement la longueur des listes* `g` *l et l).* □

Exercice 5.11 *Soit* `div` *la fonction récursive suivante :*

```
let rec div a b = if a < b then 0 else 1 + div (a-b) b
```

Démontrez les propriétés suivantes :

1. $\forall a \in \mathbb{N}, \forall b > 0 \in \mathbb{N},\ 0 \leq (\texttt{div}\ a\ b) \times b \leq a$.
2. $\forall a \in \mathbb{N}, \forall b > 0 \in \mathbb{N},\ 0 \leq (\texttt{div}\ a\ b) \times (b+1) > a$.

□

Exercice 5.12 *Soit* `modulo` *la fonction suivante, qui retourne le reste de la division entière de ces deux arguments :*

```
let rec modulo a b = if (a<b) then a else (modulo (a-b) b);;
```

1. *Donnez l'équation de* `modulo`.
2. *Démontrez que* $\forall x \in \mathbb{N},\ (\forall y > 0,\ \exists q \geq 0,\ x = qy + (\texttt{modulo}\ x\ y))$.

□

On définit une version $\mathscr{AB}'_{\mathbb{N}}$ des arbres binaires différente de celle définie précédemment, où les feuilles contiennent également une valeur entière :

- $B = \{Feuille(n) | n \in \mathbb{N}\}$;
- $\Omega = \{Node\}$, où si $fg, fd \in \mathscr{AB}'_{\mathbb{N}}$ et $n \in \mathbb{N}$ alors $Node(fg,n,fd) \in \mathscr{AB}'_{\mathbb{N}}$.

Exercice 5.13 *Démontrez que la fonction g suivante termine.*

$$g(Feuille(n)) = n \quad (5.11)$$
$$g(Node(Feuille(n_1), n, fd)) = n + g(fd) \quad (5.12)$$
$$g(Node(fg, n, Feuille(n_2))) = n + g(fg) \quad (5.13)$$
$$g(Node(Node(fg_1, n_1, fd_1), n, Node(fg_2, n_2, fd_2))) = n + g(fg_1) + g(fg_2) \quad (5.14)$$

□

Exercice 5.14 *On travaille sur un petit langage de programmation imaginaire Prog, qui contient pour l'instant seulement l'affectation de variable (:=). On définit inductivement Prog comme l'ensemble des programmes suivants :*

- $B_{Prog} = \{:=\}$ *où si* $x \in \{a,b\}^*$ *et* $n \in \mathbb{N}$ *alors* $x := n \in Prog$.
- $\Omega_{Prog} = \{$`;` , `begin`...`end`$\}$ *où*
  - *Si* $i \in Prog$ *alors* `begin` i_1 `end` $\in Prog$
  - *Si* $i_1, i_2 \in Prog$ *alors* i_1 `;` $i_2 \in Prog$

Rappel : $\{a,b\}^*$ *désigne l'ensemble des mots sur l'alphabet* $\{a,b\}$ *(par exemple aab).*

1. *Montrez que les deux programmes suivants appartiennent à Prog :*

```
begin           begin
a := 21         aa := 2;
end;            ba := 3
b := 3          end
```

2. *Démontrez que les deux programmes suivants ne sont pas dans Prog :*

```
begin           begin
end             aa:=2
```

3. *Démontrez que tout programme appartenant à Prog, contient le même nombre de* `begin` *et de* `end`.
4. *On désire ajouter le* `if` *au langage. Pour cela il faut définir les* expressions conditionnelles. *Une expression conditionnelle est*
   - *Soit* `true`, *soit* `false`,
   - *Soit un test d'égalité entre deux variables, ex : aa = ab,*
   - *Soit un* `and` *entre deux expressions conditionnelles,*
   - *Soit un* `or` *entre deux expressions conditionnelles.*

   Définissez inductivement l'ensemble Cond des expressions conditionnelles.
5. *Redéfinissez inductivement l'ensemble Prog afin qu'il possède le* `if`. *Par exemple le programme suivant devra appartenir à Prog :*

```
begin
if a = b and bb = a then a := bb else a := b;
ab := aab
end
```

□

Exercice 5.15 *On considère les équations suivantes :*
- **let** *f(x, y)* = **if** *x = y* **then** *0* **else** *1 + f(x + 1, y + 2)*
- **let** *g(x, y)* = **if** *x < y* **then** *x* **else** *g(f(x, y), y)*
- **let** *h(x, y)* = **if** *y = 0* **then** *x* **else** *h(y, g(x, y))*

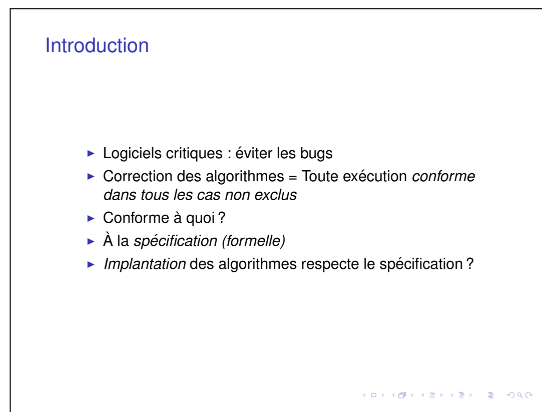
On déterminera les fonctions bien connues qui se cachent dans ces définitions. En se limitant à des arguments dans N, on précisera les domaines de définitions. On prouvera pour chacune, par induction bien fondée, qu'elle calcule bien la fonction bien connue qu'on aura reconnue. □

Deuxième partie

Preuve de programmes impératifs (`NFP209`)

Chapitre 6

Introduction



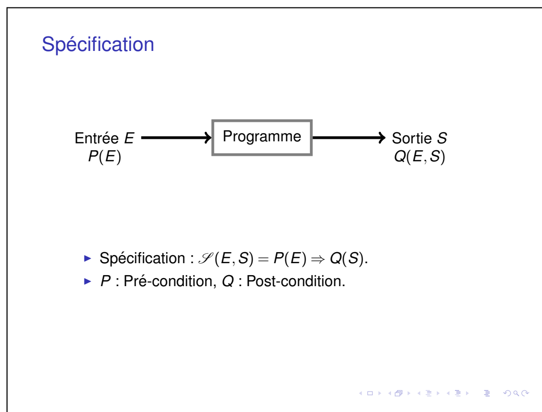
Ce cours cherche à poser les bases de l'activité qui consiste à s'assurer *formellement* de la correction d'un algorithme ou d'un programme. Informellement, on dira qu'un programme est correct si il effectue sans se tromper la tâche qui lui est confiée *dans tous les cas permis possibles*. Pour s'assurer de cela il est évidemment nécessaire de décrire précisément et sans ambiguïté la tâche en question et les cas permis. C'est ce qu'on appelle la *spécification* du programme. Si celle-ci est exprimée dans un langage mathématique non ambiguë, on qualifiera cette spécification de *formelle*. UML[1] ne fait par exemple pas partie des langages acceptables (du moins pas dans sa globalité) pour une spécification formelle car il n'est pas pourvue de sémantique rigoureuse, il est parfois qualifié de *semi-formel*. La spécification décrit à la fois les cas pour lesquels le programme doit se comporter correctement, et le comportement correct lui-même.

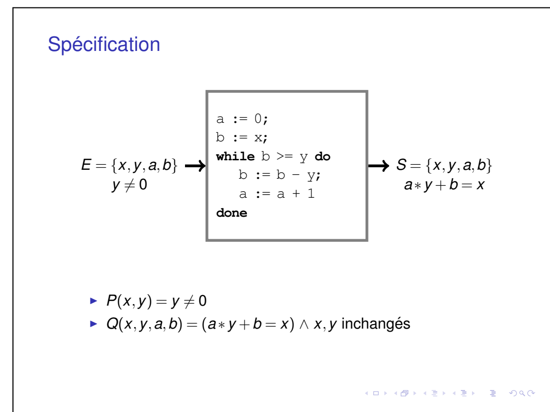
Il n'est pas ici question de *sûreté de fonctionnement*, qui s'intéresse plus spécifiquement au comportement des programmes lorsque des dysfonctionnements (physiques ou logiques) ont lieu.

