Université Paris Diderot

■ ■ ■

Cours, TD et TP

de preuves de programmes

Tester un programme peut démontrer la présence d'un bug, jamais son absence.

Dijkstra

■ ■ ■

C. Casson

Table des matières

■ Chapitre 1 ■

Introduction à la preuve de programmme

I - Informations pratiques

Période de cours : 7 Janvier 2013 au 22 mars 2013

Cours : Jeudi 9h-11h — Salle 2027
Christine Tasson (`christine.tasson@pps.univ-paris-diderot.fr`)

TD/TPs : Jeudi 9h-13h salle 2001/2027

Évaluation : 10% Partiel (7 février)+ 50% Projet (Février/mi-mars)+ 40% Examen (le 28 mars)

II - Qu'est-ce que la preuve de programme ?

II.1 - Problématique

Programmer sans erreur est une tâche difficile en raison de la taille des logiciels, du nombre de personnes impliquées dans leur confection et de leur historique. Pourtant cela est un enjeu fondamental pour les systèmes critiques (dans le domaine du médical, de l'aérospatial, des transports routiers et ferroviaires, du nucléaire...). La preuve de programme propose des outils semi-automatiques permettant de certifier la correction des programmes. Elle est aussi utile dans d'autres domaines moins critiques car elle permet de formaliser le cahier des charges d'un programme et de faire une implémentation la plus adéquate que possible.

Exemple 1 (Bugs célèbres (`http://www.cs.tau.ac.il/~nachumd/horror.html`)).

1962 Mariner 1

1985 Therac 25

1996 Ariane V. Vol 501.

2000 Radiothérapie à Panama.

etc...

Définition 1.

Un programme est *correct* s'il effectue sans se tromper la tâche qui lui est confiée et ce dans tous les cas possibles.

Définition 2.

La spécification d'un programme est la description sans ambiguïté de la tâche que doit effectuer un programme et des cas permis.

La spécification des programmes est un problème difficile. En effet, elle oblige à abstraire les propriétés d'un programme.

Une fois la spécification d'un programme établie, la preuve de la correction du programme vis-à-vis de sa spécification est un problème tout aussi difficile. En effet, une analyse exacte d'un programme est impossible comme le montre le théorème de Rice.

Théorème 1 (de Rice).

Toute propriété *extensionnelle* non triviale portant sur des programmes est indécidable.

On doit donc se contenter d'une analyse approchée des programmes et de ne vérifier que certaines propriétés.

Exemple 2 (Typage).

Le typage est un exemple de propriété approchée que l'on peut prouver sur les programmes. Il permet de prouver que les fonctions sont appliquées à des arguments compatibles.

II.2 - Quelles propriétés prouver ?

Il y a plusieurs types de propriété que l'on veut prouver. D'une part, on souhaite prouver que le programme résout le problème que l'on s'est posé. D'autre part, on souhaite prouver qu'il termine sur toutes les entrées. Enfin, on peut ajouter à ces requêtes toute une série d'erreurs que le programme ne doit pas produire à l'exécution : pas de débordement arithmétique, pas de débordement de tableau, pas de débordement de pile, pas de déréférencement de pointeur `null`, absence de deadlocks. Toutes ces propriétés peuvent entrer dans la spécification du programme.

II.3 - Quelles méthodes d'analyse des programmes ?

L'analyse dynamique. Elle est la plus répandue. Elle consiste à exécuter le code ou à le simuler en vue de faire apparaître d'éventuels bugs.

La *Méthode de tests* consiste à comparer le résultat d'un programme avec le résultat attendu. Pour que cette méthode soit efficace, il faut tester les différentes situations possibles. Il existe deux types de tests :

* les tests *fonctionnels* qui considèrent le programme comme une boîte noire et ne sont établis qu'à partir de la connaissance de la spécification du programme ;
* les tests *structurels* qui, à partir de la connaissance du programme, cherchent à exécuter toutes les parties du code.

Pour établir un plan de test, il faut énumérer les cas à tester et établir un test par cas.

Quand le programme est constitué de plusieurs modules, chacun doit être testé indépendamment avant de tester la globalité dans une série de tests dits d'*intégration*.

Il est important de souligner que les méthodes de tests aussi importantes soient elles ne sont pas (en général) exhaustive. On ne peut en effet que tester un nombre fini de valeurs. Elles ne constituent donc pas en général une preuve de la correction du programme.

L'analyse statique. Elle est utilisée surtout dans le développement de logiciels critiques (par exemple des systèmes embarqués). Elle consiste à parcourir le texte du code sans l'exécuter afin de prouver certaines propriétés.

Il existe différentes méthodes d'analyse statique :

* Le *Model Checking* part d'une représentation *finie* du système, une abstraction, et s'en sert pour *vérifier* les propriétés voulues.
* L'*interprétation abstraite* permet de calculer les intervalles dans lesquels les variables évolueront au cours de l'exécution du code. Elle est utilisée dans l'aéronautique.
* Les méthodes par *raffinement* comme la *méthode B* partent de la spécification d'un problème et implémente de façon de plus en plus précise le programme jusqu'à obtenir un code exécutable. Chaque étape du raffinement est prouvée correcte. Elle est utilisée notamment dans le métro Météor (ligne 14).
* En *logique de Hoare*, la spécification d'un programme est vue comme un théorème. Ce formalisme permet alors de prouver cette spécification à l'aide d'un système de déduction.
* La *programmation certifiée* repose sur la correspondance entre preuves mathématiques et programmes. À la spécification d'un programme est associée une formule logique (un théorème). À partir d'une preuve de cette formule, on extrait un programme et un certificat de ce programme.

III - Preuves sur papier

Nous allons passer en revue les méthodes qui permettent de prouver la correction de petits programmes sur papier. Ce sont les propriétés de terminaison et de correction vis-à-vis d'une spécification qui nous intéresse ici, nous laissons de côté les erreurs à l'exécution.

III.1 - Programmes impératifs (Invariants et terminaison)

La programmation impérative repose sur l'utilisation de boucles (`for` ou `while`) dont il faut démonter l'effet et la terminaison.

Pour montrer l'effet d'une boucle sur les variables d'un programme, on a recourt à un *invariant* de boucle. C'est-à-dire une expression mathématique reliant les variables du programme et qui est vérifiée avant l'entrée dans la boucle et à chaque passage dans celle-ci.

Pour démontrer qu'une boucle termine, on utilise la propriété suivante : « toute suite décroissante d'entier est finie ».

Exercice 1.
1. Écrire un programme impératif prenant en entrée un entier n et permettant de calculer la somme des n premiers entiers.
2. Prouver la correction du programme et sa terminaison.

Exercice 2. On considère le programme Caml suivant :

```
let f n=
    let x= ref 0 and y = ref n in
    while (!y <> 0) do
       x := !x + 3;
       y := !y - 1;
       done;
    !x;;
```

1. Donner une spécification du programme.
2. Prouver la correction et la terminaison du programme.

Exercice 3. On cherche à calculer la somme de deux polynômes représentés par des tableaux. Par exemple, $X^5 + 3X^4 + 5$ est représenté par le tableau $5, 0, 0, 0, 3, 1$.
1. Écrire une spécification du problème.
2. Écrire un programme solution.
3. Prouver la correction du programme par rapport à la spécification du problème.

Exercice 4. On cherche à déterminer l'élément minimum d'un tableau.
1. Écrire une spécification du problème.
2. Écrire un programme solution.
3. Prouver la correction du programme par rapport à la spécification.

III.2 - Programmes récursifs (Principe d'induction)

En programmation fonctionnelle, on fait souvent appel à des *fonctions récursives*, c'est-à-dire des fonctions qui dans leur définition s'appellent.

Pour prouver ce que fait un programme récursif, on fait en général appel au *principe de récurrence*.

Définition 3 (Principe de récurrence).

Étant donnée une propriété P sur les entiers, la formule suivante est vérifiée :

$$[P(0) \wedge (\forall n,\, P(n) \Rightarrow P(n+1))] \Rightarrow \forall n,\, P(n).$$

Pour démontrer qu'un programme termine, on vérifie que les appels récursifs se font sur des entiers de plus en plus petits et que $f(0)$ termine.

Exercice 5.
1. Écrire un programme récursif prenant en entrée un entier n et permettant de calculer la somme des n premiers entiers.
2. Prouver la correction du programme et sa terminaison.

Cependant, les programmes ne travaillent pas toujours avec les entiers. On peut généraliser le principe de récurrence aux ensembles bien fondés :

Définition 4 (Ensemble bien fondé).

Soit $(E, \leq)$ un ensemble ordonné. On dit que l'ordre $\leq$ est bien fondé lorsqu'il n'existe pas de suite infinie décroissante.

Exemple 3.

* Les entiers.
* L'ordre lexicographique sur un couple d'ensemble bien fondés.

Définition 5 (Principe d'induction).

Étant donnée une propriété P sur un ensemble bien fondé $(E, \leq)$, la formule suivante est vérifiée :

$$[(\forall a \in E,\, a \text{ minimal} \Rightarrow P(a)) \wedge \forall x \in E, (\forall y \in E,\, y \leq x \wedge P(y) \Rightarrow P(x))] \Rightarrow \forall x \in E,\, P(x).$$

Théorème 2 (Terminaison).

Soient $(E, \leq)$ un ensemble bien fondé et $f : E \to X$ telle que :

  * f fait un nombre fini d'appel à $f(y)$ avec $y < x$
  * pour tout x minimal, $f(x)$ termine

alors f termine.

