Implémentation d'algorithmes génériques sur les graphes en Coq

Rapport de stage de master M1

Simon Chemouil & Ludovic Arnold

responsables de stage

Sylvain Conchon & Évelyne Contejean

25 mai 2007

Résumé

Ce rapport récapitule le stage effectué sous la ligne directrice du processus de génie logiciel qu'on a suivie. Après avoir présenté une méthode de conception pour rendre les algorithmes génériques à l'aide de foncteurs dans les langages fonctionnels (Objective Caml ici), nous présentons les algorithmes qui vont illustrer le rapport et sur lesquels nous avons travaillé : le plus court chemin de Dijkstra et les composantes fortement connexes. Nous abordons une méthode pour transformer les programmes impératifs en Objective Caml en fonctionnel pur, étape nécessaire pour les porter en Coq. Nous présentons ensuite Coq, un assistant de preuve, et son fonctionnement de façon sommaire pour après y porter nos algorithmes fonctorisés dans le but d'en prouver la terminaison et de commencer une bibliothèque de graphes. Pour finir, nous prouvons la correction d'un programme trivial et proposons des spécifications pour la correction de l'algorithme du plus court chemin.

Table des matières

1 Foncteurs et algorithmes génériques **2**

1.1 Motivation . . . 2

1.2 Programmation modulaire . . . 2

1.2.1 Signatures . . . 2

1.2.2 Algèbres . . . 3

1.2.3 Les modules en Objective Caml . . . 4

1.3 Généricité des algorithmes par les foncteurs . . . 5

1.3.1 Les limites de la généricité traditionnelle en Objective Caml . . . 5

1.3.2 Les foncteurs et l'approche « un algorithme = un foncteur » . . . 5

1.3.3 Ocamlgraph . . . 8

2 Algorithmes sur les graphes **9**

2.1 Présentation des algorithmes étudiés . . . 9

2.1.1 Le chemin le plus court de Dijkstra . . . 9

2.1.2 Les composantes fortement connexes . . . 12

2.2 L'implémentation des algorithmes d'Ocamlgraph . . . 14

2.2.1 Le plus court chemin de Dijkstra . . . 14

2.2.2 Les composantes fortement connexes . . . 16

2.3 Le passage en fonctionnel pur . . . 17

2.3.1 Méthodologie . . . 18

2.3.2 Les algorithmes en fonctionnel . . . 18

2.3.3 Les limites de garantie d'Objective Caml . . . 20

3 Implémentation en Coq **22**

3.1 Présentation de l'assistant de preuve Coq . . . 22

3.1.1 L'automatisation de la preuve . . . 22

3.1.2 L'assistant de preuve Coq . . . 22

3.1.3 Rappels de logique . . . 22

3.1.4 Spécification et propositions . . . 23

3.1.5 Les preuves . . . 24

3.1.6 Mécanismes des preuves 24
3.1.7 Utilisation de tactiques 25
3.1.8 Constructions inductives 27
3.1.9 Les foncteurs dans Coq 27
3.2 Terminaison des programmes 29
3.2.1 Récurrence structurelle 30
3.2.2 Nos algorithmes en Coq et leur terminaison 30

4 Propriétés de correction **38**
4.1 Preuves de programmes en Coq 38
4.1.1 Spécification d'une fonction 38
4.1.2 Preuve 38
4.2 Spécification du plus court chemin de Dijkstra 41

Introduction

De plus en plus, les logiciels prennent une place dans les systèmes critiques où la vie humaine est en jeu : il sont déjà présents dans les voitures, avions, centrales nucléaires et les outils médicaux (rayons X, chirurgie assistée par robot, ...). Il est donc primordial de chercher à garantir leur fiabilité. Cette recherche de la correction des programmes est en vérité bien antérieure au micro-ordinateur puisqu'Alan Turing s'en préoccupait déjà. Depuis, l'informatique et le génie logiciel ont grandement progressé et de nombreux outils, dits assistants de preuve, existent pour prouver la correction d'un programme à l'aide d'un ordinateur.

L'intérêt de prouver la correction d'un programme réside aussi dans le fait qu'il est ensuite possible d'extraire un programme garanti correct à partir de la preuve (vers différents langages !). Bien que l'état de l'art ne permette pas d'obtenir la performance désirée, il s'agit d'un grand sujet de recherche qui pourrait un jour nous permettre d'écrire les logiciels différemment, nous offrant les propriétés rêvées :

- la garantie qu'on n'aura pas de « plantages » (segfaults, etc) : un bon système de types comme celui d'Objective Caml nous offre déjà cette sécurité
- la garantie que la fonction termine
- la garantie que le programme fait ce qu'il doit faire

En attendant le jour où les programmes extraits des preuves seront utilisables, il n'en demeure pas moins utile d'offrir ces mêmes garanties à partir du code d'un programme : c'est la ligne directrice de notre projet de stage.

Nous avions deux directions pour ce stage : prouver deux algorithmes d'Ocamlgraph[1], une bibliothèque de graphes pour Objective Caml, et commencer une bibliothèque de graphes pour Coq[2], un assistant de preuve. Nous avons choisi d'unir ces deux directions au possible et de porter des parties d'Ocamlgraph (l'algorithme du plus court chemin de Dijkstra et celui des composantes fortement connexes) en Coq pour prouver leur terminaison (et un début de correction, qui reste à prouver). La bibliothèque Coq utilise donc la même conception qu'Ocamlgraph.

Ocamlgraph est conçu suivant une approche qui permet de définir les algorithmes de façon générique : l'utilisateur peut fournir la structure de graphe qui lui convient le mieux et les algorithmes tels que le plus court chemin ou les composantes fortement connexes travailleront dessus. Cette approche par foncteurs, reprise pour notre bibliothèque de graphes en Coq, sera donc présentée en premier.

Nous rappellerons ensuite les algorithmes que nous avons travaillés[3], leur fonctionnement et implémentation dans Ocamlgraph. Nous développerons sur la méthode utilisée pour les porter en Coq.

Une fois les algorithmes présentés, Coq fera l'objet d'une brève présentation, nécessaire pour comprendre la présentation de notre bibliothèque et des preuves de terminaison de nos algorithmes.

Enfin, nous présenterons une preuve de la correction d'un programme trivial en Coq pour en illustrer le principe avant de proposer des spécifications de la fonction du plus court chemin de Dijkstra.

