Chapitre 9

Extraction et programmation impérative

Le système *Coq* permet de modéliser des programmes en les décrivant comme des fonctions dans un langage fonctionnel pur. Il faut cependant bien remarquer que le système *Coq* a pour but de synthétiser des programmes corrects ou de vérifier des programmes, mais pas de les éxécuter. Cette tâche est normalement déléguée aux outils habituels (compilateurs, machines abstraites, etc.)

J'ai remplacé la phrase suivante, à caractère un peu défensif (Pierre) : En revanche, ce système n'est pas un outil d'évaluation efficace pour ces programmes. C'est bon

Pour permettre la génération automatique de code exécutable certifié à partir des modèles formels, le système fournit deux approches. La première approche est basée sur la traduction des programmes fonctionnels du calcul des constructions vers des programmes fonctionnels purs d'un langage fonctionnel efficace, principalement le langage *OCAML*. La seconde approche fournit la possibilité de décrire directement des programmes *impératifs* dans le système *Coq*, ainsi que deux outils de traduction : le premier associe au programme représenté un programme fonctionnel pur qui effectue les mêmes opérations et demande à l'utilisateur de vérifier une collection d'obligations de preuves nécessaires pour que ce programme fonctionnel soit reconnu bien formé ; le second outil de traduction permet de produire un programme impératif, encore dans le langage *OCAML*. Les programmes impératifs ainsi obtenus peuvent donc mélanger des traits purement fonctionnels et des instructions impératives.

9.1 Extraction vers les langages fonctionnels

Les fonctions écrites en Coq sont souvent les modèles de fonctions écrites dans un langage fonctionnel. En retrouvant automatiquement la fonction de ce langage fonctionnel dont la fonction Coq est un modèle, on dispose d'une technique de production de logiciel. On parle alors d'*extraction*. En général, le comportement de la fonction Coq permet de prévoir le comportement de la fonction extraite : on dispose d'une chaîne de production de logiciel certifié, puisque les programmes extraits répondent aux spécifications décrites dans les développements formels.

Deux difficultés doivent être contournées dans la production de fonctions extraites. D'une part, les langages de programmation usuels ne disposent pas de types dépendants et les fonctions bien typées dans le calcul des constructions ne correspondent pas toujours à des fonctions bien typées dans le langage de programmation fonctionnel visé. D'autre part, les fonctions du calcul des constructions contiennent des calculs effectués dans le seul but de fabriquer des démonstrations, mais ces démonstrations n'ont qu'un intérêt limité pour le résultat final. Garder ces calculs mènerait à des implémentations inefficaces des algorithmes. En fait les calculs effectués pour les démonstrations correspondent à des vérifications qui devraient être faites une fois pour toutes, *au moment de la compilation*, tandis que les calculs sur les données effectives doivent être effectué pour chaque donnée présentée à la fonction, *au moment de l'exécution*. Cette séparation entre calculs *au moment de la compilation* ou *au moment de l'exécution* montre les liens qui peuvent exister entre l'extraction de programmes et les techniques de l'évaluation partielle.

Dans le calcul des constructions sous sa forme actuelle, la distinction entre les sortes `Prop` et `Set` sert justement à distinguer entre les calculs sur les aspects logiques qui peuvent être effectués au moment de la compilation et les calculs effectifs sur les données qui doivent apparaître au moment de l'exécution.

9.1.1 Oublier les preuves

Le procédé d'extraction effectue deux opérations distinctes, suivant que l'on traite une définition de type ou une définition de fonction ou de valeur. Pour une définition de type l'opération consiste à éliminer les arguments de type et de preuve dans les constructeurs. Pour les définitions de fonctions ou de valeurs, les expressions représentant des types sont éliminées, mais les expressions représentant des preuves sont remplacées par une valeur unique, ce qui confirme l'indifférence aux preuves. Le nombre d'arguments des fonctions doit alors changer, puisque certains arguments peuvent disparaître. Les contraintes de typage et les restrictions sur les règles d'élimination des types inductifs assurent que les données retournées par ces fonctions ne dépendent jamais des arguments de preuve ou de type, ce qui justifie ce remplacement systématique.