Exercice 6. Considérer le programme suivant :

```
let rec f (x,y) =
  if x =0 || y=0 then 0
  else if x=1 then f(x,y-1)
       else f(x-1,y)
```

1. Est-ce que ce programme termine ? le prouver.
2. Que fait ce programme ? le prouver.

Exercice 7. Considérer le programme suivant :

```
let rec f = fun
     (0,p) -> 47
   | (n,p) -> f(n-1,f(n-1,p+7));;
```

1. Est-ce que ce programme termine ? le prouver.
2. Que fait ce programme ? le prouver.

Exercice 8. Considérer le programme suivant :

```
let rec f n =
   if (n > 100)
   then n-10 ;
   else f(f(n+11));;
```

1. Montrer que le programme termine.
2. Montrer que pour tout n, $f(n) = 91$.

Exercice 9. (Fonction de Morris) Considérer le programme suivant :

```
let rec f = fun
    (0,y) -> 0
  | (x,y) -> f(x-1,f(x,y))
```

1. Que calcule ce programme ? le prouver.
2. Est-ce que ce programme termine ? le prouver.

Exercice 10. (Somme de polynômes) On cherche à calculer la somme de deux polynômes représentées par des listes ordonnées de monômes de la forme (a, e) où a représente le coefficient et e représente l'exposant du monôme. Par exemple, $X^5 + 3X^4 + 5$ est représenté par la liste $[(5, 0), (3, 4), (1, 5)]$.
1. Écrire une spécification du problème.
2. Écrire un programme solution.
3. Prouver la correction du programme par rapport à la spécification du problème.

■ Chapitre 2 ■

Logique de Hoare

I - Installer et se documenter

Nous utiliserons la version `0.80` de `Why3`. Pour l'installer, rendez-vous sur la page :

`http://why3.lri.fr/`.

Nous utiliserons les prouveurs `CVC3` et `alt-ergo`.

Quelques éléments bibliographiques sur lesquels reposent ce chapitre :

1. l'article de référence *An axiomatic basis for computer programming* écrit par Hoare en 1969.
2. l'ouvrage *Cours et exercices corrigés d'algorithmique, vérifier, tester et concevoir des programmes en les modélisant* de Jacques Julliand.

En `Why3`, on peut prouver des programmes écrit dans le langage `caml`, mais il existe aussi des logiciels permettant d'utiliser `Why3` pour prouver des programmes écrits en `java` (`Krakatoa`) et en `C` (plugin `Jessie` de `Frama-C`).

II - Principes de la logique de Hoare

Pour raisonner sur les programmes, on a besoin de décrire les propriétés d'un *état* et son *évolution* au cours de l'exécution des instructions.

Triplets de Hoare. La logique de Hoare permet de *prouver* qu'en partant d'un état initial vérifiant certaines propriétés (décrites par la pré condition), en effectuant une série d'instructions, on obtient un état final vérifiant d'autres propriétés (décrites par la post condition).

Définition 1 (Pre (post) condition).

> C'est une proposition portant sur l'état de la mémoire et que l'on pense vérifié avant (après) l'exécution d'un fragment de code.

En logique de Hoare, on spécifie les programmes comme s'il s'agissait de *boîte noires* dont on ne peut que tester des propriétés et dont ne connaît pas les détails d'implémentation.

Définition 2 (Specification).

> Un programme est spécifié par une précondition et une post condition déterminant les cas dans lesquels le programme va être exécuté et son résultat.

Exercice 1.

1. Donner la spécification de la racine carré d'un flottant.
2. Donner la spécification de la racine carré entière d'un entier.
3. Donner la spécification du calcul du maximum d'un tableau.

Définition 3 (Triplet de Hoare).

> Un triplet de Hoare, noté $\{P\}C\{Q\}$ est la donnée d'une pré condition, d'un fragment de code et d'une post condition.

Intuitivement, si P est vrai à l'état initial *et* C termine, alors Q est vrai à l'état final. Plus précisément,

Définition 4 (Correction partielle).

> On dit qu'une formule $\{P\}\texttt{C}\{Q\}$ est valide, c'est-à-dire qu'un programme `C` est partiellement correct par rapport à une pré condition P et une post condititon Q lorsque :
> « Si pour tout état initial vérifiant P *et* si l'exécution termine alors l'état final vérifie Q.

Attention : la correction est partielle (on ne s'intéresse pas a priori à la terminaison).

Exercice 2. Quelles sont les triplets valides ? Justifier.

$$\begin{array}{ccc} \{x=2\} & \texttt{x:=x+1} & \{x=3\} \\ \{x=a+1\} & \texttt{x:=x+1} & \{x=a\} \\ \{x>2\} & \texttt{y:=x*(x+1)} & \{y>8\} \\ \{x=0\} & \texttt{while (x=0) do y:=2 done} & \{x=3\} \end{array}$$

Afin de formaliser les raisonnements intuitifs de l'exercice précédent, nous allons tout d'abord décrire l'exécution des programmes et leur effet sur la mémoire. Ensuite, nous introduirons un système de preuve qui nous permettra, à partir de règles de déduction élémentaires, de déduire qu'un code est correct sans l'exécuter.

Langage de programmation. La grammaire du langage est donnée par la définition des expressions et des commandes.

$$\begin{array}{rcl} e & ::= & \texttt{true} \mid \texttt{false} \mid n \mid x \mid e\,op\,e \\ op & ::= & + \mid - \mid * \mid = \mid \leq \mid and \mid not \\ s & ::= & \texttt{skip} \mid x := e \mid s; s \mid \texttt{if } e \texttt{ then } s \texttt{ else } s \mid \texttt{while } e \texttt{ do } s \end{array}$$

Sémantique opérationnelle. L'état d'un programme décrit le contenu des variables globales à un instant donné.

Définition 5.

L'état d'un programme est donné par une fonction Σ qui associe à chaque variable x sa *valeur* $\Sigma(x)$.

Grâce à cette description de l'état d'un programme, on peut calculer la *valeur* d'une expression dans un état Σ donné :

$$\begin{array}{rcl} \texttt{ev}_\Sigma(\texttt{true}) & = & \texttt{true} \\ \texttt{ev}_\Sigma(\texttt{false}) & = & \texttt{false} \\ \texttt{ev}_\Sigma(n) & = & \texttt{n} \\ \texttt{ev}_\Sigma(x) & = & \Sigma(x) \\ \texttt{ev}_\Sigma(e\,op\,e') & = & \texttt{ev}_\Sigma(e)\ [op]\ \texttt{ev}_\Sigma(e') \end{array}$$

où $[op]$ est l'interprétation mathématique de l'opérateur correspondant.

Maintenant que nous savons évaluer les expressions de notre langage, nous pouvons donner une description de sa *sémantique opérationnelle*. Celle-ci peut-être considérée comme la *compilation* du langage de programmation dans les mathématiques.

Nous adoptons les règles de la sémantique opérationnelle à *petits pas* où chaque pas de réduction est exécuté individuellement. Elle est définie par un système de déduction de jugements de la forme $\Sigma, s \rightsquigarrow \Sigma', s'$ dont la signification est : « après avoir exécuté un pas du code s en partant de l'état Σ, l'état de la mémoire est Σ' et il reste à exécuter le code s' ». Les règles de déduction de ce système sont :

$$\frac{}{\Sigma, x := e \rightsquigarrow \Sigma\{x \leftarrow \texttt{ev}_\Sigma(e)\}, \texttt{skip}}$$

$$\frac{}{\Sigma, (\texttt{skip}; s) \rightsquigarrow \Sigma, s} \qquad \frac{\Sigma, s_1 \rightsquigarrow \Sigma', s_1'}{\Sigma, (s_1; s_2) \rightsquigarrow \Sigma', (s_1'; s_2)}$$

$$\frac{\texttt{ev}_\Sigma(e) = \texttt{true}}{\Sigma, \texttt{if } e \texttt{ then } s_1 \texttt{else } s_2 \rightsquigarrow \Sigma, s_1} \qquad \frac{\texttt{ev}_\Sigma(e) = \texttt{false}}{\Sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \rightsquigarrow \Sigma, s_2}$$

$$\frac{\texttt{ev}_\Sigma(e) = \texttt{true}}{\Sigma, \texttt{while } e \texttt{ do } s \rightsquigarrow \Sigma, (s; \texttt{while } e \texttt{ do } s)} \qquad \frac{\texttt{ev}_\Sigma(e) = \texttt{false}}{\Sigma, \texttt{while } e \texttt{ do } s \rightsquigarrow \Sigma, \texttt{skip}}$$

Spécification des programmes. Afin de décrire la spécification des programmes en logique de Hoare on aura besoin des prédicats du premier ordre.

Définition 6.

La logique des prédicats du premier ordre est définie par :

Le langage donné par :

- un ensemble infini de variables : $x, y, z, \ldots$
- un ensemble de constantes : $a, b, c, \ldots$
- un ensemble de prédicats : $P, Q, R, \ldots$
- des connecteurs : $\vee, \wedge, \neg, \Rightarrow$
- des quantificateurs sur les variables : $\forall x, \exists x$

Les formules : Si e, e_1, e_2 sont des formules, alors

- $P(x_1, \ldots, x_n)$ est une formule
- $\neg e, e_1 \vee e_2, e_1 \wedge e_2, e_1 \rightarrow e_2$ sont des formules
- $\forall x.e$ et $\exists x.e$ sont des formules.