Ce rapport de stage peut sembler copieux mais il correspond à la direction que nous avons suivie au cours de ce stage. Pour limiter, il aurait soit fallu faire l'impasse sur des sujets abordés et du travail effectué, soit présenter uniquement le travail sans expliquer les motivations, les raisonnements et les bases. Nous avons opté pour un rapport plus complet qui selon nous reflète le stage.

[1] `http://ocamlgraph.lri.fr/`
[2] `http://coq.inria.fr/`
[3] Simon : chemin le plus court de Dijkstra. Ludovic : composantes fortement connexes.

Chapitre 1

Foncteurs et algorithmes génériques

1.1 Motivation

Un des enseignements principaux tirés du génie logiciel est la nécessité de « découplage »(*loose-coupling*) dans le code, pour assurer l'abstraction sur les structures de données et fonctions, ainsi que la pérenité et la réutilisabilité des programmes. Il s'agit de faire en sorte qu'un composant logiciel ne connaîsse de ses « interlocuteurs » que ce dont il a besoin pour communiquer avec eux. Alors que les langages orientés objet utilisent dans ce but un mélange d'espaces de noms (*namespaces* ou *packages*) et de classes avec leur relations d'héritage, le paradigme fonctionnel propose une approche différentes : les foncteurs.

Après une présentation des modules et des foncteurs, nous allons voir comment ceux-ci peuvent être utilisés pour offrir une méthode qui permet la généricité des algorithmes, non pas seulement au niveau du type de données traitées (polymorphisme à la OBJECTIVE CAML ou types paramétrés « templates » Java), mais au niveau de l'implémentation même des structures de données utilisées par l'algorithme. C'est sous cette approche que sont conçus OCAMLGRAPH ainsi que notre projet COQ.

1.2 Programmation modulaire

Afin de rendre le code modulaire, les langages de programmation distinguent les interfaces de leur implémentation. En C par exemple, l'interface d'un module est définie dans le fichier d'en-têtes (*headers*) *.h* et les différentes implémentations possibles sont fournies dans des fichiers *.c*. On présente ici brièvement les concepts qui se cachent derrière les modules dans les langages pour ensuite présenter leur implémentation en OBJECTIVE CAML.

1.2.1 Signatures

On définit l'interface abstraite qui contiendra les types ainsi que les opérations exportés par le module, c'est-à-dire à la disposition du reste de l'environnement. Cette interface est nommée « signature » : une signature Σ est donc la donnée de deux ensembles finis $|\Sigma|$, le support de Σ, et Ω respectivement nommés « sortes » et « opérateurs » (les sortes correspondent aux types et les opérateurs aux définitions de fonctions). Les opérateurs sont déclarés par leur nom et leur « profil », c'est-à-dire leur type.

Voici un exemple simple de signature pour les naturels dans un pseudo-langage. Ici, cette signature est la donnée d'un type N (le nom donné ici est bien entendu arbitraire), de la constante zéro et d'opérations simples sur les entiers.

```
signature NAT
    sort N
    ops
```

```
zero  : N
succ  : N → N
plus  : N, N → N
mult  : N, N → N
```

1.2.2 Algèbres

Une algèbre $\mathcal{A} \hat{=} (\{\mathcal{A}_s\}_{s \in |\Sigma|}, \{f_w : \mathcal{A}_{w,s_1} \times ... \times \mathcal{A}_{w,s_n} \rightarrow \mathcal{A}_{w,s}\}_{w: s_1,...,s_n \rightarrow s \in \Omega})$ est *une* implémentation de la signature Σ. Il est donc possible de définir plusieurs algèbres différentes, c'est-à-dire plusieurs implémentations différentes, pour une signature donnée. Voici une implémentation possible d'une NAT-algèbre :

```
algebra NAT1
    implements NAT
    sets
        N    : ℕ
    functions
        zero : 0
        succ : n := n + 1
        plus : n1, n2 := n1 + n2
        mult : n1, n2 := n1 × n2
```

On remarque qu'il serait possible de définir des algèbres pour une signature donnée qui, malgré qu'elles soient bien typées, ne font pas du tout ce qu'on attendrait. Pour reprendre notre exemple, cette NAT-algèbre sur les booléens est tout-à-fait valide puisque bien typée :

```
algebra NATBool
    implements NAT
    sets
        N    : {⊤, ⊥}
    functions
        zero : ⊥
        succ : ¬
        plus : ∨
        mult : ∧

algebra NATBool2
    implements NAT
    sets
        N    : {⊤, ⊥}
    functions
        zero : ⊤
        succ : ¬
        plus : ∧
        mult : ∨
```

Ici, $NATBool$ et $NATBool2$ sont bien des NAT-algèbres car il n'y a qu'une vérification de son typage et pas du « sens désiré ». Nous verrons qu'en Coq il est possible de spécifier des propriétés et de les prouver : on peut alors garantir que l'implémentation d'un module garde le « sens désiré ». Dans ce cas, la signature contient des lemmes (une spécification sur les propriétés des paramètres du module) et l'algèbre contient leur preuve.

Par exemple, une propriété simple sur les entiers, qui ferait partie d'une spécification plus complète, qu'on pourrait ajouter à la signature NAT est le lemme $L : \forall n, plus(n, zero) = n \wedge mult(n, zero) = zero \wedge mult(n, succ(zero)) = n$. L'algèbre $NAT1$ devrait alors donner, en plus, une preuve que son

implémentation respecte bien le lemme L pour être valide (par ailleurs, on remarque que $NATBool$, même ainsi, serait toujours valide en fournissant la preuve adéquate, car le lemme L est prouvable sur l'algèbre de Boole ! En revanche, $NATBool2$ ne serait plus valide car $\exists n = \bot$, $mult(n, zero) = n \vee \top = \top \neq n$).

L'intérêt d'une spécification est donc de garantir la correction du module tout en laissant le maximum de liberté dans l'implémentation.