Extraire les types et fonctions non polymorphes

Lorsqu'un type ne présente pas de dépendance, sa mise en correspondance avec un type du langage *OCAML* est immédiate. Par exemple, la déclaration des entiers naturels en *Coq*, donnée par le type inductif suivant :

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

est facilement extraite dans la définition de type suivante :

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

Les fonctions qui calculent sur ce type sans faire apparaître de types dépendants se traduisent directement dans des fonctions du langage fonctionnel cible. Par exemple la fonction de soustraction sur les entiers naturels est décrite dans les bibliothèques de *Coq* par la définition récursive suivante :

```
Fixpoint minus [n:nat] : nat->nat :=
[m]Cases n of
     O => (O)
   | (S k) => Cases m of O => (S k) | (S l) => (minus k l) end
   end.
```

La traduction vers le langage *OCAML* est directe, en remplaçant le mot-clef `Fixpoint` par `let rec`, le filtrage `Cases..of` par le filtrage `match..with`, et l'abstraction par une expression `fun`.

```
let rec minus n =
 fun m -> match n with
            O -> O
          | (S k) -> match m with
                       O -> (S k)
                     | (S l)->minus k l;;
```

Extraire le polymorphisme

Certaines structures de données utilisent les types dépendants simplement pour décrire le polymorphisme. C'est le cas de la structure de donnée des listes, donnée par la définition inductive suivante :

```
Inductive list [A:Set] : Set :=
  nil : (list A) | cons : A -> (list A) -> (list A).
```

En pratique, cette déclaration donne aux fonctions `nil` et `cons` les types suivants :

```
  nil : (A:Set)(nil A)
  cons : (A:Set)A->(list A)->(list A)
```

Si l'on cherche à produire un programme `OCaml`, cette déclaration de type correspond à la déclaration de type suivante :

```
type 'a coqlist = Nil | Cons of 'a * 'a coqlist;;
```

D'un point de vue pratique, la fonction *Coq* `nil` est remplacée par le constructeur `Nil`, qui n'est pas une fonction mais a un type polymorphe, et la fonction *Coq* `cons` est remplacée par un constructeur, qui ne peut pas directement être utilisé comme une fonction non plus. Néanmoins, nous avons le typage suivant :

```
# Nil;;
- : 'a coqlist = Nil
# (fun x y -> Cons(x,y));;
- : 'a -> 'a coqlist -> 'a coqlist = <fun>
```

D'un point de vue purement symbolique, l'argument `A` de type `Set` disparaît dans l'extraction. Les endroits où `A` réapparaît sont remplacés par la variable de typage polymorphe `'a`. Le même procédé se reproduit pour les fonctions polymorphes. Par exemple, la fonction de concaténation de liste `app` est décrite avec une déclaration de la forme suivante[1] :

```
Fixpoint app [A:Set; l:(list A)] : (list A)->(list A) :=
      [m:(list A)]
       Cases l of
         nil => m
       | (cons a l1) => (cons A a (app A l1 m))
       end.
```

À l'extraction, le type de `app` doit perdre l'argument correspondant au polymorphisme. Les paramètres réels de `cons` et `app` correspondant aux arguments disparus sont enlevés du code :

```
let rec app l =
(fun m -> match l with
            Nil -> m
          | Cons(a,l1) -> Cons(a,app l1 m));;
```

Exercice 9.1 Construire l'extraction du type `option` et de la fonction `nth'` :

```
Inductive option [A:Set] : Set :=
 Some : A -> (option A) | None : (option A).

Fixpoint nth' [A:Set; l:(list A);n:nat] : (option A) :=
   Cases l n of
     nil _ => (None A)
   | (cons a tl) O => (Some A a)
   | (cons a tl) (S p) => (nth' A tl p)
   end.
```

Oubli des preuves

Lorsque des types inductifs contiennent des propositions, l'extraction fait disparaître ces propositions. Considérons par exemple le type `sumbool` donné dans les bibliothèques de *Coq* par la déclaration suivante :

```
Inductive sumbool [A : Prop; B : Prop]  : Set :=
      left : A->{A}+{B} | right : B->{A}+{B}.
```

L'extraction de ce type fournit le type suivant :

```
type sumbool = Left | Right;;
```

[1]Pour un discours plus clair, nous avons rendus explicites tous les arguments de `app` et `cons` qui sont normalement implicites dans les bibliothèques de *Coq*.