Une spécification ne décrit pas nécessairement tous les détails de fonctionnement du programme, on parlerait alors plutôt d'une *implantation* [1]. On y décrit au contraire uniquement les propriétés que doit satisfaire le programme pour répondre au problème posé. En général on exprime ces propriétés comme une relation entre les *entrées* (ce que l'utilisateur tape au clavier, le contenu d'un fichier, la valeur des variables avant l'exécution, la valeur des arguments d'une fonction...) et les *sorties* du programme (ce qu'il y a à l'écran, la valeur des variable ou le contenu d'un fichier après l'exécution, la valeur de retour d'une fonction). Par exemple dans un programme calculant le quotient q et le reste r de la division entière de x et y (x, y, q et r étant des variables du programme), une des propriétés voulues à la fin du programme est la suivante : $q \times y + r = x$. La plupart du temps, les entrées seront les variables du programme au début du programme

1. On utilise à tord l'anglicisme *implémentation*.

et/ou les valeurs des arguments d'une fonction et les sorties seront les variables du programme à la fin de l'exécution du programme et/ou la valeur de retour d'une fonction.





Une fois la spécification formelle écrite et acceptée par une instance compétente (des spécialistes du domaine concerné, les clients du programmeur...), il s'agit de vérifier que le programme est correct *par rapport* à cette spécification. On utilise pour cela plusieurs techniques : la relecture, le test, la publication du code source etc. Nous nous intéressons à une méthode en particulier : la *démonstration*.

L'objet de ce cours est d'aborder des techniques permettant de démontrer mathématiquement qu'un programme respecte sa spécification. Les démonstrations de correction et de complétude de ces techniques (qui de toutes façons reposent sur la correction d'autres techniques et ainsi de suite) existent et se basent sur la *sémantique* du langage utilisé [2]. Nous verrons que ces techniques nécessitent de la part du vérificateur d'effectuer de vraies démonstrations mathématiques dans lesquelles on ne peut pas tolérer d'erreur. Un outil pour aider le vérificateur à faire des démonstrations sans erreur devient alors nécessaire. C'est le domaine de l'*assistance à la preuve* et de la *preuve automatique*, que nous n'aborderons pas dans ce cours, même si lors des TP nous serons peut-être amenés à utiliser `Coq`[2] comme format de lecture des obligations de preuve.

Nous étudierons la technique la plus connue de vérification de programme : la logique de Hoare, puis nous expérimenterons ces notions à l'aide des outils de vérification de programme `Why` et `Jessie`.

2. La sémantique des langage n'est pas abordée dans ce polycopié, mais fait l'objet d'une partie du cours NFP209.

Chapitre 7

Systèmes logiques

La logique de Hoare étant un système logique formel, le présent chapitre est un rapide survol de cette notion. On y trouvera les définitions de règle d'inférence, d'arbre d'inférence et de jugement prouvable, dont nous aurons besoin pour définir la logique de Hoare plus loin. Pour plus de précision sur les systèmes logiques, voir le cours `NFP108`.

7.1 Les règles d'inférence

Un système logique est un couple $(\mathscr{J},\mathscr{R})$, où $\mathscr{J}$ est un ensemble de *jugements*, et $\mathscr{R}$ un ensemble de *règles de déduction* ou *règles d'inférence* permettant de définir le sous-ensemble $\mathscr{R}(\mathscr{J}) \subset \mathscr{J}$ des jugements dits *valides*. La forme des jugements est très variable, dans certains systèmes logiques il s'agit de formules logiques. Dans d'autres cas il s'agit de *séquents* de la forme $\Gamma \vdash F$ où Γ est un ensemble de formules et F une formule [1]. Dans la logique de Hoare *il s'agira de triplets de Hoare (voir section 8.1) ou de formules logiques*. Une *règle de déduction*, ou *règle d'inférence*, est une fraction de la forme :

$$\frac{J_1 \ \dots \ J_n}{\mathrm{J}}$$

où les J_i et J sont des *jugements*. Les J_i sont les *prémisses* et J est la *conclusion*. Il existe des règles particulière, appelées *axiomes*, qui n'ont aucune prémisse :

$$\frac{}{\mathrm{J}}$$

La signification de ce type de règle est "le jugement J est toujours vrai".

Voici un exemple de système logique $(\mathscr{J}_<,\mathscr{R}_{\text{INF}})$ dont les jugements $(\mathscr{J}_<)$ sont de la forme $i < j$ où $i, j \in \mathbb{N}$, et où les règles de déduction $(\mathscr{R}_{\text{INF}})$ nommées INF0 et INF+ sont les suivantes :

$$\text{INFXX}\ \frac{}{x \geq x} \qquad\qquad \frac{x \geq y}{x+1 \geq y}\ \text{INF+}$$

Si, comme dans la règle INF+, les jugements contiennent des variables, alors on peut *appliquer* la règle à toute valeur des variables. Par exemple les règles suivantes sont des *instances* de INFXX et INF+ :

$$\text{INFXX}\ \frac{}{2 \geq 2} \qquad \text{INF+}\ \frac{3 \geq 2}{4 \geq 2} \qquad\qquad \frac{5 \geq 1}{6 \geq 1}\ \text{INF+}$$

1. Voir le cours `NFP108`.

Dans la première x vaut 3 et y vaut 2, dans la deuxième x vaut 5 et y vaut 1.

7.2 Arbre de déduction

Un arbre de déduction dans un système logique $(\mathscr{J},\mathscr{R})$ est une combinaison finie de règles de la forme :

$$\dfrac{\dfrac{\dfrac{\vdots}{J_{11}} \cdots \dfrac{\vdots}{J_{1k_1}}}{J_1} \cdots \dfrac{\dfrac{\vdots}{J_{n1}} \cdots \dfrac{\vdots}{J_{nk_n}}}{J_n}}{J}$$

où chaque règle appliquée est une instance d'une règle de $\mathscr{R}$. J est la *racine* de l'arbre, les jugements n'ayant pas de prémisses sont les *feuilles*. Plus précisément :

Définition 7.2.1 (Arbre de déduction)**.** *L'ensemble des arbres de déduction d'un système $\mathscr{R}$ est défini récursivement comme suit :*

- *Si r est une instance d'une règle de $\mathscr{R}$, alors r est un arbre de déduction ;*
- *Si $t_1 \ldots t_2$ sont des arbres de déduction dont les racines sont $J_1 \ldots J_n$, et*

$$\frac{J_1 \ \ldots \ J_n}{J}$$

est une instance de règle de $\mathscr{R}$ alors l'arbre suivant est un arbre de déduction :

$$\frac{t_1 \ \ldots \ t_n}{J}$$

Par exemple l'arbre suivant est un arbre de déduction dans le système $(\mathscr{J}_<,\mathscr{R}_{\text{INF}})$ défini plus haut :

$$\textsc{Inf+}\ \dfrac{\textsc{Inf+}\ \dfrac{2 \geq 2}{3 \geq 2}}{4 \geq 2}$$

Définition 7.2.2 (Arbre de déduction complet)**.** *Un arbre de déduction* complet *est un arbre dans laquelle toutes les feuilles sont des (instances d') axiomes.*

Par exemple l'arbre suivant est un arbre de déduction complet dans le système $(\mathscr{J}_<,\mathscr{R}_{\text{INF}})$ défini plus haut :

$$\textsc{Inf+}\ \dfrac{\textsc{Inf+}\ \dfrac{\textsc{Infxx}\ \dfrac{}{2 \geq 2}}{3 \geq 2}}{4 \geq 2}$$

Le principe des systèmes de déduction est que les règles définissent l'ensemble des jugement *prouvables* comme l'ensemble des jugement pour lesquels il existe un arbre complet :

Définition 7.2.3 (Jugement prouvable)**.** *Soit $(\mathscr{J},\mathscr{R})$ un système logique, un jugement J est* prouvable *dans $(\mathscr{J},\mathscr{R})$ si il existe un arbre de déduction complet avec J à la racine. On parle alors d'*arbre de preuve *pour J.*

Donc pour prouver un jugement J dans un système de déduction $(\mathscr{J},\mathscr{R})$, il faut et il suffit d'exhiber un arbre de preuve complet pour J dans $\mathscr{R}$. Notez que ceci est la *définition* de "être prouvable" dans $(\mathscr{J},\mathscr{R})$. Le fait que l'ensemble des jugements prouvables est « intéressant » ou « utile » est un autre problème. Dans le cas de la logique de Hoare, où les jugements sont des triplets de Hoare, il existe un théorème (voir plus bas le théorème 8.5.1) qui établit que si un triplet est prouvable, alors il est « vrai ».