Exemple 1.

$$\exists y[P(y) \Rightarrow \forall x.P(x)]$$
$$[P(0) \wedge (\forall n\, P(n) \Rightarrow P(n+1))] \Rightarrow \forall n P(n)$$

Soit P un prédicat portant sur les variables d'un programme. On dit que l'état Σ satisfait P lorsque la formule $[P]_\Sigma$, obtenue en remplaçant dans P les variables x par leur valeur $\Sigma(x)$, est valide.

Définition 7.

Un triplet de Hoare $\{P\}s\{Q\}$ est valide lorsque pour tous états Σ et Σ' tels que $\Sigma, s \rightsquigarrow \Sigma', \texttt{skip}$, la validité de $[P]_\Sigma$ implique la validité de $[Q]_{\Sigma'}$.

Exercice 3. Reprendre l'exercice précédent et formaliser les démonstrations.

III - Preuves en logique de Hoare

Afin de prouver la validité des triplets de Hoare, on peut utiliser un système de preuve *compositionnel* (la preuve de la spécification d'un code se ramène à la preuve de la spécification des parties de ce code).

III.1 - Règles de la logique de Hoare

Affectation.

La règle d'affectation signifie intuitivement que si on veut que P soit vraie après l'affectation, c'est-à-dire dans l'état où seul x a été changée en E, on doit vérifier que $P[x := E]$ (c'est-à-dire P dans laquelle on a remplacé x par E) est vraie avant l'affectation :

$$\frac{}{\{P[x := E]\}\ x\ :=\ E\ \{P\}}$$

Remarquez que cette règle n'a pas d'hypothèse, c'est un axiome.

Exercice 4.

1. Les triplets suivants sont-ils dérivables ?

$$\begin{array}{rcl} \{y > 0\} & \texttt{x:=y+3} & \{x > 3\} \\ \{x - y \geq 0\} & \texttt{y:=y+x} & \{2x \geq y\} \\ \{x = 4\} & \texttt{x:=x+1} & \{x + 1 = 4\} \end{array}$$

2. Trouver E telle que le triplet suivant soit dérivable :

$$\{E\} \quad \texttt{x:=x+b+1} \quad \{b = 2 \wedge x = y + b\}$$

Séquence.

La règle de la séquence est la suivante :

$$\frac{\{P\}\ C\ \{Q\} \qquad \{Q\}\ D\ \{R\}}{\{P\}\ C; D\ \{R\}}$$

Exercice 5.
1. Montrer que le triplet suivant est dérivable :

$$\{x > 2\} \quad \texttt{x:=x+1; x:=x+2} \quad \{x > 5\}$$

2. Trouver E telle que le triplet suivant soit dérivable :

$$\{E\} \quad \texttt{x:=x+1; x:=x*x} \quad \{x \geq 16\}$$

Conditionnelle.

Quand une condition est exécutée, les deux morceaux de code peuvent être exécutés mais elles le seront avec des pré conditions différentes. On doit donc prouver la post condition dans les deux cas :

$$\frac{\{E = \texttt{true} \ \wedge\ P\}\ C\ \{Q\} \qquad \{E = \texttt{false} \ \wedge\ P\}\ D\ \{Q\}}{\{P\}\ \texttt{if}\ E\ \texttt{then}\ C\ \texttt{else}\ D\ \{Q\}}$$

La valeur de E permet de choisir la branche.
Exercice 6. Prouver que le triplet suivant est dérivable :

$$\{x > 2\} \quad \texttt{if (x>2) then y:=1 else y:=-1} \quad \{y > 0\}$$

Boucles.

Pour prouver la spécification d'une boucle, on a besoin d'un invariant de boucle. On doit vérifier que lorsque l'invariant est vérifié au départ de la boucle et que la condition de boucle E est vraie, alors l'invariant est toujours vrai à la fin de la boucle. Lorsque l'on sort de la boucle, l'invariant est alors encore vrai et la condition de boucle n'est plus vérifiée :

$$\frac{\{E = \texttt{true} \ \wedge\ I\ \}\ C\ \{I\}}{\{I\}\ \texttt{while}\ E\ \texttt{do}\ C\ \texttt{done}\ \{E = \texttt{false} \ \wedge\ I\}}$$

Exercice 7. Trouver l'invariant qui permet de dériver le triplet :

$$\{E\} \quad \texttt{while}\ (n \geq 0)\ \texttt{do n:=n-1 done} \quad \{n = 0\}$$

Pre/Post.

Si on a plus d'information sur l'état de la mémoire avant exécution du code, alors on peut toujours obtenir le même résultat. Si on peut obtenir un résultat après exécution, on peut toujours obtenir un résultat plus faible.

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

On verra dans le paragraphe suivant que cette condition est très utile pour ajuster les triplets de Hoare à la spécification et pouvoir appliquer les règles précédentes.

III.2 - Correction de la logique

Lemme 1.

Pour toute exécution d'une séquence qui termine $\Sigma, (s_1; s_2) \rightsquigarrow^* \Sigma', \texttt{skip}$, il existe un état intermédiaire Σ'' tel que $\Sigma, s_1 \rightsquigarrow \Sigma'', \texttt{skip}$ et $\Sigma'', s_2 \rightsquigarrow \Sigma', \texttt{skip}$.

Démonstration. Par récurrence sur la longueur de l'exécution de la séquence. □

Théorème 1 (Validité de la logique de Hoare).

S'il existe un arbre de déduction complet ayant un triplet de Hoare à sa racine, alors le programme est correct par rapport à ses pré et post conditions.

Démonstration. La preuve se fait par récurrence sur la hauteur de l'arbre de dérivation. □

III.3 - Annoter les programmes

Pour prouver la correction d'un triplet de Hoare $\{P\}\texttt{code}\{Q\}$ il faudrait écrire un arbre de preuve qui peut prendre beaucoup de place. Une autre présentation consiste à annoter les programmes et à montrer la validité des formules résultant, comme dans l'exemple suivant.

Exemple 2.

Le programme suivant implémente le calcul de la factorielle :

$\{True\}$	$\{True\}$	$\{True\} \overset{①}{\Leftrightarrow} \{1 = 0!\}$
f_1 `i := 0;`	f_1 `i := 0;`	f_1 `i := 0;`
$\{e_1\}$	$\{I[r := 1]\}$	$\{1 = i!\}$
f_2 `r := 1;`	f_2 `r := 1;1`	f_2 `r := 1;`
$\{e_2\}$	$\{I\}$	$\{r = i!\}$
`while (i<>n) do`	`while (i<>n) do`	`while (i<>n) do`
$\{e_3\}$	$\{I \wedge i \neq n\}$	$\{r = i! \wedge i \neq n\}$
f_3 `i := i+1;`	f_3 `i := i+1;`	$\overset{②}{\Rightarrow} \{r*(i+1) = (i+1)!\}$
$\{e_4\}$	$\{I[r = r * i]\}$	f_3 `i := i+1;`
f_4 `r := r*i;`	f_4 `r := r*i;`	$\{r * i = i!\}$
$\{e_5\}$	$\{I\}$	f_4 `r := r*i;`
f_5 `done;`	f_5 `done;`	$\{r = i!\}$
$\{e_6\}$	$\{r = n!\} \Leftrightarrow \{I \wedge i = n\}$	f_5 `done;`
f_6 `return r;`	f_6 `return r;`	$\{r = n!\} \overset{③}{\Leftrightarrow} \{r = i! \wedge i = n\}$
$\{r = n!\}$	$\{r = n!\}$	f_6 `return r;`
		$\{r = n!\}$

L'idée est de trouver les expressions e_i et de montrer la validité des formules $f_i : e_i$ `code` e_{i+1}.

En utilisant les règles de la logique de Hoare, on peut annoter de façon précise le programme (voir la deuxième colonne).

Pour l'invariant I, on propose $I \Leftrightarrow r = i!$. En utilisant plusieurs fois la règle d'affectation, on obtient alors le programme annoté de la troisième colonne.

Les conditions de bord ①, ② et ③ sont vérifiées à l'aide de calculs arithmétiques élémentaires.

Il nous reste à justifier de la validité des formules :

f_1, f_2, f_4 : règle d'affectation

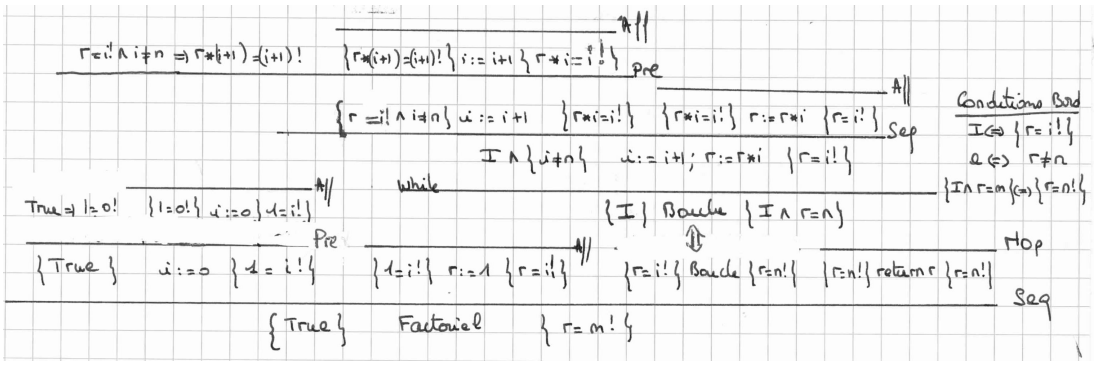
f_3 : règle d'affectation et Pre/Post

f_5 : règle de la boucle

f_6 : `return` n'affecte pas la mémoire

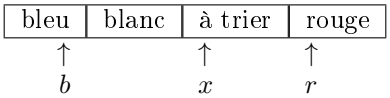
La composée des formules étant obtenue par la règle de la séquence.