Ainsi, les deux types A et B donnés en argument ne réapparaissent même pas comme arguments de polymorphisme, puisque les constructeurs n'auront aucun argument. Cette extraction respecte l'interprétation du type sumbool comme une version *enrichie* du type bool. En quelque sorte, l'extraction effectue l'*appauvrissement* inverse.

Pour certains types, comme le type sig, ceci mène à construire un type récursif à un seul constructeur qui a un seul argument (puisque le deuxième qui serait une proposition disparaît). De tels types sont inutiles et rendent le code extrait moins lisible et moins efficace. Ces types inductifs sont simplement supprimés et l'usage du constructeur dans le code disparaît systématiquement.

Pour les fonctions, le même procédé s'applique naturellement. Ainsi la fonction qui compare deux nombres naturels aurait pu être donnée en *Coq* par la définition suivante :

```
Fixpoint eq_nat_dec [n,m:nat]:{n=m}+{~n=m} :=
  <[n:nat]{n=m}+{~n=m}>
  Cases n of
    O => <[m:nat]{O=m}+{~O=m}>
         Cases m of
           O => (left ? ? (refl_equal nat O))
         | (S q) => (right ? ? (O_S q))
         end
  | (S p) => <[m:nat]{(S p)=m}+{~(S p)=m}>
         Cases m of
           O => (right ? ? (sym_not_equal nat O (S p) (O_S p)))
         | (S q) =>
           Cases (eq_nat_dec p q) of
             (left heq)=> (left ? ? (eq_S p q heq))
           | (right hneq) => (right ? ? (not_eq_S p q hneq))
           end
         end
     end.
```

En appliquant systématiquement la méthode exposée jusqu'à maintenant on pourrait obtenir l'expression suivante :

```
let rec eq_nat_dec n m =
  match n with
    O -> (match m with O -> Left | (S p) -> Right)
  | (S p) ->
    (match m with O -> Right
                | (S q) ->  (match eq_nat_dec p q with
                              Left -> Left
                            | Right -> Right));;
```

Mais une optimisation simple consiste à reconnaitre que l'expression

```
(match eq_nat_dec p q with Left -> Left | Right -> Right)
```

est équivalente à l'expression `eq_nat_dec p q`, de sorte que la fonction extraite ci-dessus devrait être présentée de la manière suivante :

```
let rec eq_nat_dec n m =
  match n with
    O -> (match m with O -> Left | (S p) -> Right)
  | (S p) -> (match m with O -> Right | (S q) -> eq_nat_dec p q);;
```

Ce type d'optimisation est également effectué par l'outil d'extraction de *Coq* (lorsque les optimisations sont autorisées).

Donc, les résultats de preuves associées aux données ne sont plus produites par les fonctions, mais ce n'est pas grave, car les fonctions qui prenaient ces preuves en argument ne s'en préoccupent plus non plus. Il y a néanmoins une difficulté avec la fonction `False_rec`, dont l'extraction devrait avoir le type polymorphe `'b -> 'a`, si `'b` est le type de la donnée utilisée pour représenter les preuves. En fait, cette difficulté est facile à résoudre car il est possible en *OCAML* de fabriquer une expression de type `'a`, mais pas de l'évaluer :

```
# let false_rec _ = assert false;;
val false_rec : 'b -> 'a = <fun>
# false_rec ();;
Exception: Assert_failure ("", 18,30 ).
```

Intuitivement, cette construction est justifiée : la proposition passée en argument à `false_rec` est sa spécification et cette fonction ne devrait donc recevoir un argument que lorsque sa spécification est satisfaite. Mais la spécification de `False_rec`, la fonction correspondante dans le calcul des constructions, est `False`. Cette spécification ne peut donc jamais être satisfaite. Prétendre le contraire provoque une erreur.