Chapitre 8

La logique de Hoare

Dans la logique de Hoare [3], un programme est considéré comme un *transformateur d'états*. Un état représente ici les valeurs de toutes les variables d'un programme [1]. L'exécution d'un programme a pour effet, si elle se termine, de transformer un état initial en un état final. La spécification d'un programme s'effectuera en formulant des propriétés sur ces deux états.

8.1 Les triplets de Hoare

Les *jugements* de la logique de Hoare sont des triplets de Hoare. Un triplet $\{P\}$ `prog` $\{Q\}$ est composé du programme `prog`, de la pré-condition P et de la post-condition Q. P et Q sont des *formules logiques* représentant des *propriétés* sur les variables du programme. En général il s'agit de formules de la *logique des prédicats* [2].

Soit le programme `divide` suivant :

```
a := 0;
b := x;
while (b >= y) do
   b := b - y;
   a := a + 1
done
```

Voici un premier exemple de triplet de Hoare :

$$\{\ y > 0 \land x > 0\ \}\ \texttt{divide}\ \{a \times y + b = x \land b \leq y\ \} \tag{8.1}$$

Il faut lire ce triplet comme suit :

1. Si la propriété $y > 0 \land x > 0$ est vraie avant l'exécution de `divide` (c'est-à-dire que l'état initial vérifie $y > 0 \land x > 0$)
2. *Et l'exécution de* `divide` *se termine*,
3. Alors la propriété $a \times y + b = x \land b \leq y$ est vraie après l'exécution de `divide`.

1. En toute rigueur ceci n'est pas suffisant : il faut aussi considérer la place mémoire disponible, le taux de remplissage des unités de stockage, la qualité du compilateur utilisé... Il n'est pas question ici de gérer tous ces paramètres.

2. Voir le cours `NFP108`.

Il faut comprendre qu'un triplet peut être vrai ou faux, exactement comme une formule logique[3]. On donne une définition précise de la notion de triplet vrai à la section 8.2. Par exemple le triplet ci-dessous n'est pas vrai car il existe des états initiaux pour lesquels l'état final ne sera pas tel que `b = 0.` :

$$\{\ y \neq 0\ \}\ \texttt{divide}\ \{a \times y + b = x \wedge b = 0\} \tag{8.2}$$

En revanche le triplet (8.1) semble vrai.

Remarque 8.1.1. *Lorsque la pré-condition est vide, cela signifie qu'on ne suppose rien avant l'exécution du programme, donc que la post-condition est toujours vraie après l'exécution. On peut toujours remplacer la pré-condition vide par* **`true`**.

De la même manière on peut remplacer la post-condition vide par **`true`**. *Cela signifie qu'aucune propriété particulière n'est vraie après l'exécution du programme (sauf* **`true`** *qui est toujours vraie de toute façon).*

8.2 Correction d'un triplet de Hoare

$$\{\ y \neq 0\ \}\ \texttt{divide}\ \{a \times y + b = x \wedge b \leq y\ \} \tag{8.3}$$

Le triplet (8.3) ci-dessus est problématique car, bien que vrai selon notre description intuitive ci-dessus, il implique des cas ou le programme `divide` ne se termine pas. Par exemple si `x` est négatif et `y` positif. On est donc amené à considérer deux formes de *correction* pour les programme.

Définition 8.2.1 (correction partielle)**.** *Le triplet de Hoare suivant* $\{P\}$ `prog` $\{Q\}$ *est vrai si pour tout état initial vérifiant P, si l'exécution de* `prog` *se termine, alors Q est vraie après l'exécution de* `prog`*. On dit que le programme* `prog` *est* partiellement correct *par rapport à* `P` *et* `Q`*.*

La correction totale s'écrit avec des $\langle\rangle$ (parfois avec des [ ]) :

Définition 8.2.2 (correction totale)**.** *Le triplet de Hoare suivant* $\langle P\rangle$ `prog` $\langle Q\rangle$ *est vrai si pour tout état initial de* `prog` *vérifiant P,* `prog` *se termine et Q est vraie après l'exécution de* `prog`*. On dit que le programme* `prog` *est* totalement correct *par rapport à* `P` *et* `Q`*.*

Nous verrons plus loin qu'il existe des règles pour prouver qu'un triplets est vrai pour chacune de ces deux définitions. Avant cela nous allons préciser le langage de programmation que nous considérons et le langage des prémisses.

8.3 Les instructions

Il n'est pas impossible – c'est un domaine de recherche actuel – de vérifier des programmes utilisant des fonctionnalités comme l'arithmétique sur les pointeurs, le partage de structures, le parallélisme etc, mais il n'en est pas question ici. On se fixe l'ensemble d'instructions autorisées suivant :

3. Voir le cours `NFP108`.

$$
\begin{array}{rl}
I ::= & I\texttt{;}I \\
| & \textbf{begin}\ I\ \textbf{end} \\
| & \texttt{x := } E \\
| & \textbf{if}\ B\ \textbf{then}\ I\ \textbf{else}\ I \\
| & \textbf{while}\ B\ \textbf{do}\ I\ \textbf{end} \\
E ::= & \texttt{x} \mid \texttt{y} \mid \ldots \mid E\texttt{+}E \mid E\star E \mid E-E \mid \ldots \mid f(E,E\ldots) \mid B \\
B ::= & B\ \textbf{and}\ B \mid B\ \textbf{or}\ B \mid \textbf{not}\ B\ldots \mid E\texttt{<}E \mid E\texttt{<=}E \mid E\texttt{=}E\ldots
\end{array}
$$

Ceci est une version simplifiée de ce que `Why` et `Jessie` autorisent. E correspond aux expressions, I aux programmes. Notons que malgré le faible nombre d'instructions on peut exprimer de nombreux programmes.

Remarque 8.3.1. *On ne considérera que les programmes bien typés. En particulier on ne cherchera pas à démontrer des triplets dans lesquels le programme provoquerait des erreurs de type à l'exécution. En pratique on supposera que le programme a été accepté par un compilateur correct.*

8.4 Les assertions

Dans les triplets ci-dessus (8.1) $\{\mathtt{y} \neq 0\}$, $\{\mathtt{x}\leq 0 \wedge \mathtt{y}\leq 0\}$ et $\{\mathtt{a}\times\mathtt{y}+\mathtt{b}=\mathtt{x}\wedge 0\leq\mathtt{b}\leq\mathtt{y}\}$ sont des *assertions*. Plus précisément dans un triplet de la forme $\{P\}\mathtt{prog}\{Q\}$ les assertions P et Q sont en général des formules de la logiques de prédicats où les prédicats portent sur les variables du programme et sur des variables supplémentaires (souvent notées avec des indices $\mathtt{x}_i$, $\mathtt{y}_0$,...).

Notez que les expression booléennes (B ci-dessus) du programme, par exemple `x <= 0` **`and`** `y <= 0` ont un équivalent immédiat dans les assertions, ici $\mathtt{x} \leq 0 \wedge \mathtt{y} \leq 0$, où $\leq$ est un prédicat binaire. Les premières sont des expressions du programme, ayant à l'exécution la valeur `vrai` ou `faux`, alors que les deuxièmes sont des formules logiques décrivant des propriétés sur les variables du programme.

Les assertions servent à écrire :

- Des pré- et post-conditions,
- Des assertions spéciales pour les boucles :
  - Les *invariants de boucle* sont des formules logiques destinées à établir qu'une propriété est vraie à chaque itération d'une boucle. Voir la section 8.5.5.
  - Les *variants de boucle* ne sont pas des formules, mais des expressions (E ci-dessus), dont la valeur doit diminuer à chaque itération, permettant ainsi de démontrer que l'itération s'arrêtera toujours. Voir la section 8.5.6.

8.5 Les règles de Hoare

Une règle de Hoare est une règle de déduction comme évoqué à la section 7.1, c'est-à-dire une fraction de la forme :

$$\frac{\text{triplet}_1 \ \ldots \ \text{triplet}_n}{\text{triplet}}$$

Les prémisses et la conséquence (c'est-à-dire les jugements) sont des triplets de Hoare. Une telle fraction se lit de la manière suivante : "Si les triplets prémisses sont prouvables, alors le triplet conclusion aussi". Dans un premier temps on n'étudiera que la correction partielle des triplets, puis on verra comment s'assurer de la terminaison des programme en modifiant certaines règles.