Présenté sous la forme d'un arbre de preuve, on obtient :



Exercice 8.
1. Après avoir énoncer en logique de Hoare la spécification du programme, annoter le programme permettant de calculer la somme des n premiers entiers, puis prouver sa correction.
2. Après avoir énoncer en logique de Hoare la spécification du programme, annoter le programme permettant de calculer la racine carré entière d'un entier n, puis prouver sa correction.

Exercice 9. (Drapeaux de Dijkstra) Le but de cet exercice est de construire un programme permettant de trier un tableau contenant trois sortes de boules (bleu, blanc, rouge). On cherche à implémenter le schéma d'algorithme :



Au fur et à mesure du parcours du tableau, on place les boules dans l'ordre, la partie à trier diminuant.

On utilisera le schéma de programme suivant et on implémentera les parties manquantes en se laissant guider par la preuve du programme.

```
b := e1; x := e2; r := e3;
while (x <> r) do
   placer la boule de position x à sa position
   x := x+1
   done;
```

1. Donner une spécification du programme sous la forme d'un triplet de Hoare.
2. À partir du schéma précédant, formaliser l'invariant de l'algorithme.
3. Annoter le programme.
4. Déterminer les expressions e_1, e_2, e_3, pour que les formules engendrées par l'annotation soient valides.
5. Spécifier puis coder une fonction d'échange de deux éléments d'un tableau, notée `swap(i,j,t)`.
6. Déterminer le code de la fonction `placer` pour que les formules de Hoare soient valides.

III.4 - Preuves de terminaison

Jusqu'à présent, nous n'avons présenter que les preuves de correction partielles, c'est-à-dire en mettant de côté la terminaison.

Définition 8 (Correction totale).

On dit qu'un programme est totalement correct par rapport à une pré condition P et une post condititon Q lorsque :

« Si pour tout état initial vérifiant P *alors* l'exécution termine *et* l'état final vérifie Q.

On dit alors que le triplet $\langle P \rangle$ `C` $\langle Q \rangle$ est valide.

Pour prouver la correction totale, il faut utiliser une règle de la boucle un peu différente, qui fait intervenir la notion de *variant* (un entier positif qui décroît à chaque passage dans la boucle) :

$$\frac{\langle E = \texttt{true} \ \wedge \ I \ \wedge \ V = z \rangle \ C \ \langle I \ \wedge \ V < z \rangle \qquad I \ \Rightarrow \ V \geq 0}{\langle I \rangle \ \texttt{while} \ E \ \texttt{do} \ C \ \texttt{done} \ \langle E = \texttt{false} \ \wedge \ I \rangle}$$

Exercice 10. Donner une preuve de la correction totale des programmes permettant de calculer
* la factorielle
* la somme des n premiers entiers
* la racine carré entière d'un entier

IV - Calcul de plus faible pré condition

Rappelons que grâce à la règle `Pre`, si P' est une condition plus faible que P, c'est-à-dire que $P \Rightarrow P'$ et $\langle P' \rangle \texttt{C} \langle Q \rangle$ est un triplet valide, alors $\langle P \rangle \texttt{C} \langle Q \rangle$ est aussi un triplet valide.

Le problème est de trouver quelle est la pré condition P la plus faible pour que le triplet $\langle P \rangle \texttt{C} \langle Q \rangle$ soit valide.

Dijkstra introduit en 1976 le *calcul de plus faible pré condition* qui reformule les règles de la logique de Hoare afin de répondre au problème ci-dessus.

Définition 9.

Soient Q un prédicat sur la mémoire et C un morceau de code. On note $\mathrm{WP}(C, Q)$ la fonction de C et de Q qui calcule la plus faible pré condition telle que $\langle P \rangle$ `C` $\langle Q \rangle$. Elle est définie par induction sur le code :

$$\mathrm{WP}(\texttt{x:=E}, Q(\texttt{x})) = Q(\texttt{E})$$

$$\mathrm{WP}(\texttt{S}_1\texttt{;S}_2, Q) = \mathrm{WP}(\texttt{S}_1, \mathrm{WP}(\texttt{S}_2, Q))$$

$$\mathrm{WP}(\texttt{if E then C else D}, Q) = (\texttt{E} \wedge \mathrm{WP}(\texttt{C}, Q)) \vee (\neg E \wedge \mathrm{WP}(\texttt{D}, Q))$$

$$\mathrm{WP}(\texttt{while E do C done}\{\mathrm{inv}I, \mathrm{var}V\}, Q) = I$$

$$\text{avec les conditions de bord } \begin{cases} \texttt{E=true} \wedge I \wedge V = z \Rightarrow \mathrm{WP}(\texttt{C}, I \wedge V < z) \\ I \Rightarrow V \geq 0 \\ (\texttt{E=false} \wedge I) \Rightarrow Q \end{cases}$$

Exercice 11. Que signifient les équations suivantes :

1. $\mathrm{WP}(\texttt{C}, Q) = \texttt{True}$
2. $\mathrm{WP}(\texttt{C}, \texttt{True}) = \texttt{True}$
3. $\mathrm{WP}(\texttt{C}, Q) = \texttt{False}$
4. $\mathrm{WP}(\texttt{C}, \texttt{True}) = \texttt{False}$

Exercice 12. Calculer les plus faibles pré conditions suivantes :

1. $\mathrm{WP}(\texttt{x:=y+3}, x > 3)$
2. $\mathrm{WP}(\texttt{n:=n+1}, n > 4)$
3. $\mathrm{WP}(\texttt{y:=x+2;y:=y-2}, y > 5)$
4. $\mathrm{WP}(\texttt{if x>2 then y:=1 else y:=-1}, y > 0)$
5. $\mathrm{WP}(\texttt{while n<>0 do n:=n-1 done } \{I = n \geq 0, V = n\}, n = 0)$

V - Automatisation des preuves

Le calcul de plus faible pré condition permet d'automatiser en partie la preuve de programme. En effet, trouver les variants et invariants nécessite la compréhension de l'algorithme et les preuves des conditions de bords nécessitent parfois une aide humaine.

Principe de Why/Frama C

FramaC est un logiciel qui permet de faire de l'analyse statique de programmes écrits dans le langage C. Le module `Jessie` permet de faire de la preuve de programme annotés comme présenté dans ce chapitre. Il repose sur l'outil de preuve Why et permet en plus de vérifier l'absence de certains bugs classiques (dépassement arithmétique, division par zéro, déréférencement de pointeurs null,...).

Why implémente le calcul de plus faible pré condition pour un langage annoté et génère un ensemble d'obligations de preuves compatibles avec plusieurs assistants à la preuve. FramaC utilise Why en traduisant le langage C dans le langage *idéalisé* de Why.

Annotations et logique de Hoare

Le langage d'annotation des programmes que l'on utilisera est l'ACSL.

`requires`	Introduit une pré condition. `requires n>=0;`
`\result`	Représente le résultat du programme.
`ensures`	Introduit une post condition. `ensures \result == 0;`
`\validrange`	Prédicat assurant que les indices d'un tableau varient entre deux bornes. `\valid_range(t,0,n-1);`
`\forall`	Introduit une quantification universelle. `\forall integer x, y; x <= y ==> x <= (x+y)/2 <= y;`
`\exists`	Introduit une quantification existentielle. `\exists integer k; 0 <= k <= n-1 && t k == 0;`
`loop invariant`	Introduit un invariant de boucle. `loop invariant 0 <= x && y <= n-1;`
`loop variant`	Introduit un variant de boucle. `loop variant y-x;`

`lemma`	Introduit un lemme qui peut être utilisé dans la preuve de correction du programme. Et qui doit être vérifié par l'assistant de preuve ou par l'utilisateur.
	`lemma mean:`
	`  \forall integer x, y; x <= y ==> x <= (x+y)/2 <= y;`

Arithmétique

Les entiers machines étant codés en 32 ou 64 bits diffèrent des entiers mathématiques. FramaC prévoit de pouvoir tester la validité d'un algorithme avec les deux modèles d'entiers.

Ainsi, en ajoutant la ligne de commande `#pragma JessieIntegerModel(math)` on utilise les entiers mathématiques, sans cette ligne on utilise les entiers machines 32 bits.

Exemple 3 (Recherche du maximum dans un tableau).

Voici un programme annoté calculant l'élément maximum d'un tableau.

```
/*@ requires n >=1 && \valid_range(t, 0, n-1);
  @ ensures (\exists integer k; 0 <= k < n &&
  @ t[k] == \result) && (\forall integer j; 0 <= j < n ==> t[j]<=\result);
  @*/



int max(int t[], int n) {
  int max = t[0], i=0;
  /*@ loop invariant
    @ (0 <= i < n) &&
    @ (\exists integer k; 0<=k<=i &&  t[k]==max) &&
    @ (\forall integer j; 0 <= j <= i ==> t[j]<=max);
    @ loop variant n-i;
    @*/
 while (i+1 != n){
    i++;
    if (max <= t[i])
      max = t[i];
    }
 return max;
}
```

■ Chapitre 3 ■

L'assistant à la preuve Coq

I - Installer et se documenter

Nous utiliserons l'assistant de preuve Coq. Dans les systèmes Debian et Ubuntu, ce logiciel est distribué sous forme de paquet. Pour les autres systèmes il suffit de télécharger la dernière version disponible sur la page : http://coq.inria.fr/download. Nous recommandons d'utiliser `Emacs` avec le module `Proof General` qui facilite l'édition et l'exécution du code Coq, il est disponible à l'adresse :
http://proofgeneral.inf.ed.ac.uk/,
ou d'utiliser `coqide` qui est disponible sous forme de paquet debian.