Voyons un exemple de fonction prenant une preuve en argument et raisonnant sur cette preuve dans certains cas. Il s'agit de la fonction qui calcule le prédecesseur d'un nombre naturel mais n'est pas définie si ce nombre est `O` :

```
Definition pred' : (n:nat)(~n=(O))->nat :=
[n:nat]<[n:nat](~n=O)->nat>
       Cases n of
         O => [h:~O=O](False_rec nat (h (refl_equal nat O)))
       | (S p) => [h:~(S p)=O]p
       end.
```

Dans coq V.7.3 ; en faisant Extraction pred', le code obtenu n'a plus que n comme argument (Pierre) AFPY (idem pour les 4 remarques suivantes)

La fonction extraite pour pred' a la forme suivante :

```
let pred' n = match n with O -> assert false | (S p) -> p;;
```

La fonction `pred'` dans le code extrait prend un argument de moins que la fonction `pred'` du calcul des constructions, mais le second argument n'était pas utilisé pour faire le calcul. Dans les usages manuels, c'est la responsabilité de l'utilisateur de n'utiliser cette fonction que lorsque l'argument n'est pas nul. Pour le code extrait, le procédé d'extraction assure que les témoins pour les spécifications ne sont fournies aux fonctions qui en requièrent que lorsque suffisamment de tests ont été effectués pour garantir que la spécification est satisfaite.

Prenons un exemple de fonction faisant appel à la fonction `pred'`.

```
Definition pred2 : nat -> nat :=
  [n:nat]Cases (eq_nat_dec n O) of
           (left h) => O | (right h') => (pred' n h')
         end.
```

Dans cette fonction, l'hypothèse `h'` est la justification qui assure que la spécification est satisfaite pour les données en entrée de la fonction `pred'`. Dans le code extrait cette hypothèse est oubliée mais on sait au moment de l'extraction que cette fonction est nécessaire.

Meme remarque que la marginpar précédente : dans l'extraction de pred2, on appelle seulement pred' n (Pierre) ; faire une bonne intro (AFPY)

```
let pred2 n =
   match eq_nat_dec n O with Left -> O | Right -> pred' n;;
```

Ici nous sommes assurés que `pred'` n'est appelée que lorsque `n` est non nul, car la valeur `Right` retournée par `eq_nat_dec` contient « moralement » l'assurance nécessaire. C'est cette assurance morale qui est donnée par l'argument `prop`.

Il existe néanmoins des valeurs pour lesquelles l'argument de preuve ne peut pas être enlevé directement. Ce sont des valeurs pour lesquelles il n'existe pas de valeur licite dans le monde des programmes. Par exemple, nous pouvons considérer la fonction suivante :

```
Definition pred'_on_O : (pred' O).
```

Si l'on appliquait brutalement la technique de faire disparaitre les arguments de preuve, on devrait obtenir la valeur suivante :

```
let pred'_on_O = pred' O
```

Mais cette définition demande d'exécuter `assert false`, ce qui provoque une erreur à l'exécution. Pour contourner ce problème, l'argument représentant une proposition n'est pas enlevé des arguments de la fonction.extraite, même si cet argument n'est effectivement pas utilisé pour le calcul.

```
let pred'_on_O _ = pred' O
```

Aucune fonction extraite ne fournira l'argument manquant à la fonction `pred'_on_O` et pour cause, ceci signifierait que l'on aurait trouvé une preuve de `O=O`.

Récursion bien fondée et extraction

La récursion bien fondée sera présentée en détail dans la section 14.2, mais nous l'abordons ici sous l'angle unique de l'extraction et lecteur novice pourra laisser cette section en première lecture. Cette technique de programmation certifiée est particulière car elle est traitée en *Coq* par une récurrence sur une propriété inductive qui, puisqu'elle est une propriété, est amenée à disparaître dans l'extraction.