Toute la technique de la logique de Hoare repose sur le théorème suivant, que nous ne démontrons pas ici mais qui est démontrable *par rapport* à la sémantique du langage présenté en section 8.3.

Théorème 8.5.1 (Correction des règles de Hoare)**.** *Étant donné un triplet de Hoare t, si il existe un arbre de déduction de Hoare complet ayant t à sa racine, alors le triplet est vrai au sens de la section 8.2.*

Nous allons définir et illustrer ce qu'est un arbre de déduction de Hoare dans la suite en détaillant chaque règle. Démontrer cette propriété nécessite de définir la sémantique mathématique de chaque instruction, de vérifier cette sémantique dans le compilateur utilisé pour compiler ce programme et enfin d'en déduire la correction des règles de Hoare une par une. Nous ne chercherons pas à le faire, nous admettrons simplement que *si il est possible de construire un arbre de dérivation complet* (c'est-à-dire dont toutes les branches se terminent par un axiome) *dont la racine est un triplet T, alors T est vrai.*

8.5.1 Affectation

L'exemple suivant illustre la règle d'affectation, dont la première formulation ci-dessous est peu intuitive. Soit le triplet, manifestement vrai, `{z-y≥0} x:=z-y {x≥0}`. La règle de déduction de l'affectation doit permettre de prouver ce triplet donc la règle de l'affectation doit pouvoir s'appliquer comme suit :

$$\frac{}{\{\mathtt{z-y}\geq 0\}\ \mathtt{x:=z-y}\ \{\mathtt{x}\geq 0\}}$$

On voit que $\mathtt{x} \geq 0$ est vrai après l'exécution du programme seulement si la même propriété est vraie *pour* `z-y` *avant le programme*. La règle de l'affectation exprime ceci de la manière suivante :

$$\textsc{Aff1}\ \frac{}{\{P[\mathtt{x} \leftarrow E]\}\mathtt{x:=}E\{P\}}$$

où la notation $P[\mathtt{x} \leftarrow E]$ signifie « P dans laquelle x a été remplacé par E ». Si on lit cette règle de la manière suivante elle devient claire : Pour que P soit vraie pour x après le programme `x:=`E, il fallait qu'elle soit vraie pour E avant le programme. On peut trouver une forme plus intuitive pour cette règle en essayant d'avoir la pré-condition P et une expression en fonction de P dans la post-condition. Soit une règle de cette forme :

$$\frac{}{\{P\}\quad \mathtt{x:=}E\quad \{??\}}$$

Malheureusement ce n'est pas simple car l'affectation entraîne les deux choses suivantes :

- Si x apparaît dans E, il s'agit de l'ancienne valeur de x. Cette ancienne valeur de x, qu'on notera $\mathtt{x}_0$ [4] est donc telle que après l'affectation on a : $\mathtt{x} = E[\mathtt{x} \leftarrow \mathtt{x}_0]$.
- Si x apparaît dans P, alors P n'est plus vraie car x a changé, en revanche $P[\mathtt{x} \leftarrow \mathtt{x}_0]$ est encore vraie car $\mathtt{x}_0$ est l'ancienne valeur de x.

On exprime ces deux propriétés dans la règle de hoare de l'affectation (deuxième version), dans laquelle $p[x \leftarrow v]$ signifie p « dans lequel on remplace » x par v :

$$\textsc{Aff}\ \frac{}{\{P\}\quad \mathtt{x:=}E\quad \{P[\mathtt{x} \leftarrow \mathtt{x}_0]\ \wedge\ \mathtt{x}{=}E[\mathtt{x} \leftarrow \mathtt{x}_0]\}}$$

4. le nom $\mathtt{x}_0$ de cette variable importe peu, il suffit qu'il soit « frais » c'est-à-dire qu'aucune autre variable ne possède déjà ce nom.

On gagnera en compréhension en voyant que le x_0 est en fait une variable existentielle :

$$\text{AFF}\ \frac{}{\{P\}\quad x:=E\quad \{\exists x_0,\ P[x\leftarrow x_0]\ \wedge\ x=E[x\leftarrow x_0]\}}$$

Dans l'exemple, on applique la règle comme suit :

$$\text{AFF}\ \frac{}{\{\ x-y\ \geq 0\}\quad x:=x-y\quad \{\ x_0 - y \geq 0 \wedge x = x_0 - y\}}$$

Ce qui permet de déduire que $x \geq 0$ après l'exécution de cette instruction (voir la règle CONSEQ pour l'insertion de déductions logique dans un arbre de déduction de Hoare).

Si on désire comparer la valeur de x avant et après un programme, alors on fait apparaître une autre version de x, par exemple x_i, dans la pré-condition :

$$\text{AFF}\ \frac{}{\{\ x = x_i\}\quad x:=x+1\quad \{\ x_0 = x_i \wedge x = x_0 + 1\}}$$

d'où l'on peut déduire que la valeur de x est plus grande après le programme qu'avant (de même, voir la règle CONSEQ les déductions logique).

Exercice 8.16 *Écrivez un arbre de dérivation ayant à sa racine un triplet de la forme :* $\{\ y\geq 2\ \}y:=2*y\ \{??\ \}$. □

8.5.2 Séquence

Pour la « séquence » d'instructions, la règle est relativement simple à construire :

$$\text{SEQ}\ \frac{\{P\}C_1\{Q\}\qquad \{Q\}C_2\{R\}}{\{P\}C_1\ ;\ C_2\{R\}}$$

On applique cette règle sur un exemple (notez que les deux prémisses sont prouvées avec la règle AFF1) :

$$\text{SEQ}\ \frac{\text{AFF1}\ \frac{}{\{0+x\geq 0\}\quad a:=0\quad \{a+x\geq 0\}}\qquad \frac{}{\{a+x\geq 0\}\quad b:=x\quad \{a+b\geq 0\}}\ \text{AFF1}}{\{0+x\geq 0\}\quad a:=0;b:=x\quad \{a+b\geq 0\}}$$

Exercice 8.17 *Écrivez un arbre de dérivation ayant à sa racine un triplet de la forme :* $\{\ x = x_i \wedge x-y\geq 0\ \}x:=x-y;y:=y+x\ \{??\ \}$ □

Exercice 8.18 *Démontrez la correction des triplets suivants :*

1. `{x=0} x:=x+1; x:=x+1 {x=2}`
2. `{x>0} x:=x+1; x:=x+1 {x>2}`
3. `{x=0} x:=x+1; x:=x+1 {x≥2}`
4. `{x>0} x:=x+1; x:=x+1 {x≥2}`
5. `{P} swap {Q}` *où* `swap` *est le programme qui intervertit deux variables et P et Q la spécification naturelle pour* `swap` *(utilisant* $\mathtt{x_i}$ *et* $\mathtt{y_i}$*).*

□

Exercice 8.19 *Démontrez la règle suivante :*

$$\textsc{Seq3}\ \frac{\{P\}C_1\{Q\} \qquad \{Q\}C_1\{R\} \qquad \{R\}C_2\{S\}}{\{P\}C_1\ ;\ C_2\{S\}}$$

□

8.5.3 Conséquence

Les propriétés que l'on peut prouver directement à partir des règles vues jusqu'ici sont d'une forme très contrainte, or il faut pouvoir déduire de ces propriétés celles qui nous intéressent. Ceci se fait par simple *déduction logique* à partir des propriétés prouvées par les règles précédentes. Pour cela on ajoute la règle suivante (notez que les deux implications ne sont pas dans le même sens) :

$$\textsc{Conseq}\ \frac{P \Rightarrow P' \qquad \{P'\}C\{Q'\} \qquad Q' \Rightarrow Q}{\{P\}C\{Q\}}$$

Ce qui permet de déduire dans l'exercice ci-dessus que le triplet de Hoare suivant est valide (car $0+x = x$) :

$$\textsc{Conseq}\ \frac{\mathtt{x} \geq 0 \Rightarrow 0+\mathtt{x} \geq 0 \qquad \textsc{Seq}\ \dfrac{\textsc{Aff1}\ \dfrac{}{\{0+x\geq 0\}\mathtt{a:=0}\{a+x\geq 0\}} \qquad \dfrac{}{\{a+x\geq 0\}\mathtt{b:=x}\{a+b\geq 0\}}\ \textsc{Aff1}}{\{0+x\geq 0\}\mathtt{a:=0;b:=x}\{a+b\geq 0\}} \qquad \mathtt{a}+\mathtt{b} \geq 0 \Rightarrow \mathtt{a} \geq -\mathtt{b}}{\{x\geq 0\}\mathtt{a:=0;b:=x}\{a \geq -b\}}$$

Exercice 8.20 *Prouvez le triplet suivant à l'aide des règle* Aff *et* Seq *(mais pas* Aff1*) :*

`{0+x≥0} a:=0; b:=x {a+b≥0}`

□

Exercice 8.21 *Complétez la dérivation de l'exercice 8.17 pour en déduire une post-condition plus compacte.* □

Les preuves des implications sont à prouver aussi, On peut ajouter des règles de déduction logique pour les vérifier en même temps que les triplets de Hoare. Dans le cadre de `Why` les preuves logiques sont laissées à l'utilisateur sous la forme d'obligations de preuves dans le format d'un assistant de preuve (`Coq`, PVS, Simplify etc). Ces implications ne sont en général pas prouvables automatiquement. En particulier, il peut arriver que la correction d'un algorithme repose sur des résultats mathématiques difficiles, dont la démonstration est hors de portée des outils de preuve.