Quelques éléments bibliographiques sur lesquels reposent ce cours :

1. l'ouvrage *The Coq Art* par Yves Bertot et Pierre Castéran, disponible à la page :
   http://www.labri.fr/perso/casteran/CoqArt/index.html.
2. le petit guide de survie en Coq par Alexandre Miquel :
   http://www.pps.jussieu.fr/~miquel/ens-0607/lc/guide.html.

II - Correspondance de Curry-Howard

Introduite dans les années 1960, la correspondance de Curry-Howard permet de faire le lien entre la logique et l'informatique en remarquant que les preuves des théorèmes sont des programmes. Plus précisément, on a la correspondance :

Logique	Informatique
Formule	Type
Preuve	Programme
Élimination des coupures	Calcul

Nous verrons par la suite que le logiciel Coq repose sur cette correspondance.

II.1 - Lambda-calcul simplement typé

On se donne un type de base c et on construit les types par la grammaire :

$$F \quad := \quad c \quad | \quad F \to F$$

où c est un (ou plusieurs) type(s) de base comme par exemple les types `bool` ou `nat`.

La flèche $F_1 \to F_2$ représente l'ensemble des fonctions prenant un argument de type F_1 et renvoyant un résultat de type F_2.

Le lambda-calcul simplement typé est construit à partir des termes donnés par la grammaire suivante :

$$s, t \quad := \quad x \quad | \quad \lambda x.s \quad | \quad (s)\ t$$

où x appartient à un ensemble de variables donnés.

Dans le lambda-calcul simplement typé, on ne considère que les termes *typables*. On dit qu'un terme s est de type F s'il existe un *jugement de typage* $\Gamma \vdash u : F$ dérivable en appliquant les règles de typage suivantes :

$$\frac{}{\Gamma,\ x : F \vdash x : F}\ \text{Var} \qquad \frac{\Gamma, x : F_1 \vdash s : F_2 \quad \text{où } x \notin \Gamma}{\Gamma \vdash \lambda x.s : F_1 \to F_2}\ \text{Abs} \qquad \frac{\Gamma \vdash s : F_1 \to F_2 \quad \Gamma \vdash t : F_1}{\Gamma \vdash (s)t : F_2}\ \text{App}$$

où Γ est un environnement de la forme $x : F_1, \ldots, x_n : F_n$ avec $x_i \neq x_j$ si $i \neq j$.

Pour calculer le résultat d'un terme, on utilise la règle de calcul appelée β-réduction :

$$(\lambda x.s)t \to s[x := t]$$

où $s[x := t]$ est la substitution dans s de toutes les occurrences de la variable x par le terme t.

Exercice 1. Pour chacun des termes suivants, donner dans le langage de programmation Caml un programme et un type correspondants, puis dériver une preuve de typage.

* L'identité : $\lambda x.x$
* L'évaluation : $\lambda af.(f)a$
* La projection : $\lambda ab.a$

II.2 - Logique minimale intuitionniste

La logique minimale est construite à partir de formules définies grâce à la grammaire suivante :

$$F \quad := \quad c \quad | \quad F \Rightarrow F$$

où c est une variable propositionnelle et $\Rightarrow$ est l'implication.

Les formules prouvables de la logique intuitionniste sont des jugements de la forme $\Gamma \vdash F$ (où Γ est une liste éventuellement vide de formule et F est une formule) qui sont obtenus comme conclusion de règles de déduction :

$$\frac{}{\Gamma,\ F \vdash F}\ \text{Ax} \qquad \frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \Rightarrow F_2}\ \Rightarrow\text{-intro} \qquad \frac{\Gamma \vdash F_1 \Rightarrow F_2 \quad \Gamma \vdash F_1}{\Gamma \vdash F_2}\ \Rightarrow\text{-elim}$$

II.3 - Correspondance de Curry-Howard

En effaçant les termes des dérivations de typages, on obtient des preuves de la logique minimale intuitionniste. On a donc une correspondance exacte entre les λ-termes et les preuves.

$$\frac{}{\Gamma,\ \mathtt{x} : F \vdash \mathtt{x} : F}\ \text{Var} \qquad \frac{\Gamma, \mathtt{x} : F_1 \vdash \mathtt{s} : F_2 \quad \text{où } x \notin \Gamma}{\Gamma \vdash \lambda \mathtt{x.s} : F_1 \Rightarrow F_2}\ \text{Abs} \qquad \frac{\Gamma \vdash \mathtt{s} : F_1 \Rightarrow F_2 \quad \Gamma \vdash \mathtt{t} : F_1}{\Gamma \vdash \mathtt{(s)t} : F_2}\ \text{App}$$

Un terme peut-être vu comme une preuve d'une formule (son type) et un théorème (une formule de la logique) est prouvable s'il existe un arbre de preuve et donc un terme typé par cette formule, « un habitant » de ce type.

Exercice 2. Reprendre les termes que vous avez typés à l'exercice précédent et les traduire dans le langage logique.

La correspondance de Curry-Howard s'étend à de nombreux systèmes logiques. Nous nous intéressons à la logique propositionnelle du premier ordre et à l'extension adéquate du λ-calcul.

Extension au type produit : La conjonction $A \wedge B$ est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim}$$

elle correspond au type produit :

$$\frac{\Gamma \vdash \mathtt{t} : A \quad \Gamma \vdash \mathtt{u} : B}{\Gamma \vdash \langle \mathtt{t,u} \rangle : A \times B}\ \text{Pair} \qquad \frac{\Gamma \vdash \mathtt{t} : A \times B}{\Gamma \vdash \pi_1 \mathtt{t} : A}\ \text{Proj-l} \qquad \frac{\Gamma \vdash \mathtt{t} : A \times B}{\Gamma \vdash \pi_2 \mathtt{t} : B}\ \text{Proj-r}$$

Extension au type somme : La disjonction $A \vee B$ est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-intro} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-intro} \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-elim}$$

elle correspond au type somme :

$$\frac{\Gamma \vdash \mathtt{t} : A}{\Gamma \vdash \mathtt{inl\ t} : A \vee B}\ \text{Emb-l} \qquad \frac{\Gamma \vdash \mathtt{t} : B}{\Gamma \vdash \mathtt{inr\ t} : A \vee B}\ \text{Emb-r} \qquad \frac{\Gamma \vdash \mathtt{t} : A \vee B \quad \Gamma, \mathtt{x} : A \vdash \mathtt{v} : C \quad \Gamma, \mathtt{y} : B \vdash \mathtt{w} : C}{\Gamma \vdash \mathtt{match\ t\ with\ inl\ x\ ->\ v\ |\ inr\ y\ ->\ w} : C}\ \text{Case}$$

Exercice 3. Pour chacune des formules suivantes, donner un arbre de preuve et le terme correspondant :

$$(P \wedge Q \Rightarrow R) \Rightarrow (P \Rightarrow (Q \Rightarrow R))$$
$$P \wedge Q \Rightarrow P$$
$$P \wedge Q \Rightarrow Q \wedge P$$
$$P \vee Q \Rightarrow Q \vee P$$

Extension à la quantification existentielle : La quantification existentielle est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A[x \leftarrow a]}{\Gamma \vdash \exists x.A}\ \exists\text{-intro} \qquad \frac{\Gamma \vdash \exists x.A \quad \Gamma, A \vdash B \quad \text{où } x \notin FV(\Gamma) \text{ et } x \notin FV(B)}{\Gamma \vdash B}\ \exists\text{-elim}$$

elle correspond à :

$$\frac{\Gamma \vdash \texttt{t} : A[x \leftarrow a]}{\Gamma \vdash \texttt{(a,t)} : \exists x.A}\ \text{Wit} \qquad \frac{\Gamma \vdash \texttt{c} : \exists x.A \quad \Gamma, A \vdash \texttt{u} : B \quad \text{où } x \notin FV(\Gamma) \text{ et } x \notin FV(B)}{\Gamma \vdash \texttt{let x=c in u} : B}\ \text{Let}$$

Extension à la quantification universelle : est introduite par les règles de déductions :

$$\frac{\Gamma \vdash A \quad \text{où } x \notin FV(\Gamma)}{\Gamma \vdash \forall x.A}\ \forall\text{-intro} \qquad \frac{\Gamma \vdash \forall x.A}{\Gamma \vdash A[x \leftarrow a]}\ \forall\text{-elim}$$

elle correspond à :

$$\frac{\Gamma \vdash \texttt{t} : A \quad \text{où } x \notin FV(\Gamma)}{\Gamma \vdash \lambda\texttt{x.t} : \forall x.A} \qquad \frac{\Gamma \vdash \texttt{t} : \forall x.A}{\Gamma \vdash \texttt{ta} : A[x \leftarrow a]}$$

Exercice 4. Pour chacune des formules suivantes, donner un arbre de preuve et le terme correspondant :

$$(\forall x, Px \Rightarrow Qx) \Rightarrow (\forall x, Px) \Rightarrow (\forall x, Qx).$$
$$(\exists x, Px \Rightarrow Qx) \Rightarrow (\forall x, Px) \Rightarrow (\exists x, Qx)$$

II.4 - L'assistant à la preuve Coq

Coq est un assistant à la preuve interactif. Il repose sur un langage de programmation fonctionnel (une extension du λ-calcul) et sur le calcul des constructions inductives (une extension de la logique intuitionniste).