La récursion bien fondée est basée sur la propriété d'accessibilité décrite par une définition similaire à la suivante, où nous avons changé quelques noms pour éviter les conflits de nommage avec la définition utilisée dans *Coq* :

```
Inductive acc [A : Set; R : A->A->Prop]  : A->Prop :=
  acc_intro : (x:A)((y:A)(R y x)->(acc A R y))->(acc A R x).
```

Chaque preuve d'accessibilité pour un `x` donné contient une fonction qui montre que les prédécesseurs de `x` par `R` sont accessibles. On peut également définir une fonction `acc_inv` qui a la valeur suivante :

```
Definition acc_inv :
 (A:Set;R:A->A->Prop;x:A)(acc A R x)->
     (y:A)(R y x)->(acc A R y) :=
[A,R,x,Hacc,y,Hr]
  <[x:A;H:(acc A R x)](y:A)(R y x)->(acc A R y)>
   Cases H of (acc_intro x f) -> f end.
```

Si x et y sont des éléments d'un type A, si R est une relation sur A, H_a est une preuve de (`acc` A R x), et H_r est une preuve de (R y x), alors le terme (`acc_inv` A R x H_a y H_r) est une preuve de (`acc` A R y). De plus, cette preuve est admise comme une preuve structurellement plus petite que H_a, ce qui sera utilisé pour définir des fonctions par la commande `Fixpoint`. Ainsi, un récurseur pour ce type inductif est décrit par la définition suivante (ce récurseur est plus simple que celui qui est effectivement utilisé dans *Coq*) :

```
Fixpoint acc_rec
[A:Set;R:A->A->Prop;P:A->Set;
 f:(x:A)((y:A)(R y x)->(P y))->(P x);
 x:A;hacc:(acc A R x)] : (P x) :=
 (f x [y:A;hy:(R y x)](Acc_rec A R P f y (Acc_inv A R x hacc y hy))).
```

Une fonction `well_founded_induction'` est ensuite définie de la façon suivante :

```
Definition well_founded_induction' :
 (A:Set;R:A->A->Prop;Rwf:(x:A)(acc A R x);P:A->Set;
  F:(x:A)((y:A)(R y x)->(P y))->(P x); x:A)(P x):=
[A,R,Rwf,P, F, x] (Acc_rec A R P F x (Rwf x)).
```

Meme remarque sur la dsiparition du 3ème argument en 7.3 (Pierre) ; (Yves : normalement c'est corrigé)

A l'extraction, on obtient les fonctions suivantes :

```
let rec acc_rec f x = f x (fun y _ -> acc_rec f y);;

let well_founded_induction' f x =
        acc_rec f x
```

En effet, les trois premiers arguments de `Acc_rec` sont soit des types, soit des fonctions retournant des types. Le quatrième argument est une fonction qui retourne un élément dans un ensemble de type `Set`, il est donc conservé, le cinquième argument a pour type `A` et `A` a pour type `Set` et est donc conservé et le sixième argument est de type `(Acc A R x)`. La fonction `acc_rec` a donc seulement deux arguments significatifs. Observons maintenant son premier argument, qui est extrait du troisième argument de `acc_rec`. C'est une fonction qui prend trois arguments. Le premier argument est de type `A` et est conservé parce que `A` est de sorte `Set`. le second argument est une fonction qui retourne un élément dans un type de sorte `Set` et est donc conservé. La fonction `f` est donc représentée dans le code extrait par une fonction à deux arguments également.

La valeur de `acc_rec` est une application de la fonction `f`. Le deuxième argument est une application de `acc_rec` dont les arguments de rangs un à trois et le sixième disparaissent. Ceci explique la simplicité de la fonction `acc_rec` obtenue.

Si l'on définit une fonction récursive bien fondée on est amené à construire la fonctionnelle qui sera donnée en 5e argument de `well_founded_induction'`, il s'agira encore d'une fonction qui fait intervenir données et preuves. Pour montrer rapidement le fonctionnement de l'extraction sur de telles fonctions, nous allons supposer que les preuves nécessaire sont déjà faites pour une description d'une fonction de calcul de logarithme discret.

```
Fixpoint div2 [n:nat] : nat :=
  Cases n of (S (S p))=>(S (div2 p))| _ => O end.

Hypotheses
  div2_lt : (x:nat)(lt (div2 (S x)) (S x));
  lt_acc : (x:nat)(acc nat lt x).

Definition log2_aux_F : (x:nat)((y:nat)(lt y x)->nat)->nat :=
 [x:nat]<[x:nat]((y:nat)(lt y x)->nat)->nat>
        Cases x of O => [_]O
        | (S p) => [f](S (f (div2 (S p))(div2_lt (S p))))
        end.

Definition log2_aux :=
  (well_founded_induction' nat lt lt_wf [_:nat]nat
      log2_aux_F).

Definition log2 : (x:nat)~x=O->nat :=
  [x;h](log2_aux (pred x)).
```