Lors de l'utilisation de `Jessie` en TP, nous nous contenterons de lire les obligations de preuve (voire de les simplifier à l'aide de commandes basiques) dans le format `Coq` afin de nous persuader qu'elles sont justes.

8.5.4 Conditionnelle

On ajoute ensuite la règle de la conditionnelle, qui exprime que la post-condition doit être vérifiée dans les deux branches du **`if`**, *chacune sous la condition que le résultat du test correspond* à la branche (**`true`** pour le **`then`**, **`false`** pour le **`else`**).

$$\textsc{Cond}\;\frac{\{\{P\wedge B\}\quad I_1\quad\{Q\}\qquad\{P\wedge\neg B\}\quad I_2\quad\{Q\}}{\{P\}\quad \mathbf{if}\ B\ \mathbf{then}\ I_1\ \mathbf{else}\ I_2\quad\{Q\}}$$

Exemple :

$$\textsc{Cond}\;\frac{\textsc{Aff}\;\dfrac{}{\{\ y=0\}\ \ x:=y\ \ \{x=0\}}\qquad \dfrac{(y\neq 0)\Rightarrow(0=0)\quad \{0=0\}\ \ x:=0\ \ \{x=0\}}{\{\ y\neq 0\ \}\ \ x:=0\ \ \{x=0\}}\;\textsc{Conseq}}{\{\ \}\ \mathbf{if}\ y=0\ \mathbf{then}\ x:=y\ \mathbf{else}\ x:=0\ \{x=0\}}$$

En particulier si une des branches est impossible, alors la prémisse correspondante est juste grâce à la règle CONSEQ, (voir l'exercice ci-dessous).

*** Exercice 8.22** *Prouvez que le triplet suivant est vrai :*
{ x ≥0 } **`if`** *x >= 0* **`then`** *y:=8* **`else`** *y:=9 {y=8 }*. □

Exercice 8.23 *Prouvez le triplet suivant {x ≥ 0}* **`if`** *x<>0* **`then`** *x:=x-1* **`else`** *x:=x+1 {x≥0}*
□

8.5.5 While

Le corps d'une boucle sera exécuté un nombre arbitraire de fois à l'exécution du programme. En conséquence pour démontrer qu'une propriété P est vraie en sortant de la boucle il est nécessaire de démontrer que P est vraie quelque soit le nombre d'itération(s), en particulier :

- si le nombre d'itérations est zéro, P doit donc être vraie avant d'exécuter la boucle ;
- si le nombre d'itérations est au moins un, P doit donc rester vrai à chaque itération.

C'est ce qu'exprime La règle suivante, dans laquelle P est appelée *invariant* de la boucle. Notez que P doit être vraie avant la boucle, et que le corps de la boucle doit maintenir P vraie (sous la condition du while).

Notez également qu'on sait qu'à la sortie de la boucle, le test est faux. Notez enfin que rien ne garantit dans cette règle que la boucle s'arrêtera effectivement. Il s'agit bien de correction *partielle*.

$$\textsc{While}\ \frac{\{P \wedge B\}\quad C \quad \{P\}}{\{P\}\quad \texttt{while}\ B\ \texttt{do}\ C\ \texttt{done}\quad \{P \wedge \neg B\}}$$

Exemple 8.5.2. *Exemple d'application de la règle* WHILE *:*

$$\textsc{While}\ \dfrac{\textsc{Conseq}\ \dfrac{(x \geq 0 \wedge x < b) \Rightarrow x+1 \geq 0 \qquad \overline{\{x+1 \geq 0\}x\texttt{:=}x\texttt{+}1\{x \geq 0\}}\ \textsc{Aff}}{\{x \geq 0 \wedge x < b\}x\texttt{:=}x\texttt{+}1\{x \geq 0\}}}{\{x \geq 0\}\quad \texttt{while}\ x\texttt{<}b\ \texttt{do}\ x\texttt{:=}x\texttt{+}1\ \texttt{done}\quad \{x \geq 0 \wedge \neg(x < b)\}}$$

Exercice 8.24 `{ } while x<>0 do x:=x-1 done {x=0}` □

8.5.6 While avec variant (correction totale)

Pour prouver la correction totale d'un programme, il faut d'une part prouver sa correction partielle, d'autre part s'assurer que les boucles (et les fonctions récursives, ce que nous ne considérons pas dans ce cours) s'arrêtent nécessairement lorsque les pré-conditions sont vérifiées. On modifie donc la règle du while afin d'exhiber une expression, qu'on appellera le *variant*, qui doit :

- être une fonction des variables du programme ;
- être positive lorsqu'on entre dans la boucle (c'est-à-dire lorsque la condition de la boucle est vérifiée *et* (donc) lorsque l'invariant de la boucle est vérifié ;
- décroître strictement à chaque itération de la boucle.

Voici la règle modifiée dans laquelle le variant est noté E. la variable n est une variable ajoutée, qui n'apparaît pas dans le programme (comme x_i et y_i plus haut) :

$$\textsc{WhileT}\ \frac{\langle P \wedge B \wedge (E=n)\rangle\, C\, \langle P \wedge E < n \wedge E \geq 0 \rangle}{\langle P\rangle\quad \texttt{while}\ B\ \texttt{do}\ C\ \texttt{done}\quad \langle P \wedge \neg B\rangle}$$

Exercice 8.25 *Que faut-il faire pour prouver la correction totale des triplets de l'exercice 8.18 ?* □

Exercice 8.26 *Démontrez la correction de ces triplets (la syntaxe* ⟨ ⟩ *indique qu'on doit prouver la correction totale) :* `⟨x≥0⟩ while x>0 do x:=x-1 end ⟨x=0⟩`

□

Exercice 8.27 *Même question :*

```
⟨ y > 0 ⟩ while y <= r do r:=r-y; q=q+1 done ⟨?⟩
```

□

8.5.7 Les tableaux

On s'intéresse maintenant aux programmes utilisant les tableaux. Dans le cas des tableaux de taille prédéfinie et en l'absence d'*aliasing* (partage par plusieurs entité d'un même espace mémoire, par exemple par superposition de tableaux) on peut considérer chaque case de chaque tableau comme une nouvelle variable.

Comme pour les variables nous posons que le compilateur est correct et qu'il effectuera les vérifications de typage des instructions : les éléments des tableaux sont donc utilisés conformément à leur type. En revanche le typage ne garantit pas que les accès aux tableaux se font toujours à l'intérieur des bornes de ces tableaux. Or un accès à l'extérieur d'un tableau est une opération incorrecte. Un programme ne peut donc être correct au sens de la logique de Hoare que si quand la pré-condition est vraie avant l'exécution du programme alors tous les accès aux tableaux pendant l'exécution sont valides.