En pratique, lorsque l'on prouve une proposition en Coq, on part d'un but (*Goal*) et d'hypothèses (*Assumptions*) et on construit l'arbre de preuve du jugement *Assumptions* $\vdash$ *Goal* à l'aide de tactiques qui transforment un but en une liste de buts à résoudre.

Les tactiques de Coq qui permettent de faire des preuves logiques correspondent aux règles de la logique intuitionniste :

Tactique	**Règle logique**
`Assumption, Apply H`	Axiome
`intro, intros`	$\Rightarrow,\forall,\neg$-intros
`apply`	$\Rightarrow,\forall$-elim
`split`	$\wedge,\top$-intro
`left, right`	$\vee$-intro left, right
`exists`	$\exists$-intro
`destruct`	$\wedge,\vee,\bot,\exists$-elim

Exercice 5. Pour chacune des formules suivantes, donner une preuve en logique intuitionniste et une preuve Coq. Regarder le programme correspondant en utilisant la commande `Print`.

```
Lemma and_to_imp: (P /\ Q -> R) -> (P -> (Q -> R)).
Lemma and_e_left : P /\ Q -> P.
Lemma and_sym : P/\Q -> Q/\P.
Lemma or_sym : P\/Q -> Q\/P.
```

Grâce à la correspondance de Curry-Howard, Coq permet à la fois de prouver des théorèmes (des formules logiques représentant des énoncés mathématiques) et de certifier les programmes.

Plus précisément, un programme bien typé est une preuve de la formule mathématique correspondant à son type. D'autre part, une formule est prouvée lorsqu'elle correspond à un type *habité*, c'est-à-dire qu'il existe un programme ayant ce type.

Ainsi, en Coq, prouver une proposition, un lemme, c'est construire son arbre de preuve.

```
Lemma lem_identity : A -> A.
   intros.
   assumption.
   Qed.
```

```
Lemma lem_identity : A -> A.
   exact (fun (a:A) => a).
   Qed.
```

Une fois la preuve faite, on fournit le λ-terme correspondant au noyau de Coq qui se charge de vérifier que le λ-terme est bien typé, c'est-à-dire que le terme prouve bien le lemme.

D'ailleurs, il est possible de prouver ce lemme en donnant directement le programme correspondant à son arbre de preuve.

Exercice 6. Prouver la formule suivante en fournissant un terme dont le type correspond à la formule.

```
Lemma evaluation : A -> (A -> B) -> B.
```

Réciproquement, il est possible de définir un terme en donnant l'arbre de preuve correspondant à sa dérivation de typage.

Par exemple la première projection peut-être définie par un terme ou un arbre de preuve :

```
Definition fst_proj :
 nat -> nat -> nat.
   intros n m.
   apply n.
   Defined.
```

```
Definition fst_proj :
  nat -> nat -> nat :=
    fun a _ => a.
```

Exercice 7. Définir le programme correspondant à la seconde projection en donnant la preuve de son type.

```
Definition snd_proj : nat -> nat -> nat.
```

Pour finir, quelques mots clefs du langage de Coq à retenir :

`Lemma`, `Proposition`, `Theorem : type.`	Introduisent des formules à prouver
`Qed.`	Termine une preuve
`Definition cst : type := terme.`	Introduit une constante en donnant son terme
`Check t.`	Donne le type d'un terme, la formule prouvée
`Print t.`	Donne le terme et le type correspondant à une preuve

III - Les inductifs

Cette partie du cours est illustrée par le fichier `coq_inductive.v`.

III.1 - Types inductifs

Afin de construire des types de donnés, on utilise l'instruction `Inductive`. Elle permet d'énumérer les termes (programmes) de base habitant ce types.

Exemple 1.

Le type des booléens est construit de cette façon :

```
Inductive bool:Set := true: bool | false: bool
```

Il existe donc deux valeurs de type `bool` : `true` et `false`.

Ce type peut-être utilisé pour définir des programmes à l'aide de définitions par cas (*pattern matching*).

Exemple 2.

```
Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true => b2
  | false => false
  end.
```

Enfin, pour tout type énuméré, la tactique `destruct` permet de raisonner par cas.

Exemple 3.

« Pour prouver un prédicat sur les booléens, il suffit de le vérifier pour `true` et `false` ».

```
Check bool_rect.
 bool_rect: forall P: bool -> Prop,
        P true => P false => forall b:bool, P b
```

En pratique, on peut utiliser ce théorème comme dans la preuve suivante :

```
Lemma no_other_bool :
 forall b, b = true ⋁ b = false.
Proof.
 intro.
 destruct b.
 (* premier cas: b=true *)
 left. reflexivity.
 (* deuxième cas: b=true *)
 right; reflexivity.
Qed.
```

Exercice 8. Définir le type `option_nat` permettant de construire des programmes de type `nat` lorsqu'ils sont définis.

L'instruction `Inductive` permet de construire des types plus complexes que les types énumérés : les types *inductifs* dont les constructeurs de termes peuvent faire appel à d'autres termes comme dans l'exemple des entiers.

Exemple 4.

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat
```

Il y a deux façons de construire un terme de type `nat` : 0 est de type `nat` ou à partir d'un terme `n` de type `nat` et du constructeur successeur `S` on obtient un terme `S n` de type `nat`.

Comme pour les types énumérés, les types inductifs permettent des définitions par cas.

Exemple 5.

Ainsi, le type `nat` peut être utilisé pour construire des programmes récursifs

```
Fixpoint fact (n:nat) : nat :=
   match n with
   | 0 => 1
   | S u => (S u)*(fact u)
   end.
```

ou pas :

```
Definition pred (n:nat) : nat :=
   match n with
   | 0 => 0
   | S u => u
   end.
```

Tout type inductif vient avec un *principe d'induction*. On utilise ce principe dans une preuve via la tactique `induction`.

Exemple 6.

Le principe d'induction se ramène au principe de récurrence sur les entiers : « pour prouver un prédicat sur les entiers, il suffit de le montrer pour le cas de base 0 et de montrer qu'il est stable par le constructeur `S` (s'il est vrai pour `n` alors il est vrai pour `S n`).

```
Check nat_rect.
 nat_rect : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Exercice 9. Définir le type des listes d'entiers, puis le programme permettant de calculer la longueur d'une liste d'entiers. Définir le programme permettant de concaténer deux listes et montrer que la longueur de la liste concaténée est égale à la somme des longueurs des deux listes.

Cet exercices est illustré par le fichier `lists.v`.

III.2 - Prédicats inductifs

Cette partie du cours est illustrée par le fichier `lesson_ind_predicate.v`.

Rappelons qu'un *prédicat* est une proposition portant sur une ou plusieurs variables.

Exemple 7.

Par exemple, le prédicat « être dans l'ordre lexicographique » porte sur deux couples d'entier. En Coq, il prendra la forme :

```
Check lexico.
lexico : nat*nat -> nat*nat -> Prop
```

Le prédicat « être pair » porte sur les entiers. En Coq, il prendra la forme :

```
Check even.
even : nat -> Prop
```

Un prédicat peut-être défini ou bien de manière directe, en utilisant la construction `Definition` ou la construction `Fixpoint` lorsque la définition est récursive.

Exemple 8.

```
Definition lexico (p q :nat*nat)  : Prop :=
   (fst p < fst q) \/ (fst p = fst q /\ snd p < snd q).

Fixpoint Even (n:nat) : Prop :=
  match n with
  | 0 => True
  | S 0 => False
  | S (S n') => Even n'
  end.
```

Mais on peut aussi utiliser une définition *inductive*. Dans ce cas, on donne les *constructeurs* élémentaires de preuve des formules construites à partir de ce prédicat.

Exemple 9.

```
Inductive lex : nat*nat -> nat*nat -> Prop :=
  | lex_fst : forall a a' b b',  a < a' -> lex_lt (a,b) (a',b')
  | lex_snd : forall a b b',  b < b' -> lex_lt (a,b) (a,b').
```

Il y a deux constructeurs de preuves de l'ordre lexicographique. Le premier `lex` construit une preuve de `lex (a,b) (a',b')` à partir d'une preuve de `a < a'`. Le second construit une preuve de `lex (a,b)  (a,b')` à partir d'une preuve de `b < b'` (remarquez que les deux premières composantes sont identiques dans ce cas).

```
Inductive even : nat -> Prop :=
 | even_0 : even 0
 | even_SS : forall n, even n -> even (S (S n)).
```

Il y a deux constructeurs de preuves de la formule `even n`. La première, lorsque $n = 0$ on a, par définition du prédicat `even` la preuve `even_0` de `even 0`. La seconde, lorsque n est de la forme $S(Sn')$, le constructeur `even_SS` permet d'obtenir une preuve de `even S(S n)` à partir d'une preuve de `even n`.

Les deux manières de définir les prédicats sont équivalentes, mais elles servent dans des cas différents selon le contexte où ils sont utilisés. Par exemple, lorsque l'on veut faire un calcule, on utilisera une définition directe, alors que l'on veut faire un preuve on utilisera une définition inductive.

Exemple 10.