1.2.3 Les modules en Objective Caml

La syntaxe des modules en Objective Caml est légèrement différente de celle présentée plus haut. Les signatures sont placées dans un fichier *.mli* et leurs algèbres dans les fichiers *.ml*. Voici l'exemple des naturels repris en Objective Caml.

```
(* une signature pour les naturels en Objective Caml *)
module type NAT = sig
    type n
    val zero : n
    val succ : n -> n
    val plus : n * n -> n
    val mult : n * n -> n
end
```

On peut alors définir plusieurs NAT-algèbres ainsi :

```
(* une implémentation de NAT avec les entiers natifs *)
module NAT1 = struct
    type n         = int
    let zero       = 0
    let succ n     = n+1
    let plus (n,m) = n + m
    let mult (n,m) = n * m
end

(* une implémentation de NAT avec les types algébriques *)
module NAT2 = struct
    type n             = Z | S of n
    let zero           = Z
    let succ n         = S n
    let rec plus (n,m) = match m with
                         | Z -> n
                         | S p -> S (plus (n,p))
    let rec mult (n,m) = match m with
                         | Z -> Z
                         | S p -> plus (n, (mult (n, p)))
end

(* une implémentation de NAT avec les booléens *)
module NATBool = struct
    type n     = bool
    let zero   = false
    let succ   = not
    let plus   = (||)
    let mult   = (&&)
end
```

1.3 Généricité des algorithmes par les foncteurs

1.3.1 Les limites de la généricité traditionnelle en Objective Caml

Afin d'augmenter la réutilisabilité du code (et en règle générale éviter au maximum toute redondance), les langages de programmation ont évolué pour proposer des moyens de rendre les algorithmes plus génériques. Ainsi, des langages fonctionnels tels qu'Objective Caml disposent d'une inférence de types accompagné de la prise en charge de types polymorphiques et paramétrés qui permettent en partie d'atteindre cet objectif. Par exemple, la fonction *map* qui applique une fonction f à une liste l pour renvoyer la liste $l' = (f\,l_1, ..., f\,l_{|l|})$ est définie ainsi :

```
# let rec map l f = match l with
| [] -> []
| hd : :tl -> f(hd) : :(map tl f)
; ;
```

et a pour type $\alpha\,list \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta\,list$, c'est-à-dire qu'elle attend :

- une liste générique d'éléments de type α
- une fonction qui prend un paramètre de type α et retourne un élément de type β

et qu'elle renvoie une liste d'éléments de type β.

C'est son système d'inférence qui permet à Objective Caml de déterminer qu'il travaille sur une liste (car il remarque les deux constructeurs de la liste l, la valeur liste vide *nil* notée [] ainsi que le constructeur *cons* noté ::). En revanche, le système ne connaît rien du type des éléments de cette liste, car aucune opération qui permettrait de les inférer ne s'applique sur eux. Objective Caml utilise alors le type le plus général possible : un type polymorphique. Le type de la fonction f est inféré après avoir déterminé ce type, et comme Objective Caml ne connaît rien de cette fonction, elle est aussi une fonction polymorphique : c'est n'importe quelle fonction à un seul argument. Enfin, le type de la liste renvoyée par *map* est construit de la même manière.

On voit donc qu'on dispose ici d'un système très confortable qui permet au programmeur de ne pas spécifier les types et qui lui garantit malgré tout un choix plus général que s'il le faisait à la main (tout en conservant d'autres propriétés intéréssantes comme le typage statique). Cependant, cette généricité reste ici limitée aux types des données traitées (et fonctions traitées).

Par ailleurs, une autre préoccupation importante en Objective Caml est de pouvoir garantir que deux fonctions génériques auront le même type : les modules permettent de garantir que plusieurs fonctions du même module partagent des types. C'est aspect est primordial pour la généricité.

Comment donc étendre la généricité aux structures de données traitées ? Par exemple, comment permettre que notre fonction *map* s'applique sur n'importe quel conteneur ?

1.3.2 Les foncteurs et l'approche « un algorithme = un foncteur »

L'objectif est donc de rendre génériques les algorithmes non pas uniquement en leur permettant de traiter des données de types génériques, mais aussi de travailler sur des structures génériques. En reprenant notre exemple précédent sur les naturels, nous allons illustrer cette approche à l'aide des algorithmes d'Euclide sur la division entière, le modulo et le plus grand commun diviseur. L'algorithme de la division nécessitant la soustraction, et le plus grand commun diviseur nécessitant une fonction testant l'égalité et l'infériorité, nous définissons une nouvelle signature pour NAT.

```
module type NAT = sig
    type n
    val zero  : n
    val succ  : n -> n
    val plus  : n * n -> n
```

```
    val mult  : n * n -> n
    val prev  : n -> n
    val minus : n * n -> n
    val eq    : n * n -> bool
    val gt    : n * n -> bool
end
```

Disposant de cette signature, nous pouvons définir la signature notre algorithme du plus grand commun diviseur :

```
module PGCD (N : NAT) : sig
    val pgcd : N.n * N.n -> N.n
end
```

Le module *PGCD* requiert un module *N* de type *NAT* : il s'agit d'un module paramétré, nommé communément un *foncteur*. La possibilité de paramétrer les modules offerte par OBJECTIVE CAML et d'autres langages fonctionnels permet d'établir une relation de hiérarchie entre les foncteurs et modules : les foncteurs sont en quelques sortes des modules d'ordre supérieur, là ou les modules non paramétrés sont de premier ordre. Tout comme les valeurs fonctionnelles de second ordre, les foncteurs ont besoin d'être appliqués à des valeurs de premier ordre, les modules non paramétrés, pour être utilisés (dans le cas contraire, la définition n'est que partielle et ne peut pas être utilisée).

Les foncteurs sont donc les outils que nous allons utiliser pour rendre nos algorithmes génériques : ils seront paramétrés par des modules ou d'autres foncteurs qui exportent les opérations nécessaires au déroulement de l'algorithme sur une structure de données inconnue de l'algorithme ! Par ailleurs, il est possible pour un foncteur d'instancier un de ses modules paramètres avec le type d'un autre de ses modules, garantissant ainsi d'avoir des fonctions définies dans des modules différents qui ont le même type.

Notre foncteur *PGCD* dipose donc d'une fonction *pgcd* de type $N.n \times N.n \rightarrow N.n$. Le code qui réalise *pgcd* est donné dans le fichier d'implémentation *.ml*. Dans le cadre de la méthode de conception présentée, l'implémentation de l'algorithme fonctorisé est en quelque sorte l'algorithme « canonique » : il n'y a pas beaucoup de choix possibles puisqu'on doit se baser exclusivement sur le module paramètre du foncteur et ses fonctions. La possibilité n'est toutefois pas retirée de proposer une autre implémentation qui travaillerait sur le module paramètre, mais souvent elle requiert d'utiliser un autre module paramètre avec une signature appropriée aux besoins du nouvel algorithme.

```
module PGCD
  (N : NAT) =
struct
    open N

    (* algorithme de la division euclidienne *)
    let rec eucl n m i =
        if (gt (n,m) || eq (n,m)) then
            eucl (minus (n,m)) m (succ i)
        else
            (n,i)

    (* modulo *)
    let modulo (n,m) = fst (eucl n m zero)

    (* division entière *)
    let div  (n,m) = snd (eucl n m zero)

    (* plus grand commun diviseur *)
    let rec pgcd (n,m) =
        if (eq (m,zero)) then
```

```
                n
            else
                let c = modulo (n,m) in
                pgcd (m,c)
    end
```

Le module *PGCD* réalise bien sa signature car il fournit une fonction *pgcd* de type $N.n \times N.n \rightarrow N.n$. Cependant, rien n'empêche d'ajouter au module des fonctions utilitaires dont a besoin l'algorithme : ici, une fonction qui déroule l'algorithme de la division euclidienne et renvoie la paire (quotient, reste) ainsi que deux accesseurs pour le modulo et la division.