Il faut donc démontrer pour chaque accès `t[i]` contenu dans le programme qu'à l'exécution `i` sera compris entre 0 et taille de $t - 1$. En conséquence on ajoute des *obligations de preuves* supplémentaires concernant les accès aux tableaux. Par exemple, dans le triplet de Hoare suivant :

```
{ ∀ 0 ≤ i < n, t[i]=0 ∧ |t|=n }
t[0] = 1;
j=1;
while j<n do
 t[j]=t[j-1]*2;
 j:=j+1
done
{ ∀ 0 ≤ i < n, t[i]>0 }
```

Il y a un certain nombre d'obligations de preuve supplémentaires à démontrer, correspondant aux *assertions implicites* suivantes :

```
{ ∀ 0 ≤ i < n, t[i]=0 ∧ |t|=n }
{ 0 ≤ 0 < n }
t[0] = 1;
j=1;
while j<n do
 { 0 ≤ j < n ∧ 0 ≤ (j-1) < n }
 t[j]=t[j-1]*2;
 j:=j+1
done
{ ∀ 0 ≤ i < n, t[i]>0 }
```

Lors des TP `Jessie`, ces assertions supplémentaires apparaîtront sous la forme d'obligations de preuves supplémentaires utilisant le *prédicat* `\valid_range(t,i)` qui signifie que `i` est un numéro de case valide pour le tableau `t` :

```
{ ∀ 0 ≤ i < n, t[i]=0 ∧ |t|=n }
{ \valid_range(t,i) }
t[0] = 1;
j=1;
while j<n do
 { \valid_range(t,j) ∧ \valid_range(t,j-1)}
 t[j]=t[j-1]*2;
```

```
 j:=j+1
done
{ ∀ 0 ≤ i < n, t[i]>0 }
```

Exercice 8.28 *Prouvez l'algorithme de la recherche dans un tableau d'entiers.* □

Exercice 8.29 *Prouvez un algorithme qui inverse l'ordre des éléments d'un tableau d'entiers.* □

Exercice 8.30 *Prouvez un algorithme de tri sur un tableau d'entiers.* □

Exercice 8.31 *Prouvez un algorithme qui calcul la moyenne des éléments d'un tableau d'entiers.* □

Chapitre 9

Conclusion

9.1 Les difficultés

En général ce qui est difficile lors de la preuve d'un programme se situent à deux endroits : (a) L'écriture de assertions, en particulier les variants et invariants de boucles sont parfois durs à trouver, ils nécessitent une compréhension profonde le l'algorithme. (b) les preuves des étapes de déduction (règle CONSEQ) peuvent être arbitrairement difficiles (ex : théorème de Bertrand).

9.2 Programme annoté

Un programme annoté est un programme dans lequel des assertions sont insérées. On peut insérer des annotations avant et après chaque instruction. On peut même introduire plusieurs annotations entre deux instructions. On considère cependant en général qu'un programme est suffisamment (ce qui ne veut pas dire correctement) annoté si il y a des assertions aux endroits suivants :

- Avant et après le programme,
- avant chaque instruction qui n'est pas une affectation,
- après le mot-clé **do** dans les boucles **while**, c'est là qu'on insère l'invariant et le variant de la boucle.

Exemple :

```
{ x =n }
y:=1;
{ y = 1 ∧ x = n }
while x != 0 do { y * x! = n! } (* x! représente la factorielle au sens mathématique *)
  y:=y * x;
  x:=x-1
done
{ y = n! }
```

En théorie il n'y a pas besoin de mettre autant d'annotations, il suffit de préciser les variants et invariants de boucle. L'ajout d'annotation supplémentaire permet en fait de déclencher la règle de conséquence, afin de déduire des propriétés sur certains points intermédiaires du programme. Dans cet exemple, on voit que l'annotation $\{y = 1 \land x = n\}$ n'est en fait pas nécessaire car elle est exactement identique à ce que la règle AFF permet de déduire à partir du triplet incomplet suivant : `{x=n} x:=1 {?}`. L'ajout d'annotations en dehors des (in)variants de boucles est toutefois utile pour simplifier les propriétés en des points intermédiaires du programme, grâce à la règle de conséquence, afin de garder des formules de taille raisonnable.

9.3 exercices

Exercice 9.32 *Vérifiez la correction partielle du programme annoté de la section 9.2. Quelle annotation faut–il ajouter pour la correction totale ? Vérifiez.* □

Exercice 9.33 *Écrivez et annotez le programme* `divide` *pour la correction partielle. Attention, Les annotations doivent assurer que le dividende et le diviseur n'ont pas changer. Recommencez pour la correction totale.* □

Chapitre 10

Jessie

Exercice 10.34 *Traduisez en* `Jessie` *le triplet suivant :*

```
{x = 1 ∨ x =−1} if x<=0 then x:=x+1 else x:=x−1 end { x = 0 }
```

Les obligations de preuve générées sont-elles démontrables ?

□

10.1 max

Exercice 10.35 *Écrivez et spécifiez le programme* `max` *qui retourne le maximum de ces deux arguments.*
□

10.2 swap

Exercice 10.36 *Écrivez et spécifiez le programme* `swap` *qui échange les valeurs de deux variables globales. Les obligations de preuve générées sont-elles démontrables ?*

```
int q;
int r;

/*@ requires ...
    ensures ... */

void swap (...){
  ...
}
```

□

10.3 divide

Exercice 10.37 *Écrivez et prouvez le programme* `divide`*. Les obligations de preuve générées sont-elles démontrables ? Le quotient q et le reste r seront stockés dans des variables globales.* □

10.4 Autres exemples

Autres exemple à faire en TP :

La valeur absolue, deux version :

```
//@ ensures *p >= 0
void abs1(int *p) {
  if (*p < 0) *p = -*p;
}

/*@ requires \valid(p)
  @ ensures *p >= 0
  @*/
void abs2(int *p) {
  if (*p < 0) *p = -*p;
}



/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" abs.c ; make -f abs.makefile coq.dep
*** End: ***
*/
```

La factorielle.

La recherche dans un tableau (il faut avoir vu les tableaux) :

```
/*@ requires \valid_range(t,0,n-1)
  @ ensures
  @   (0 <= \result < n => t[\result] == v) &&
  @   (\result == n => ∀ int i; 0 <= i < n => t[i] != v)
  @*/
int index(int t[], int n, int v) {
  int i = 0;
  /*@ invariant 0 <= i && ∀ int k; 0 <= k < i => t[k] != v
    @ variant n - i */
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
```

```
  return i;
}



/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" search.c ; make -f search.makefile c
*** End: ***
*/
```

Bibliographie

[1] Grady Booch James Rumbaugh Ivar Jacobson. *Le guide de l'utilisateur UML.* , 2000.

[2] Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.

[3] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 :576–583, 1969.

[4] Arnaud Lallouet. Mathématiques discrètes pour l'informatique. Polycopié – Université d'Orléans, 1997.

Troisième partie

Solutions des exercices

Solution de l'Exercice 2.1

On veut démontrer la propriété suivante : $\forall x \in \mathbb{N}, \texttt{pos}\ x \geq 0$. L'équation de la fonction pos est la suivante :

$$\texttt{pos}\ x = \begin{cases} x & \text{si } x \geq 0 \\ x+1 & \text{sinon} \end{cases}$$

La démonstration se fait par cas :
- Soit x est positif ou nul, et alors $\texttt{pos}\ x = x$ l'est aussi. OK.
- Soit x est négatif, et alors $\texttt{pos}\ x = (-x)$ est positif. OK.

□

Solution de l'Exercice 3.2

Il y a trois appels récursifs dans cette fonction. le premier se fait sur $(x-1, 1)$, le deuxième sur $(x-1, (\texttt{ack}\ x\ (y-1)))$ et le troisième se fait sur $(x, y-1)$. On prend la composition lexicographique suivante :

$$(x,y) < (x',y') ssi \begin{cases} x <_{\mathbb{N}} x' \text{ ou bien :} \\ x = x' \text{ et } y <_{\mathbb{N}} y' \end{cases}$$

Les appels récursifs sont tous décroissants pour cet ordre. Comme il s'agit d'une composition lexicographique de deux ordres bien fondés, c'est un ordre bien fondé. Conclusion ack termine. □

Solution de l'Exercice 3.3

On prend la composition lexicographique de l'ordre standard sur $\mathbb{N}$ sur chacun des deux arguments, en prenant l'ordre sur le second argument en premier. Soit finalement :

$$(x,y) <_1 (y',z')\ \ ssi\ \begin{cases} y & < & y' \text{ ou,} \\ y & = & y' \text{ et } x < x' \end{cases}$$