Il est plus facile de montrer que 100 est pair en utilisant la première définition :

```
Lemma even_100 : Even 100.
 simpl. auto.
Qed.
```

Il est plus facile de montrer le lemme suivant en utilisant la définition inductive car elle vient avec un principe d'induction qui permet de retrouver les cas permettant de construire une preuve du prédicat :

```
Check even_ind.
 even_ind : forall P : nat -> Prop,
       P 0 ->
       (forall n : nat, even n -> P n -> P (S (S n))) ->
       forall n : nat, even n -> P n

Lemma even_double :
 forall n, ev n -> exists m, m+m=n.
Proof.
 intros n H.
 induction H.
 (* La preuve H:ev n provient du constructeur ev_0 et n=0*)
  exists 0; reflexivity.
 (* La preuve H:ev n provient du constructeur plus_2_ev à partir d'une
 preuve IHev:exists m, m+m=n' avec n= S S n'*)
  destruct IHev as [m].
  exists (S m).
  omega.
Qed.
```

En pratique, on donne souvent les deux types de définition, on prouve qu'elle sont équivalentes et on utilise l'une ou l'autre selon les cas.

Exemple 11.

```
Lemma lex_equiv : forall p q,
 lexico p q <-> lex p q.
Proof.
 intros (a,b) (a',b').
 split.
 intro H. destruct H. simpl in H.
 apply lex_fst.  assumption.
 simpl in H. destruct H. rewrite <- H.
 apply lex_snd.  apply H0.
 intro H. inversion H.
 left. auto.
 right. auto.
Qed.
```

Les deux exemples précédents utilisent des tactiques qui permettent de déconstruire les prédicats. La première tactique permet de faire une étude par cas : `inversion H` avec `H:lex (a, b) (a', b')`. La seconde tactique permet de faire une preuve par induction `induction  H` où `H:even  n`, elle utilise le principe d'induction décrit précédemment.

Exercice 10.

1. Donner une définition directe et une définition inductive du prédicat `sorted` « être trié » sur les listes.

2. Montrer que les deux définitions sont équivalentes.

3. Montrer le théorème `sorted le (1::2::3::nil).`

4. Montrer le théorème `~(sorted le (1::3::2::nil)).`, on pourra introduire les lemmes suivant : `forall (A:Set) (R: A -> A-> Prop) (x:A) (l: list A), sorted R (cons x l) -> sorted R l.` et `forall (A:Set) (R: A -> A-> Prop) (x y :A) (l:list A), sorted R (cons x (cons y l)) -> R x y.`

Exercice 11. À l'aide de la commande `Print` comprendre comment sont construits les prédicats `True` et `False`.

IV - Spécification et certification des programmes en Coq

IV.1 - Preuve de programme, un exemple

Cette partie du cours est illustrée par le fichier `binaire.v`

Nous allons détailler l'exemple des *arbres binaires de recherche* afin de montrer comment utiliser Coq pour construire des programmes certifiés.

Les arbres binaires de recherche forment une structure de données permettant de représenter les ensembles avec une fonction de recherche efficace.

On considère les arbres binaires de recherche sur les entiers. Ce sont des arbres binaires vérifiant la structure suivante : s'il n'est pas vide, l'arbre possède une racine dont l'étiquette est plus grande que les racines du sous-arbre gauche et plus petite que celles du sous-arbre droit et les deux sous-arbres sont eux mêmes des arbres binaires de recherche.

Pour représenter les arbres binaires de recherche en Coq, on commence par définir le *type* inductif des arbres binaires :

```
Inductive tree : Set :=
| Empty
| Node : tree -> nat -> tree -> tree.
```

Ensuite, on définit un *prédicat inductif* vérifiant qu'un arbre binaire est un arbre binaire de recherche.

```
Inductive bst : tree -> Prop :=
| bst_empty : bst Empty
| bst_node : forall x l r, bst l -> bst r ->
                  (forall y, In y l -> y < x) ->
                  (forall z, In z r -> x < z) -> bst (Node l x r).
```

On veut définir une fonction `search x t` de recherche qui retourne le booléen `true` si et seulement si `x` appartient à l'arbre binaire de recherche `t` en tirant partie de la structure particulière de l'arbre.

Pour pouvoir énoncer le théorème de correction de l'arbre, il nous faut introduire le prédicat d'appartenance :

```
Inductive In (n:nat) : tree -> Prop :=
| Inleft : forall l x r, (In n l) -> In n (Node l x r)
| Inright : forall l x r, (In n r) -> In n (Node l x r)
| Inroot : forall l r, In n (Node l n r).
```

On peut alors énoncer la spécification de la fonction `search`

```
Theorem search_correct: forall (n:nat) (t:tree), bst t ->  (search n t = true <-> In n t).
```

Définissons maintenant cette fonction :

```
Fixpoint search (n:nat) (t:tree) :=
match t with
 | Empty => false
 | Node l x r => match nat_compare n x with
                     | Lt => search n l
                     | Eq => true
                     | Gt => search n r
                 end
end.
```

Exercice 12. Prouver en Coq le théorème de correction.

Enfin, on peut extraire le programme `Caml` associé à cette fonction à l'aide de la ligne de commande Coq

```
Extraction "search.ml" search.
```

On obtient le code Caml suivant :

```
type bool =
| True
| False

type nat =
| O
| S of nat

type comparison =
| Eq
| Lt
| Gt

(** val nat_compare : nat -> nat -> comparison **)

let rec nat_compare n m =
  match n with
  | O ->
    (match m with
     | O -> Eq
     | S n0 -> Lt)
  | S n' ->
    (match m with
     | O -> Gt
     | S m' -> nat_compare n' m')

type tree =
| Empty
| Node of tree * nat * tree

(** val search : nat -> tree -> bool **)

let rec search n = function
| Empty -> False
| Node (l, x, r) ->
  (match nat_compare n x with
   | Eq -> True
   | Lt -> search n l
   | Gt -> search n r)
```

IV.2 - Spécifier les fonctions partielles

Cette partie du cours est illustrée par les fichiers `facto.v` et `pred.v`

Représenter un programme par un terme en Coq est parfois difficile, parfois même impossible. En effet, tous les termes doivent définir des calculs qui terminent.

Exemple 12.

Le programme factoriel en Caml peut être défini par :

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact(n-1);;
```

Ce programme ne termine pas toujours. En effet, si $n \leq 0$, alors elle boucle.

Il existe plusieurs solutions pour contourner cette difficulté.

On peut utiliser un prédicat caractérisant la relation entre un argument et le résultat d'un programme.

Exemple 13.

On définit le prédicat `Pfact n m` qui est vrai lorsque le calcule de la factorielle de `n` termine et

vaut `m` :

```
Require Import ZArith.
Open Scope Z_scope.

Inductive Pfact : Z -> Z -> Prop :=
  | Pfact_0 : Pfact 0 1
  | Pfact_h : forall n v : Z, n <> 0%Z -> Pfact (n-1) v -> Pfact n (n*v).
```

Exercice 13. Montrer que le domaine de définition de la factorielle est l'ensemble des entiers positifs, c'est-à-dire la proposition : `forall n v : Z, Pfact n v -> 0 <= n.` Il est possible de prouver le contraire en utilisant les ordres bien fondés.

On peut utiliser le type `option`.

```
Inductive option (A : Type) : Type :=
    Some : A -> option A
  | None : option A
```

Exemple 14.

On peut définir une fonction partielle représentant le prédecesseur :

```
Definition pred (n:nat) : option nat :=
  match n with
      0 -> None
    |S n -> Some n
  end.
```

IV.3 - Spécifier avec les types

Les types permettent de donner des spécifications plus ou moins fortes des programmes.

Exemple 15.

* Un entier premier et plus grand que 5 est de type `nat`
* Une fonction de tri sur les entiers peut être typée par : `nat -> nat`
* La fonction prédecesseur peut être typée par : `nat -> option nat`

Les spécifications de l'exemple précédent n'apportent pas beaucoup d'information sur le résultat si ce n'est qu'il est bien défini.

Afin de définir des spécifications plus précises, on introduit des nouveaux types.

Exemple 16.

En Coq, le type `{p:nat | 5<p /\ is_prime p}` est le sous-ensemble des entiers qui vérifient les deux propositions « être plus grand que 5 » et « être premier ».

Exercice 14.

1. Quel serait le type de la fonction de tri sur les entiers ?

2. Quel serait le type pour la division euclidienne ?

Le constructeur de type `{x:A | P x}` se rapproche de l'existentiel : un programme de ce type — c'est-à-dire cette spécification devra fournir un témoin de l'existence de `x:a` et un certificat (une preuve) de `P x`.

coq_inductive.v

```
Require Import Arith List.

(* 1 Basic usage of inductive types. *)

Print bool.

(* case definition *)
Check andb.

Definition andb' (b1:bool) (b2:bool) : bool :=
  match b1 with
    | true => b2
    | false => false
  end.
Print andb'.

(* inductive principle *)
Check bool_rect.
Print bool_rect.

Definition bool_r (P:bool->Type) (f_0: P true) (f_1: P false) (b:bool): P b :=
  match b with
    |true => f_0
    |false => f_1
  end.
Print bool_r.

(* elim/induction : direct use of inductive principle  *)

Lemma no_other_bool :
 forall b, b = true \/ b = false.
Proof.
  destruct b.
  left. reflexivity.
  right; reflexivity.
Qed.
Print no_other_bool.

(* case : simple pattern matching *)

Lemma no_other_bool' : forall b, b = true \/ b = false.
Proof.
  intro b.
  case b.
  left; reflexivity.
  right; reflexivity.
Qed.
Print no_other_bool'.