On remarque bien que toutes les opérations effectuées dans le module sur les naturels définis par *NAT* n'utilisent que des fonctions qui y sont définies. On obtient ainsi le découplage espéré puisque notre algorithme fonctionnera avec n'importe quelle implémentation de *NAT*. Les développeurs d'une bibliothèque utilisant cette approche fonctorisée pour la généricité des algorithmes peuvent choisir de fournir différentes implémentations ou laisser au soin de l'utilisateur de les fournir. Voici une implémentation utilisant les types natifs d'Objective Caml :

```
module NAT  = struct
    type n          = int
    let zero        = 0
    let succ n      = n+1
    let plus (n,m)  = n + m
    let mult (n,m)  = n * m
    let prev n      = n-1
    let minus (n,m) = n - m
    let eq (n,m)    = (n=m)
    let gt (n,m)    = (n>m)
end
```

Enfin, il ne reste plus qu'à instancier le module *PGCD* paramétré par notre nouveau module *NAT* enfin implémenté :

```
module NATPGCD = PGCD(NAT)
```

On teste si notre algorithme fonctionne bien :

```
# Main.NATPGCD.pgcd ; ;
- : Main.NAT.n * Main.NAT.n -> Main.NAT.n = <fun>
# Main.NATPGCD.pgcd (225,375) ; ;
- : Main.NAT.n = 75
```

Nous avons donc vu à travers cet exemple trivial une méthode, pour les langages fonctionnels comme Objective Caml mais qui peut aussi inspirer d'autres paradigmes, qui répond au besoin d'étendre la généricité aux structures de données utilisées par un algorithme. En ayant chaque algorithme sous forme de foncteur, on permet à la fois :

- à chaque algorithme de pouvoir être implémenté différemment et/ou sur des structures de données différentes
- aux algorithmes fonctorisés de s'utiliser les uns les autres
- aux utilisateurs d'utiliser ces algorithmes dans des conditions et sur des structures non imaginées par le développeur !

Cette approche est par conséquent particulièrement appropriée pour des bibliothèques d'algorithmes d'un domaine qui travaillent sur des structures de données dont l'implémentation est susceptible de varier.

1.3.3 OCAMLGRAPH

OCAMLGRAPH[1] est une bibliothèque de graphes pour OBJECTIVE CAML écrite au LRI conçue suivant l'approche de la fonctorisation des algorithmes qui fournit les principaux algorithmes sur les graphes ainsi que différentes implémentations de structures graphes.

Comme nous l'avons vu, le « design pattern » utilisé pour obtenir des algorithmes génériques consiste à les mettre sous forme de foncteurs mais aussi à en donner l'implémentation (partielle car elle ne dispose pas encore des implémentations des modules contenant les les structures de données et les fonctions pour les traiter).

OCAMLGRAPH est conforme à cette conception et propose ainsi des foncteurs pour les algorithmes du chemin le plus cours de Dijkstra (module *Path*), les composantes fortements connexes (module *Components*), les flots (module *Flow),* les différents parcours comme DFS (*Depth-First Search*) et BFS (*Breadth-First Search*) (module *Traverse*), etc. Ces algorithmes s'appliquent sur les signatures définies dans le module *Sig.*

Le module *Sig* est divisé en deux groupes de signatures : celles qui sont plutôt générales (utilitaires), et celles qui définissent les objets courants en graphes (métier). Le premier groupe contient par exemple la signature *ORDERED_ TYPE* qui permet d'assurer qu'un élément est comparable (et donc qu'on peut établir un ordre total sur le type) :

```
module type ORDERED_TYPE = sig
    type t
    val compare : t -> t -> int
end
```

Tout module qui définit un type t ainsi qu'une fonction $compare : t \rightarrow t \rightarrow int$ réalise donc la signature *ORDERED_ TYPE.*

De la même façon, le reste du module *Sig* définit les signatures des sommets (module *VERTEX*), arcs et arêtes (module *EDGE*), des graphes qui les utilise (module *G* paramétré par *VERTEX* et *EDGE*) et différentes signatures pour les implémentations en utilisant les fonctionnalités impératives d'OBJECTIVE CAML ou fonctionnelles dont les types diffèrent. L'utilisateur peut choisir ou fournir l'implémentation de ces modules, à moins d'utiliser celle proposée par défaut.

Le module *Sig_pack* permet en effet d'utiliser directement la bibliothèque OCAMLGRAPH en appliquant les foncteurs : une implémentation complète est fournie (elle utilise l'implémentation impérative pour des raisons de performance).

La bibliothèque OCAMLGRAPH propose par sa conception un maximum de flexibilité : le chapitre suivant présente plus en détails deux algorithmes d'OCAMLGRAPH : le chemin le plus court (par Dijkstra) et les composantes fortement connexes.

[1] `http://ocamlgraph.lri.fr`

Chapitre 2

Algorithmes sur les graphes

2.1 Présentation des algorithmes étudiés

2.1.1 Le chemin le plus court de Dijkstra

2.1.1.1 Présentation rapide de l'algorithme

L'algorithme du plus court chemin, comme son nom l'indique, permet de trouver, dans un graphe orienté ayant des poids positifs, le plus court chemin (ou tous les plus courts chemins en fonction de la variante) d'un sommet s à un sommet t.

L'algorithme conserve pour chaque sommet v le coût $d[v]$ du chemin le plus court entre s et v connu à cet instant. Au début de l'algorithme, le coût $d[s] = 0$ et tous les autres sont infinis (c'est-à-dire qu'il n'existe pas de chemin connu à cet instant pour accéder à ce sommet).

Une fois l'algorithme fini, tous les $d[v]$ ont pour valeur le coût de s à v. D'autres variantes de l'algorithme renvoient plutôt le chemin de s à t ainsi que son coût.