Le code extrait pour la fonction `log2_aux` a la forme suivante :

```
let log2_aux x =
  well_founded_induction'
   (fun x f ->
```

```
match x with
  O -> O
| (S p) -> (S (f (div2 (S p)) __));;
```

Si l'on disposait d'outils de transformations de programmes pour le langage *OCAML*, on s'apercevrait après quelques $\beta\eta$-conversions, que cette définition est équivalente à la suivante :

```
let rec log2_aux x =
    match x with O -> O | (S p)-> (S (log2_aux (div2 (S p))))
```

Nous avons fini d'exposer une méthode complète d'extraction de code fonctionnel sans types dépendants à partir d'expressions du calcul des constructions inductives. Cette méthode est approximativement celle qui était utilisée dans *Coq*, dans la version `7.2`. En fait, l'outil d'extraction contient quelques optimisations spécifiques pour la récursion bien fondée, de telle sorte que si l'on utilise `well_founded_induction` au lieu de la fonction que nous avons décrite dans cette section, le code extrait est directement le code équivalent que nous venon de présenter. Notre introduction du type `acc` n'avait qu'un but didactique permettant de mieux comprendre le mécanisme d'extraction.

9.1.2 La dualité `Prop`/`Set` et l'extraction

La façon dont l'extraction exploite la sorte `Prop` est un exemple supplémentaire du principe d'indifférence aux preuves. Les valeurs des preuves n'ont pas d'importance, seule leur existence compte. La première méthode d'extraction allait même plus loin dans l'indifférence, puisque même l'existence des preuves était laissée dans l'oubli par cette méthode, avec l'argument informel suivant : si le calcul d'une certaine fonction est requis, c'est que le terme du calcul des constructions correspondant a reçu les preuves nécessaires, il n'est donc pas nécessaire de vérifier même l'existence de la preuve. Nous l'avons vu, cette justification n'est valable que si la stratégie d'exécution n'effectue des calculs que par besoin.

Nous avons volontairement évité d'expliquer les justifications qui assurent que chaque fonction extraite se comporte de manière cohérente avec la fonction du calcul des constructions associée. Ce sujet est assez complexe et au delà des prétentions de cet ouvrage et nous invitons la lectrice à se reporter à l'article [?] pour une référence plus exacte.

dans les remerciements, suggérer que La Lectrice n'est autre que Laurence Rideau

Référence manquante, et puis toujours la meme question sur l'oubli des preuves en 7.3 (Pierre)

Néanmoins, la cohérence de l'extraction permet de justifier les choix qui ont été effectués dans le typage des constructions de filtrage.

Filtrage sur les propositions

Il n'est pas permis de filtrer une proposition inductive pour produire une donnée de type `Set` ou une donnée de type `Type`. Attardons-nous sur la première partie de cette restriction. S'il était permis de filtrer une propriété inductive pour construire une valeur de type `Set`, on pourrait ainsi obtenir obtenir des données différentes en fonction de preuves différentes.

Supposons que l'on ait permis cette forme de filtrage, on pourrait alors écrire une valeur de la forme suivante :

```
Definition or_to_nat : (A,B:Prop)(A\/B)-> nat :=
[A,B,H]Cases H of (or_introl h)->(S O) | (or_intror h) -> O end.
```

L'application de la fonction `(or_to_nat A B H)` retournerait forcément 1 lorsque `B` est faux et forcément 0 lorsque `A` est faux, mais cette fonction serait extraite en une valeur `or_to_nat ()`, qui serait toujours la même, quelque soit le contexte d'exécution. La correspondance entre le code extrait et les fonctions *Coq* initiales serait rompue.