□

Solution de l'Exercice 5.4

Comme on a déjà démontré que $\forall n \in \mathbb{Z}, \texttt{f}\ n \geq 0$, on sait que si $\texttt{f}\ n \leq 0$ alors la propriété est vraie (car il n'y a pas d'entier positif strictement inférieur à 0). Il reste à prouver cela pour $\texttt{f}\ n \neq 0$, c'est-à-dire pour $n > 0$. On se place donc sur $\mathbb{N}^+$, et on prouve la propriété *sans récurrence* :

Pour tout $i > 0$, on a par 5.1 : $\texttt{f}((i+1)) = i+1+\texttt{f}(i)$, or $i+1 > 0$ donc $\texttt{f}((i+1)) > \texttt{f}(i)$. □

Solution de l'Exercice 5.5

1. Par induction sur l.
   Base : l = `[]`. `filtrez` l = `[]` et la propriété est vérifiée (car il n'y a pas d'élément dans `filtrez` l).
   Induction : Soit l une liste, supposons que $\forall x \in \texttt{filtrez}\ l, x \geq 0$. Alors pour tout y :

$$\texttt{filtrez}\ (y::l) = \begin{cases} y :: \texttt{filtrez}\ l & \text{si } y \geq 0 \\ \texttt{filtrez}\ l & \text{sinon} \end{cases}$$

   On distingue donc deux cas :
   (a) $y \geq 0$ et $\texttt{filtrez}\ y :: l = y :: \texttt{filtrez}\ l$ comme par hypothèse d'induction $\forall x \in \texttt{filtrez}\ l, x \geq 0$, on a bien que $\forall x \in (y :: \texttt{filtrez}\ l), x \geq 0$.
   (b) $y < 0$ et $\texttt{filtrez}\ y :: l = \texttt{filtrez}\ l$, par hypothèse d'induction, $\forall x \in (\texttt{filtrez}\ l), x \geq 0$.

2. Il faut spécifier que *tous* les éléments du résultats appartienne à l'argument :

$$\forall l, \forall x \in \mathbb{Z}, x \geq 0 \rightarrow x \in \texttt{filtrez}\ l$$

$$\forall l, \forall x \in \texttt{filtrez}\ l, x \in\ l$$

Par récurrence sur l également.

□

Solution de l'Exercice 5.6

Même preuve que l'exercice 5.5. □

Solution de l'Exercice 5.8

1.
```
let f l =
  match l with
  | [] -> []
  | x::[] -> []
  | x::y::l' -> x::f l'
```

2. Montrons cette propriété par récurrence forte sur la longueur de la liste.
   – Base : $|l| = 0$ donc $l = Nil$. $|f(Nil)| = |Nil| = 0 \quad \leq \quad |Nil|/2 = 0$ OK.
   – Soit n un entier, supposons que pour toute liste l de longueur $m \leq n$, $|f(l)| \leq |l|/2$, montrons qu'alors pour toute liste L de longueur $n+1$, $|f(L)| \leq |L|/2$.
   L est nécessairement de la forme $e :: l$ où l est de longueur n. On distingue 2 cas :
     – $L = e :: Nil$, alors $|f(L)| = |nil| = 0 \leq |L|/2$ OK.
     – $L = e :: e' :: l'$ donc $|f(L)| = |f(e :: e' :: l')| = |e :: f(l')| = 1 + |f(l')|$.
     Or l' est de longueur inférieure à n donc par hypothèse de récurrence : $|f(l')| \leq |l'|/2$.
     Donc $|f(L)| \leq 1 + |l'|/2 = (2 + |l'|)/2 = |L|/2$. OK.

□

Solution de l'Exercice 5.9

1. Facile, on utilise les équations. c est un couple, donc $c = (x,y)$, et donc

$$\begin{aligned} (fst(c), snd(c)) &= (fst(x,y), snd(x,y)) \\ &= (x,y) \ = \ c \quad OK. \end{aligned}$$

2. On montre cette propriété par induction.
   – Base : $h(g(Nil)) = Nil$ OK.
   – Soit l une liste telle que $h(g(l)) = l$, alors pour tout couple (x,y) :

$$\begin{aligned} h(g((x,y) :: l)) &= h((x :: fst(g(l)))\ ,\ (y :: snd(g(l)))) && \text{par (5.8)} \\ &= (x,y) :: (h(fst(g(l)))\ , snd(g(l)))) && \text{par (5.10)} \\ &= (x,y) :: h(g(l)) && \text{par (A)} \\ &= (x,y) :: l && \text{par hypothèse d'induction, OK.} \end{aligned}$$

□

Solution de l'Exercice 5.10

Par induction sur l.

Base : $l = \texttt{[]}$, alors $g\ l = \texttt{[]}$. OK.

Induction : $l = n :: l'$, alors $g\ l = (n+1) :: g\ l'$ et par hypothèse d'induction $|g\ l'| = |l'|$, donc $|g\ l| = |(n+1) :: g\ l'| = 1 + |g\ l'| = 1 + |l'|$. Comme $|l| = 1 + |l'|$, la propriété et bien vérifiée.

Par induction la propriété est vérifiée pour toute liste l. □

Solution de l'Exercice 5.11

L'équation de `div` est : $\texttt{div}\ a\ b = \begin{cases} 0 & \text{si } a < b \\ 1 + \texttt{div}\ (a-b)\ b & \text{sinon} \end{cases}$

La partie $0 \leq (\texttt{div}\ a\ b) \times b$ se montre par récurrence très facilement. La partie $(\texttt{div}\ a\ b) \times b \leq a$ se démontre par *récurrence forte* sur a comme suit :

Soit $b > 0$ un entier naturel non nul quelconque,

Base : Si $a = 0$, alors $a < b$ et par l'équation : $(\texttt{div}\ a\ b) \times b = 0 \leq a$. OK.

Récurrence : Supposons que pour tout $i < a, (\texttt{div}\ i\ b) \times b \leq i$. On procède par cas :

- Si $a < b$, alors $(\texttt{div}\ a\ b) \times b = 0 \leq a$. OK.
- Sinon $\texttt{div}\ a\ b = 1 + \texttt{div}\ (a-b)\ b$. Or $a - b < a$ et donc par hypothèse de récurrence $(\texttt{div}\ (a-b)\ b) \times b \leq (a-b)$. Donc $(\texttt{div}\ a\ b) \times b = (1 + \texttt{div}\ (a-b)\ b) \times b = (\texttt{div}\ (a-b)\ b) \times b + b \leq (a-b) + b = a$. OK.

□

Solution de l'Exercice 5.12

1. $\texttt{modulo}\ a\ b = \begin{cases} a & \text{si } a < b \\ \texttt{modulo}\ (a-b)\ b & \text{sinon} \end{cases}$
2. Par récurrence forte sur x également. Soit x un entier, supposons que $(\forall y > 0,\ \exists q \geq 0,\ x = qy + (\texttt{modulo}\ x\ y))$. Deux cas :
   (a) $x < y$, alors $\texttt{modulo}\ x\ y = x$, la propriété est vérifié avec $q = 0$ ($x = Oy + \texttt{modulo}\ x\ y$).
   (b) $x \geq y$, Alors $\texttt{modulo}\ x\ y = \texttt{modulo}\ (x-y)\ y$ or $x - y < x$ donc par hypothèse de récurrence il existe q tel que $x - y = qy + \texttt{modulo}\ (x-y)\ y$. Donc $x = y + qy + \texttt{modulo}\ (x-y)\ y = (q+1)y + \texttt{modulo}\ (x-y)\ y$. Donc la propriété est vérifiée pour x.