2.1.1.2 L'algorithme en pseudo-code

Voici une version de l'algorithme du plus court chemin de Dijkstra en pseudo-code impératif qui calcule ici le plus court chemin vers tous les autres sommets :

```
(* G graphe, w fonction de poids, s le sommet initial *)
function Path(G, w, s)

    (* Initialisations  *)
    for each vertex v in V[G]
         d[v] := infinity
         previous[v] := undefined
    d[s] := 0
    (* sommets visités, initialisé à l'ensemble vide *)
    S := empty set
    (* sommets non visités, initialisé à l'ensemble des sommets de G *)
    Q := V[G]
    while Q is not an empty set
        (* trouver le sommet dont le coût du chemin est le plus faible *)
        u := extract_min(Q)
        (* marquer u comme visité *)
        S := S union {u}
```

```
(* Relâchement (u,v), c'est-à-dire que si on trouve en passant
   par u un meilleur chemin pour aller à v, on l'adopte *)
for each edge (u,v) outgoing from u
    if d[u] + w(u,v) < d[v]
        d[v] := d[u] + w(u,v)
        previous[v] := u
```

2.1.1.3 Description de l'algorithme pas à pas

Soit G le graphe de la figure 2.1 :

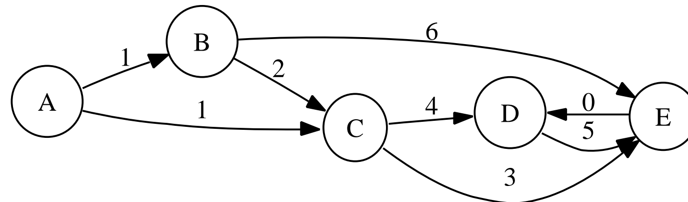

Fig. 2.1 – Le graphe G

Nous allons dérouler l'algorithme du chemin le plus court sur le graphe G de A à E. On utilise la notation suivante : dans les sommets, la valeur $d[v]$ le coût du meilleur chemin connu pour aller à v : le noeud gris clair est le sommet courant, les sommets foncés sont déjà visités.

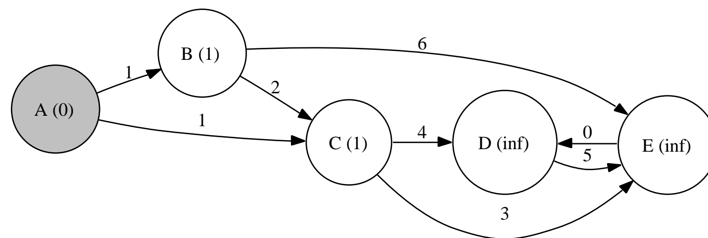

Fig. 2.2 – Recherche du plus court chemin : première étape

Dans cette première étape (Fig. 2.2), le sommet source A explore ses voisins B et C et calcule le coût $d[B] = d[C] = 1$. Leurs coûts étants égaux, il choisit donc l'un ou l'autre, ici B.

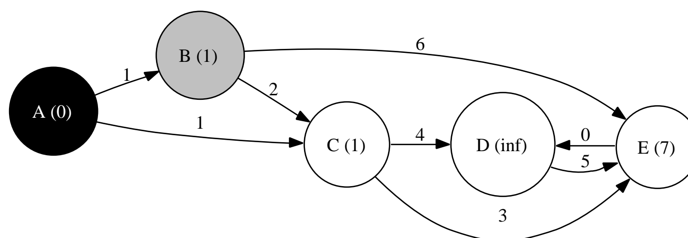

Fig. 2.3 – Recherche du plus court chemin : deuxième étape

Dans la deuxième étape (FIG. 2.3), le somme B explore ses voisins E et C : le relâchement sur E permet de déterminer (pour l'instant) que $d[E] = 7$. En revanche, B ne fournit pas de meilleur chemin pour aller à C car $d[C] = 1 < d[B] + w(B \to C) = 1 + 2 = 3$. C'est donc à partir de C que s'exécutera la prochaine itération de l'algorithme.

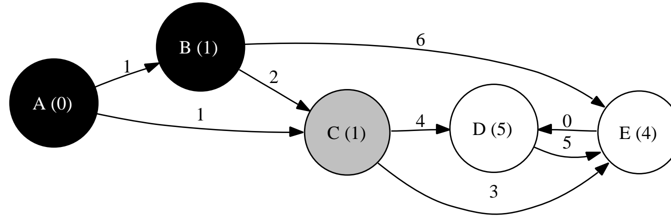

FIG. 2.4 – Recherche du plus court chemin : troisième étape

Le sommet C explore dans la FIG. 2.4 ses voisins non visités D et E et détermine que $d[D] = 5$ et $d[E] = 4$. L'algorithme sait donc ici que le plus court chemin de A à E est $p = A \xrightarrow{1} C \xrightarrow{3} E$ et $d[E] = 4$. La variante de l'algorithme qui retourne le plus court chemin de A à E (et notamment la version d'OCAMLGRAPH) s'arrête donc ici et retourne la paire $(p, d[E])$. La variante qui calcule les chemins les plus cours de la source (ici A) à tous les autres sommets continue depuis E.

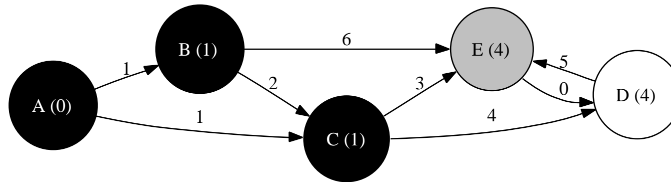

FIG. 2.5 – Recherche du plus court chemin : quatrième étape

Ici (FIG. 2.5), le relâchement sur D permet de déterminer que $d[D] = 4$ par le chemin $A \xrightarrow{1} C \xrightarrow{3} E \xrightarrow{0} D$. Comme D n'a pas été visité, l'algorithme doit explorer ses voisins (il pourrait en avoir qui ne sont ni voisins de A, B, C et E)

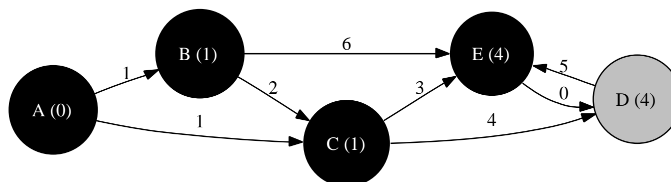

FIG. 2.6 – Recherche du plus court chemin : cinquième étape

On voit dans FIG. 2.6 que D n'ayant pas de voisins sur lesquels le relâchement marche, et comme tous les sommets ont déjà été visités, l'algorithme s'arrête.