Choix de types inductifs

Il existe trois variantes de quantifications existentielles dans *Coq*, suivant la sorte donnée au type et aux composantes. Nous avons décrit ces variantes dans la section 8.1.1. La différence de traitement entre les types de sorte `Prop` et les types de sorte `Set` dans l'extraction apporte une justification pratique importante à ces variantes. D'un point de vue logique, les trois expressions suivantes sont analogues :

$$\texttt{\{m : A \& \{ n : B | P\}\}} \tag{9.1}$$

$$\texttt{\{m : A | (EX n : B | P)\}} \tag{9.2}$$

$$\texttt{(Ex m : A | (EX n : B | P))} \tag{9.3}$$

Mais une fonction *Coq* calculant une valeur de la forme 9.1 sera extraite en une fonction calculant deux valeurs `m` et `n` satisfaisant la propriété `P`, tandis qu'une fonction calculant une valeur de la forme 9.2 ne calcule qu'une valeur `m`, telle qu'il existe un `n` tel que la propriété `P` soit satisfaite. Enfin, une fonction calculant une valeur de la forme 9.3 est la preuve d'un théorème et ne sera tout simplement pas extraite, ou alors seulement vers la valeur pré-définie `prop`.

D'un point de vue logique, les fonctions retournant un type `or` et les fonctions retournant un type `sumbool` sont également analogues, mais les premières devront être préférées lorsque l'on ne veut pas voir les calculs qu'elles contiennent apparaître dans le type extrait. Une méthode simple pour obtenir une fonction de la première forme lorsque l'on dispose déjà d'une fonction de la deuxième forme est d'utiliser la fonction de « dégradation » suivante :

```
Definition sumbool_to_or : (A,B:Prop){A}+{B}->(A\/B) :=
[A,B:Prop;v:{A}+{B}]
 Cases v of
   (left Ha) -> (or_introl A B Ha)
 | (right Hb) -> (or_intror A B Hb)
 end.
```

Nous avons appelé cette fonction une fonction de dégradation parce que le trajet inverse, de `or` vers `sumbool` est impossible, pour la raison que nous avons exposée dans la section précédente.

9.1.3 Production effective de code *OCAML*

La commande la plus simple pour produire un fichier *OCAML* contenant les fonctions extraites a la forme suivante :

```
Extraction "log.ml" log.
```

Si nous voulons mettre en œuvre cette fonction, il est utile de se munir de fonctions de conversion entre les nombres de type `int` en *OCAML* et la représentation de type `nat`. On pourra donc tester la fonction, en la combinant avec ces fonctions de conversion, par exemple de la façon suivante :

```
let rec int_to_nat = function 0 -> O | n -> S(int_to_nat (n-1))

let rec nat_to_int = function O -> 0 | S n -> (nat_to_int n)+1

let e_log n = nat_to_int (log (int_to_nat n))
```

9.2 Description de programmes impératifs

En programmation impérative usuelle, les effets de bord sont concentrés sur l'opération d'affectation. Lorsqu'une affectation a lieu, on comprend habituellement que l'état de la machine change, sans que cet état ait été décrit précisément. Si nous voulons rendre compte de ce genre d'opération, il faudra considérer que toutes les fonctions accédant à des variables ré-affectables sont représentables dans le calcul fonctionnel pur par des fonctions prenant un argument supplémentaire, l'état, et toutes les fonctions faisant des effets de bords devront retourner le nouvel état.

Attention à l'effte subliminal ! Dire dans l'intro de ce chapitre qu'on verra d'abord une approche *manuelle* mais aue les outis existent pour s'en abstraire (Pierre)

9.2.1 Représentation de l'état

On pourra tirer avantage de la possibilité de construire des structures enregistrements « `Record` » pour représenter l'état. Ainsi pour un programme manipulant une variable mutable booléenne `b` et une variable entière `x`, on construira l'état suivant :

```
Record tuple : Set := mk_tuple {b:bool;x:Z}.
```

Nous rappelons que cette définition définit `tuple` comme un type inductif et `b` et `x` comme des fonctions de type `tuple->bool` et `tuple->Z` respectivement.

Typiquement les fonctions `b` et `x` seront utilisées pour représenter les accès aux variables de même nom dans la représentation impérative. Par exemple si l'on veut considérer l'expression $x + 3$ dans le contexte impératif, on construira naturellement l'expression $(x\ t) + 3$ dans la traduction fonctionnelle, si t est la variable de type `tuple` représentant l'état courant de la machine.