Lemma case_danger : forall b, b = true -> b = true.
Proof.
  intros.
  (*case b.*) (*Does not work ... for that use destruct*)
  assumption.
Qed.
```

```
Print option.
Lemma inverse_option :
  forall A:Set, forall a b:A,
  Some a = Some b -> a = b.
Proof.
  intros.
  (* injection H. *)
  (* how it works : *)
  set (phi := fun o =>
    match o with
      | Some x => x
      | None => a
    end).
  change (phi (Some a) = phi (Some b)).
  rewrite H.
  reflexivity.
Qed.

(* NB: This kind of result is only true with inductive objects :
       in general we don't have f x = f y  -> x = y. *)
(* NB: But the reverse fact is general : x = y -> f x = f y.
   See rewrite ... *)

(* la tactique discriminate *)

Inductive day:= lun | mar | mer | jeu | ven | sam | dim.

Lemma dd : lun <> mar.
Proof.
(* discriminate. *)
  intuition.
  change ((fun d:day => match d with |lun => True | _ => False end) mar).
  rewrite <- H. trivial.
Qed.

(* 2 Really recursive Types *)

Print nat.
Check nat_rect.
Print nat_rect.

Fixpoint rec (P : nat-> Type) (p_0 : P 0) (p_1 : forall n, P n -> P (S n)) (n:nat) : P n
  match n with
    |0 => p_0
    | S n_0 => p_1 n_0 (rec P p_0 p_1 n_0)
  end.
Print rec.

(** In fact, any induction is a use of such induction principle *)

Lemma test: forall n, n+0 = n.
induction n.
simpl. reflexivity.
simpl. rewrite IHn. reflexivity.
Qed.
```

```
Print test. (** note: nat_ind = nat_rect *)

Print nat.
Check nat_rect.

Require List.
Print list.
Check list_rect.
Check cons.
Print cons.

(* pattern matching definition *)

Fixpoint length (X:Type) (l:list X) : nat :=
  match l with
  | nil => 0
  | cons h t => S (length X t)
  end.
Print length.
```

lists.v

```
(** Donnees **)

Inductive list (A : Set) : Set :=
| Nil : list A
| Cons : A -> list A -> list A.

(** Programmes **)

Fixpoint concat (A : Set) (l1 l2 : list A) {struct l1} : list A :=
 (* fonction recursive definie par cas *)
  match l1 with
  | Nil       => l2
  | Cons x tl => Cons A x (concat A tl l2)
  end.
Check concat.

Fixpoint length (A : Set) (l : list A) {struct l} : nat :=
 (* fonction recursive definie par cas *)
  match l with
  | Nil       => 0
  | Cons x tl => 1 + length A tl
  end.

(** Des proprietes **)

Lemma Concat_Nil: forall (A : Set) (l : list A), concat A (Nil A) l = l.
 (* Introduit les hypotheses *)
  intros A l.
 (* On calcule *)
  simpl.
 (* Reflexivite de l'egalite *)
  reflexivity.
Qed.
Lemma Concat_Nil': forall (A : Set) (l : list A), concat A l (Nil A) = l.
 (* Introduit les hypotheses *)
  intros A l.
 (* On raisonne par induction sur la structure de la liste *)
  induction l.
 (* Premier cas : l = Nil *)
  simpl. auto.
 (* Deuxieme cas : l = a::l *)
  simpl. rewrite IHl. auto.
Qed.
Lemma Concat_Length : forall (A : Set) (l1 l2 : list A),
  length A (concat A l1 l2 ) = length A l1 + length A l2.
 (* Introduit hypotheses *)
  intros A l1 l2.
 (* Induction sur la structure de l1 *)
  induction l1.
 (* Premier cas : l1 = Nil *)
  simpl. auto.
 (* Deuxieme cas : l1 = a::l1 *)
  simpl. rewrite IHl1. auto.
Qed.
```

lesson_ind_predicate.v

```
Require Import Arith Bool List Omega.

(* 1 non-recursive predicate *)
(* first possibility : just reformulation of a predicate ... *)

Definition lex_lt_orig (p q :nat*nat) :Prop :=
  (fst p < fst q) \/ (fst p = fst q /\ snd p < snd q).

Inductive lex_lt : nat*nat -> nat*nat -> Prop :=
  | lex_fst : forall a a' b b',  a < a' -> lex_lt (a,b) (a',b')
  | lex_snd : forall a b b',  b < b' -> lex_lt (a,b) (a,b').

Lemma lex_equiv : forall p q,
 lex_lt_orig p q <-> lex_lt p q.
Proof.
 intros (a,b) (a',b').
 split.
 intro H. destruct H. simpl in H.
 apply lex_fst.  assumption.
 simpl in H. destruct H. rewrite <- H.
 apply lex_snd.  apply H0.
 intro H. inversion H.
 left. auto.
 right. auto.
Qed.

(* Here, no recursion. Interest:
  - more readable:
      * splits the cases
      * allows to speak of sub-objects (builds instead of breaking)
      * some equalities can be avoided (no a' above)*)

Print True.
Print False.
Print or.
Print or_ind.
Lemma or_sym : forall A B,
 A\/B -> B\/A.
Proof.
intros.
(*
destruct H.
right; assumption.
left; assumption.
*)
exact
(match H with
 | or_introl a => or_intror _ a
 | or_intror b => or_introl _ b
end).
Qed.
Print and.
Print prod.

(* 2 A first inductive predicate with recursion. *)
```

```
(* The main use of inductive predicate is recursivity: *)

Inductive even : nat -> Prop :=
 | even_O : even O
 | even_SS : forall n, even n -> even (S (S n)).

Lemma even_4 : even 4.
Proof.
apply even_SS.
apply even_SS.
apply even_O.
Qed.

Lemma odd_1 : ~ even 1.
Proof.
intro.
inversion H.
Qed.
Check even_ind.

(* Meaning: the smallest set of integers closed under ... *)
(* See for example a Prolog program:
      even(o).
      even(s(s(N))) :- even(N).
*)

(* 3 Could/Should we do otherwise ? *)

(* An equivalent predicate without inductive type: *)
Fixpoint Even (n:nat) : Prop := match n with
 | 0 => True
 | 1 => False
 | S (S n) => Even n
end.

(* Interest here: you can simplify automatically a concrete problem *)

Lemma Even_100 : Even 100.
 simpl.
 exact I.
Qed.

Lemma Odd_1 : ~ Even 1.
Proof.
 simpl.
 intro.
 auto.
Qed.
Print Even_100.
Print even_4.

Hint Constructors even.
Lemma even_100 : even 100.
 auto 51.
Qed.
Print even_100.
```

```
(* But the computability is not necessary implied by a Fixpoint.*)

Fixpoint Even' (n:nat) : Prop := match n with
 | 0 => True
 | S n => ~Even' n
end.
Lemma Even'_6 : Even' 6.
 simpl.
 intuition.
Qed.
Print Even'_6.

(* For ensuring computability, you should rather use bool. *)

Fixpoint even_bool (n:nat) : bool := match n with
 | 0 => true
 | S n => negb (even_bool n)
end.
Lemma even_bool_100 : even_bool 100 = true.
 simpl.
 reflexivity.
Qed.

(* 4 Inductive predicate & induction principle : *)

Print even.
Check even_ind.
Print even_ind.
Lemma even_double :
 forall n, even n -> exists m, m+m=n.
Proof.
intros n H.
induction H.
exists 0; reflexivity.
destruct IHeven as [m HT].
exists (S m). omega.
Qed.

(* one more tactic of interest : inversion. *)
Lemma one_not_even : ~ (even 1).
Proof.
  unfold not.
  intro.
  inversion H.
Qed.
```

facto.v

```
(* Definir une  fonction par un predicat *)

Require Import ZArith.
Open Scope Z_scope.
Print Scope Z_scope.

Inductive Pfact : Z -> Z -> Prop :=
| Pfact_0 : Pfact 0 1
| Pfact_h : forall n v : Z, n > 0%Z -> Pfact (n-1) v -> Pfact n (n*v).

(* Proprietes du predicat *)

Lemma pfact3: Pfact 3 6.
Proof.
  apply (Pfact_h 3 2). auto with zarith.
  apply (Pfact_h 2 1). auto with zarith.
  apply (Pfact_h 1 1). auto with zarith.
  apply Pfact_0.
Qed.

(* Domaine de la fonction *)

Check Pfact_ind.
Theorem f_dom : forall n v : Z, Pfact n v -> 0 <= n.
Proof.
  intros.
  induction H.
  auto with zarith. (* cas de base *)
  auto with zarith. (* heredite *)
Qed.

Print Z.
Print positive.
Require Import Zwf.
Print well_founded_ind.
Check (Zwf 0).

Theorem f_dom' : forall n:Z, 0 <= n -> exists v:Z, Pfact n v.
Proof.
  intros n.
  (* on applique well_founded_ind a la relation Zwf 0 bien fondee*)
  elim n using (well_founded_ind (Zwf_well_founded 0)).
  intros x Hrec Hle.
  (* on separe cas de base et heredite *)
  SearchPattern (_<_ \/ _=_).
  elim (Zle_lt_or_eq _ _ Hle).
  (* cas1: x<0 - heredite *)
  intros. elim (Hrec (x-1)).
  intros. exists (x*x0). apply (Pfact_h x x0). auto with zarith. apply H0.
  unfold Zwf. auto with zarith.
  auto with zarith.
  (* cas2: x=0 - base*)
  intros. rewrite <- H.  exists 1. apply Pfact_0.
Qed.
```