2.1.2 Les composantes fortement connexes

2.1.2.1 Présentation rapide de l'algorithme

L'algorithme des composantes fortement connexes trouve l'ensemble des composantes fortement connexes (CFC) d'un graphe, c'est à dire les plus grands ensembles de sommets pour lesquels l'opération $chemin\,existe(a, b : sommet)$ qui détermine s'il existe un chemin de a vers b est réflexive.

L'algorithme parcourt le graphe et pour chaque sommet a, il parcourt une nouvelle fois le graphe à la recherche de sommets b tels que $chemin\,existe(a, b) \wedge chemin\,existe(b, a)$.

2.1.2.2 L'algorithme en pseudo-code

```
(* G graphe, s t sommets, visited ensemble *)
function chemin_existe_aux(G,s,t,visited)
    (* Pour tous les successeurs v de s dans G *)
    for each edge (s,v) outgoing from s
        if v = t then return True
        if not (v in visited) then
            if chemin_existe v t (G,visited union {v})
                return True
    return False.

(* G graphe, s t sommets *)
function chemin_existe(G,s,t)
    chemin_existe(G,s,t,{s})

(* G graphe *)
function components(G)
    (* ensemble de sommets visités *)
    visited := empty set
    (* ensemble d'ensembles de sommets *)
    result  := empty set
    for each vertex v1 in V[G]
        (* si un vertex a déjà été visité, sa composante connexe
           est déjà dans le résultat *)
        if v1 in visited then continue
        (* un sommet est dans sa composante connexe *)
        current_component := {V1}
        visited := visited union {v1}
        (* pour tous les vertex qui ne sont pas déjà liés à
           une composante connexe *)
        for each vertex v2 in (V[G]-visited)
            (* vérifier la réfléxivité de chemin_existe *)
            if chemin_existe(G,v1,v2) and chemin_existe(G,v2,v1) then
                (* v1 et v2 sont dans la même composante *)
                current_component := current_component union {v2}
                visited := visited union {v2}
        (* la composante courante est complête *)
        result := result union {{current_component}}
    return result
```

2.1.2.3 Description de l'algorithme pas à pas

Soit G le graphe de la figure 2.7 :

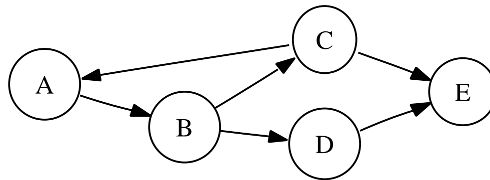

FIG. 2.7 – Le graphe G

On considère la fonction `chemin_existe(a,b)` connue. Dans la suite, les sommets visités sont en gris ou noir, le noir étant utilisé pour ceux qui sont dans la CFC courante, et le gris pour ceux qui n'en font pas partie.

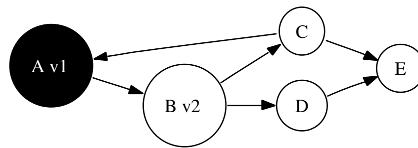

FIG. 2.8 – Composantes fortement connexes : noeuds sélectionnés

Un sommet `v1` est selectionné (A), la CFC courante devient $\{A\}$ et A est ajouté à l'ensemble des sommets visités. Un second sommet est selectionné (B), et on teste l'existance des chemins $A \rightarrow B$ et $B \rightarrow A$. Ces chemins existent donc B est ajouté à la CFC courante et aux sommets visités.

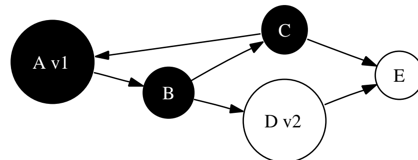

FIG. 2.9 – Composante fortement connexes : ajout de noeuds à une composante

Le sommet C est ajouté à la CFC courante selon le même principe, et l'algorithme choisit alors un autre sommet `v2` (D).

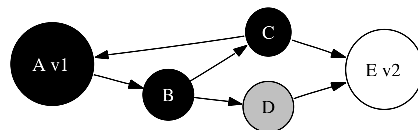

FIG. 2.10 – Recherche de chemin : noeuds rejetés

Il n'existe pas de chemin $D \rightarrow A$, D ne fait donc pas partie de cette CFC. Le sommet E est également rejeté et la première itération sur `v1` se termine. La CFC $\{A, B, C\}$ est alors ajoutée au résultat.

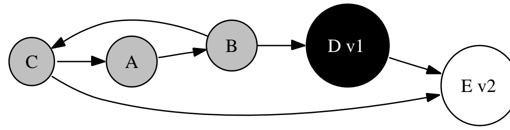

FIG. 2.11 – Recherche de chemin : troisième étape

Une nouvelle valeur est affectée à `v1` (D), et la CFC à laquelle il appartient est déterminée en vérifiant l'existance des chemins $D \rightarrow E$ et $E \rightarrow D$. On peut remarquer que l'algorithme ne teste pas si sommets A, B et C appartiennent à la CFC courante car il ont déjà eté affectés à une CFC lors de la première itération.

L'algorithme ajoute ensuite la CFC $\{D\}$ au résultat puis $\{E\}$ en suivant le même principe. Les trois composantes fortement connexes du graphe $\{A, B, C\}$,$\{D\}$ et $\{E\}$ sont alors retournées par la fonction.

2.2 L'implémentation des algorithmes d'OCAMLGRAPH

Nous détaillons ici les détails de l'implémentation d'OCAMLGRAPH pour les algorithmes du plus court chemin et des composantes connexes.

2.2.1 Le plus court chemin de Dijkstra

Comme nous l'avons vu précédemment, OCAMLGRAPH fournit des signatures, pour abstraire les structures de données qu'utilisent les algorithmes, qui peuvent être implémentées soit par l'utilisateur soit en utilisant les implémentations proposées dans la bibliothèque. Nous allons donc ici détailler l'interaction des foncteurs pour le plus court chemin.

OCAMLGRAPH définit tout d'abord une signature *WEIGHT* :

```
module type WEIGHT = sig
  type label
  type t
  val weight : label -> t
  val zero : t
  val add : t -> t -> t
  val compare : t -> t -> int
end
```

On voit que le module *WEIGHT* fournit le nécessaire à l'algorithme du plus court chemin pour lui permettre de comparer les poids des arcs et d'additionner les poids (pour que l'algorithme puisse calculer le coût des chemins connus). La valeur *zero* permet de s'abstraire complètement des poids « entiers » : il est possible d'utiliser n'importe quel type disposant d'une relation d'ordre bien définie et d'un élément neutre.