□

Solution de l'Exercice 5.13

□

Solution de l'Exercice 5.14

- $B_{Cond} = \{\texttt{true}, \texttt{false}, \texttt{=}\}$ où si $x, y \in \{a, b\}^*$ alors $x = y\ \in Cond$.
- $\Omega_{Cond} = \{\ \&\ , \texttt{or}\}$ où
  - Si $c_1, c_2 \in Cond$ alors $c_1\ \ \&\ \ c_2\ \in Cond$
  - Si $c_1, c_2 \in Cond$ alors $c_1\ \texttt{or}\ c_2\ \in Cond$

□

Solution de l'Exercice 8.16

$$\textsc{Aff}\ \dfrac{}{\{\ y\ \geq 2\ \}\quad y:=2*y\quad \{\ y_0\ \geq 2\ \land y\ =\ 2\ *\ y_0\}}$$

□

Solution de l'Exercice 8.17

$$\textsc{Seq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{\ x=x_i\land x-y\geq 0\ \}\quad x:=x-y \quad \{\ x_0=x_i\land x_0-y\ \geq 0\ \land x=x_0-y\}} \qquad \dfrac{}{\{\ x_0=x_i\land x_0-y\ \geq 0\ \land x=x_0-y\ \}\quad y:=y+x \quad \{\ x_0=x_i\land x_0-y_0\geq 0\land x=x_0-y_0\land y=y_0+x\}}\ \textsc{Aff}}{\{\ x=x_i\land x-y\geq 0\ \}\quad x:=x-y;y:=y+x\quad \{\ x_0=x_i\land x_0-y\geq 0\land x=x_0-y_0\land y=y_0+x\ \}}$$

□

Solution de l'Exercice 8.18

Facile. □

Solution de l'Exercice 8.19

Pour cela on exhibe un arbre de déduction dont la racine est la racine voulue et les feuilles les prémisses voulues :

$$\textsc{Seq}\ \dfrac{\textsc{Aff}\ \dfrac{\{\ P\ \}\ i_1\ \{Q\} \quad \{\ Q\ \}\ i_2\ \{R\}}{\{P\}\ i_2;\ i_2\ \{R\}} \qquad \{R\}\ i_3\ \{S\}}{\{P\}\ i_2;\ i_2;\ i_3\ \{S\}}$$

□

Solution de l'Exercice 8.20

$$\textsc{Conseq}\ \dfrac{\textsc{Seq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{0+x\geq 0\}\ a:=0;\ \{a=0\land 0+x\geq 0\}} \qquad \dfrac{}{\{a=0\land 0+x\geq 0\}\ b:=x\ \{b=x\land a=0\land 0+x\geq 0\}}\ \textsc{Aff}}{\{0+x\geq 0\}\ a:=0;\ b:=x\ \{b=x\land a=0\land 0+x\geq 0\}}}{\{0+x\geq 0\}\ a:=0;\ b:=x\ \{a+b\geq 0\}}$$

□

Solution de l'Exercice 8.21

$$\textsc{Conseq}\ \dfrac{\textsc{Seq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{\ x=x_i\land x-y\geq 0\ \}\quad x:=x-y \quad \{\ x_0=x_i\land x_0-y\ \geq 0\ \land x=x_0-y\}} \qquad \dfrac{}{\{\ x_0=x_i\land x_0-y\ \geq 0\ \land x=x_0-y\ \}\quad y:=y+x \quad \{\ x_0=x_i\land x_0-y_0\geq 0\land x=x_0-y_0\land y=y_0+x\}}\ \textsc{Aff}}{\{\ x=x_i\land x-y\geq 0\ \}\quad x:=x-y;y:=y+x\quad \{\ x_0=x_i\land x_0-y\geq 0\land x=x_0-y_0\land y=y_0+x\ \}} \quad x_0=x_i\land x_0-y\geq 0\land x=x_0-y_0\land y=y_0+x\ \Rightarrow\ y=x_i\land x\ =\ x_i-\ y_i}{\{\ y=y_i\land x=x_i\land x-y\geq 0\ \}\quad x:=x-y;y:=y+x\quad \{\ y=x_i\land x\ =\ x_i-\ y_i\}}$$

□

Solution de l'Exercice 8.22

$$
\textsc{Cond}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{\mathtt{y}\geq 0 \wedge \mathtt{y}\geq 0\}\ \ \mathtt{x:=8}\ \ \{\mathtt{x}=8\}} \qquad \dfrac{\mathtt{x}\geq 0\wedge \mathtt{x}<0 \Rightarrow \mathit{false} \quad \{\mathit{false}\}\ \mathtt{x:=9}\ \{\mathit{false}\} \quad \mathit{false}\Rightarrow \mathtt{x}=8}{\{\mathtt{x}\geq 0\wedge \mathtt{x}<0\}\mathtt{x:=0}\{\mathtt{x}=8\}}\ \textsc{Conseq}}{\{\mathtt{x}\geq 0\}\ \ \mathbf{if}\ \mathtt{x\ >=\ 0}\ \mathbf{then}\ \mathtt{y:=8}\ \mathbf{else}\ \mathtt{y:=9}\ \ \{\mathtt{y}=8\}}
$$

□

Solution de l'Exercice 8.23

Facile. □

Solution de l'Exercice 8.24

La boucle ne terminera que si x est positif ou nul au départ. Ce qui n'empêche pas de prouver la correction partielle : x sera nul si la boucle s'arrête.

$$
\textsc{While}\ \dfrac{\textsc{Conseq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{\}\ \ \mathtt{x:=x-1}\ \ \{\mathtt{x=x_0-1}\}} \quad \mathtt{x=x_0-1}\Rightarrow \mathit{true}}{\{\}\ \ \mathtt{x:=x-1}\ \ \{\}\text{(ou enlève le } \mathit{true})}}{\{\}\ \ \mathbf{while}\ \mathtt{x<>0}\ \mathbf{do}\ \mathtt{x:=x-1}\ \mathbf{done}\ \ \{\mathtt{x=0}\}}
$$

□

Solution de l'Exercice 8.25

Rien car il n'y a pas de while ni de fonction récursive. □

Solution de l'Exercice 8.26

On prend $E = x$

$$
\textsc{While}\ \dfrac{\textsc{Conseq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\begin{array}{c}\langle x\geq 0\wedge x>0\wedge x=n\rangle \\ x:=x-1 \\ \langle x_0\geq 0\wedge x_0>0\wedge x_0=n\wedge x=x_0-1\rangle\end{array}}}{\langle x\geq 0\wedge x>0\wedge x=n\rangle\ x:=x-1\ \langle x\geq 0\wedge x<n\rangle} \quad x\geq 0\wedge x>0\Rightarrow x\geq 0}{\langle x\geq 0\rangle\ \mathbf{while}\ x>0\ \mathbf{do}\ x:=x-1\ \mathbf{end}\ \langle x=0\rangle}
$$

□

Solution de l'Exercice 8.27

On prend :

- $P =$ y>0
- $B =$ y$\leq$ r
- $E =$r

– C =`r:=r-y; q=q+1`

□

Solution de l'Exercice 8.28

Première solution, sans tester les bornes (on considère juste les `t[i]` comme des variables).

```
Aff ─────────────────────────────────────────────────────────────
        {i<n ∧ n=|t| ∧ ∀0≤x<i,t[i]≠m ∧ i≤n}  i:=i+1
        {i=i_0+1 ∧ i_0<n ∧ n=|t| ∧ ∀0≤x<i_0,t[x]≠m ∧ i_0≤n}
While ───────────────────────────────────────────────────────────
        {n=|t| ∧ ∀0≤x<i,t[x]≠m ∧ i≤n}  while i<n and t[i]<>m do i:=i+1 done
        {(i≥n ∨ t[i]=m) ∧ i≤n ∧ n=|t| ∧ ∀0≤x<i,t[x]≠m ∧ i≤n}
Conseq ──────────────────────────────────────────────────────────
        ...{n=|t|∧i=0}while i<n and t[i]<>m do i:=i+1 done{t[i]=m ∨ i=n ∧ ∀j, t[j]≠m}
Seq ─────────────────────────────────────────────────────────────
        {n=|t|}  i:=0; while i<n and t[i]<>m do i:=i+1; done
        {t[i]=m ∨ i=n ∧ ∀j, t[j]≠m}
```

□

Solution de l'Exercice 10.34

Facile. □

Solution de l'Exercice 10.35

```
/*@ ensures
  @   \result >= x && \result >= y &&
  @   ∀ int z; z >= x && z >= y => z >= \result
  @*/
int max(int x, int y) {
  if (x > y) return x; else return y;
}




/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" max.c ; make -f max.makefile coq.dep
*** End: ***
*/
```

□

Solution de l'Exercice 10.36

□

Solution de l'Exercice 10.37

```
int a;
int b;

/*@ requires y>0 && x >= 0
  @ ensures a*y+b==x && b >= 0 && b <= y @*/



int divide (int x, int y){
  a=0;
  b=x;

  /*@ invariant ( a ) * y + ( b ) == x && b >= 0
          variant b - y + 1 @*/
  while (b>=y)
          {
                b = b - y;
                a = a + 1;
          }
}



/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" divide.c ; make -f divide.makefile c
*** End: ***
*/
```

□