On définit ensuite la signature G (pour graphe) :

```
module type G = sig
  type t
```

```
  module V : Sig.COMPARABLE
  module E : sig
    type t
    type label
    val label : t -> label
    val dst : t -> V.t
  end
  val iter_succ_e : (E.t -> unit) -> t -> V.t -> unit
end
```

Le module G définit deux sous-signatures. La première, V, représente les sommets (*vertices*). L'algorithme ne requiert d'eux que le fait qu'ils soient comparables (si jamais les poids de deux chemins de s à deux autres sommets sont égaux). La deuxième signature E représente les arcs (*edges*) et requiert d'eux qu'elles aient un type « *label* », une fonction *label* qui associe un noeud à un *label*, et enfin une fonction *dst* qui donne le noeud distant.

On définit enfin le foncteur *Dijkstra* qui implémente l'algorithme du plus court chemin.

```
module Dijkstra
  (G : G)
  (W : WEIGHT with type label = G.E.label) =
struct
  open G.E

  module H =  Hashtbl.Make(G.V)        (* Une hashtable de sommets *)

  module Elt = struct
    (* Triplet : poids d'un vertex * le vertex en question * chemin pour y parvenir *)
    type t = W.t * G.V.t * G.E.t list
    (* weights are compared first, and minimal weights come first in the
       queue *)
    let compare (w1,v1,_) (w2,v2,_) =
      let cw = W.compare w2 w1 in
      if cw != 0 then cw else G.V.compare v1 v2
  end

 (* Le tas (trié) qui permet de toujours piocher le vertex avec le coût le plus faible *)
  module PQ = Heap.Imperative(Elt)

  let shortest_path g v1 v2 =
    let visited = H.create 97 in
    let q = PQ.create 17 in

    (* le début de l'itération *)
    let rec loop () =
      if PQ.is_empty q then raise Not_found ; (* tous les noeuds visités, v2 inaccessible  *)
      let (w,v,p) = PQ.pop_maximum q in      (* piocher le meilleur coût pour aller à v2  *)
      if G.V.compare v v2 = 0 then           (* si on est arrivé à v2 alors..             *)
        List.rev p, w                        (*    renvoyer le chemin et son poids        *)
      else begin                             (* sinon                                     *)
        if not (H.mem visited v) then begin  (* si le vertex v n'a pas encore été visité  *)
          H.add visited v () ;               (*    le marquer comme visité                *)
                                             (*                                           *)
          G.iter_succ_e                      (* pour chaque successeur s de v dans G,     *)
                                             (*  ajouter dans le tas PQ le triplet :      *)
            (fun e -> PQ.add q (W.add w      (*    - poids du chemin pour y aller         *)
```

```
                (W.weight (label e)),         (*    - s                              *)
                dst e, e : : p)) g v          (*    - le chemin pour y aller         *)
                                              (*                                     *)
            end ;                             (*                                     *)
            loop ()                           (* appel récursif                      *)
          end                                 (*                                     *)
        in                                    (*                                     *)
        PQ.add q (W.zero, v1, []) ;           (* valeurs initiales                   *)
        loop ()                               (* appel à la fonction d'itération     *)
      end
```

Le foncteur requiert deux modules paramètres : G qui doit représenter une structure de graphe telle que définie par la signature G définie plus haut, W qui fournit à l'algorithme ses fonctions de calcul des poids définies dans un module réalisant la signature *WEIGHT* définie plus haut. Par ailleurs, l'algorithme définit un module *Elt* qui réalise la signature *Sig.COMPARABLE* en fournissant une méthode de comparaison des éléments stoqués dans le tas *PQ* (l'équivalent des tableaux d et *previous* dans la présentation de l'algorithme en pseudo-code).

Le déroulement de l'algorithme est très similaire à celui proposé en pseudo-code même s'il est écrit légèrement différemment. On remarquera surtout que toutes les fonctions agissant sur les structures de données passées au foncteur *Dijkstra* appartiennent bien à ses modules paramètres.

L'algorithme utilise ici, pour des raisons de performance, un mélange de code fonctionnel et impératif : en effet les tables de hashage (*Hashtbl*) ainsi que le tas (*Heap.Imperative*) sont des structures impératives qui fonctionnent à l'aide de références, et la fonction *iter_succ_e* qui leur est appliquée produit des effets de bords (c'est-à-dire que l'environnement est changé par l'invoquation de *loop* au sein de *shortest_path*).

Nous verrons par la suite comment transformer cet algorithme en fonctionnel pur (pour éviter les effets de bords et pouvoir le porter facilement dans Coq).

2.2.2 Les composantes fortement connexes

Tout comme l'algorithme du plus court chemin, celui des composantes fortement connexes (*CFC*) est fonctorisé. Cet algorithme est capable de travailler sur tout graphe qui respecte la signature suivante G.

```
module type G = sig
  type t
  module V : Sig.COMPARABLE
  val iter_vertex : (V.t -> unit) -> t -> unit
  val iter_succ : (V.t -> unit) -> t -> V.t -> unit
end
```

La première chose que l'on peut constater est que cette signature est différente de celle utilisée par l'algorithme de Dijkstra. C'est là un des intérêts des signatures : n'éxiger de l'utilisateur que le strict nécessaire. Le module `G` ne définit qu'une sous-signature `V` qui représente les sommets. Comme pour le cas de Dijkstra, les sommets ont seulement besoin d'être comparables. L'utilisateur doit également implémenter les deux fonctions `iter_vertex` et `iter_succ` qui permettent respectivement d'itérer sur les sommets d'un graphe, et les successeurs d'un sommet.

On introduit enfin l'algorithme fonctorisé de la recherche des *CFC* :

```
module Make(G : G) = struct

  module H = Hashtbl.Make(G.V)                          (* Une hashtable de sommets    *)
  module S = Set.Make(G.V)                              (* Un ensemble de sommets      *)

  (* [scc g] associe chaque vertex à une CFC et renvoie une fonction [f]
     telle que [f u=f v] <=> u et v font partie de la même CFC *)
```

```
let scc g =
  let root = H.create 997 in                (* hierarchie sur les sommets              *)
  let hashcomp = H.create 997 in            (* association sommet->int                 *)
  let stack = ref [] in                     (*                                         *)
  let numdfs = ref 0 in                     (*                                         *)
  let numcomp = ref 1 in                    (* numéro de composante                    *)
  let rec pop x = function                  (* Dépiler la pile pour associer des sommets *)
    | (y, w) :: l when y > x ->             (* à leur cfc                              *)
        H.add hashcomp w !numcomp ;         (*                                         *)
        pop x l                             (*                                         *)
    | l -> l                                (*                                         *)
  in                                        (*                                         *)
  let rec visit v =                         (* Trouver la cfc qui contient v           *)
    if not (H.mem root v) then              (* si v n'a pas déja été visité            *)
      begin                                 (*                                         *)
        let n = incr numdfs ; !numdfs in    (* incrémenter numdfs                      *)
        H.add root v n ;                    (* associer v à numdfs dans root           *)
        G.iter_succ                         (* Pour chaque successeur w de v :         *)
          (fun w -> visit w ;               (*   - visiter w                           *)
             if not (H.mem hashcomp w)      (*   - si il n'est pas déjà associé à une cfc *)
               then H.replace root v        (*     on remplace la valeur de v dans root *)
                   (min (H.find root v)     (*     par celle de w si elle est plus petite *)
                    (H.find root w)))       (*     ( elle est plus petite si il existe un *)
          g v ;                             (*     chemin de w vers v )                *)
        if H.find root v = n then           (* Si v n'a pas été remplacé il est le     *)
          (H.add hashcomp v !numcomp ;      (* premier noeud de sa composante.         *)
          let s := pop n !stack in          (* On complète la composante               *)
          stack := s ;                      (*                                         *)
          incr numcomp)                     (*                                         *)
        else stack := (n,v) :: !stack ;     (* Si il a été remplacé, il est mis sur le *)
      end                                   (* haut de la pile.                        *)
  in                                        (*                                         *)
  G.iter_vertex visit g ;                   (* On visite tous les sommets du graphe    *)
  fun v -> H.find hashcomp v                (* On renvoie la fonction à partir de hcomp *)

let scc_list g =                            (* Trouver la liste des cfc                *)
  let scc = scc g in                        (* On récupère la fonction qui associe les *)
  let tbl = Hashtbl.create 97 in            (* noeuds à leur numéro de composante      *)
  G.iter_vertex                             (* Pour chaque sommet v :                  *)
    (fun v ->                               (*   - soit n le numéro de composante de v *)
       let n = scc v in                     (*                                         *)
       try                                  (*   - Ajouter v à la liste de sommets de la *)
         let l = Hashtbl.find tbl n in      (*     composante ayant n pour numéro      *)
         l := v :: !l                  (*                                         *)
       with Not_found ->                    (*                                         *)
         Hashtbl.add tbl n (ref [ v ]))     (*     si elle n'existe pas, la créer      *)
    g ;                                     (*                                         *)
  Hashtbl.fold                              (*                                         *)
    (fun _ v l -> !v :: l) tbl []           (* recupérer toutes les listes de sommets  *)
end
```

2.3 Le passage en fonctionnel pur

Notre but est de prouver des propriétés sur les algorithmes d'OCAMLGRAPH en COQ, et ce dernier ne dispose que d'un support de programmation en fonctionnel pur. Le code d'OCAMLGRAPH étant dans un

mélange des paradigmes impératifs et fonctionnels, nous devons porter le code impératif en fonctionnel. Nous présentons brièvement la méthode utilisée puis nous montrons une implémentation en fonctionnel pur des algorithmes.

2.3.1 Méthodologie

Il faut commencer par identifier les structures sémantiques (fonctions, modules) impératives utilisées par la fonction que l'on veut transformer. On les repère généralement facilement : elles sont de type « unit » puisqu'elles sont souvent appelées sur des références et travaillent avec effets de bords, et c'est « par adresse » que l'utilisateur récupère ensuite le résultat du traitement de sa fonction.

Au contraire, en fonctionnel, toute fonction de traitement est bien de type « utile » et retourne le résultat de son travail à l'utilisateur directement. La première étape pour transformer une fonction impérative Objective Caml en fonctionnel est donc de transformer ces fonctions (et donc parfois les modules et signatures qui les contiennent) qui ont des traits impératifs en fonctions purement fonctionnelles. On commence ainsi par tous les appels de fonctions impératives et on remplace ces appels par des appels à des fonctions équivalentes en fonctionnel.

Bien entendu, le processus nécessite parfois de modifier la « forme » du programme : les fonctions impératives sont utilisées en partie aussi car elles allègent le code.

Enfin, les exceptions doivent être retirées. Il existe plusieurs méthodes, mais quand il n'y a qu'une exception ou qu'on n'a pas besoin de faire la différence entre les exceptions, on peut utiliser le type paramétré « option ». Ce type permet d'indiquer soit si on n'a « rien » (*None*) soit si on a bien un élément (*Some* élément).

Une fois toutes les fonctions (et les modules) impératifs remplacées, il faut les appeler de façon à garder leur résultat et dans le cas d'une fonction récursive, ajouter ces résultats aux arguments de la fonction en tant qu'accumulateurs. De la même façon, il faut remplacer les structures d'itération impératives par des appels récursifs.

2.3.2 Les algorithmes en fonctionnel

2.3.2.1 Le plus court chemin de Dijkstra

Le code étant très similaire, voilà la définition de shortest_path : les modules restent fonctorisés de la même manière.

```
let shortest_path g v1 v2 =
  let rec loop visited queue =
    match queue with
    | [] -> None
    | w,v,p ->   (* prend le premier de la queue : le meilleur chemin disponible *)
          begin
              if G.V.compare v v2 = 0 then
                      Some(List.rev p, w)
              else
              begin
                  let new_visited,new_queue =
                  if not (List.mem v visited)
                  then
                      let visited = v : :visited in


                      (* queue_add effectue un tri par insertion de façon à ce que le
                         « meilleur » chemin soit toujours le premier de la queue
                         List.map fonction définie dans le chapitre 1
```

```
                            succ_e retourne la liste des successeurs de v dans g
                         *)
                         nqueue = queue_add
                                     (List :map
                                         (fun e ->
                                             (W.add w (W.weight (label e)),dst e, e : : p))
                                         (succ_e g v))
                                     tlqueue
                         in
                         visited,nqueue
                     else
                         visited,queue
                         in
                         loop new_visited new_queue
                 end
    in
    loop visited_start  (W.zero, v1, [])
```

2.3.2.2 Les composantes fortement connexes

Pour l'algorithme des composantes fortement connexes fonctionnel, on utilise une signature de graphe une peu différents de celle d'OCAMLGRAPH. Les itérateurs sont remplacés par des fonctions qui renvoient une liste d'éléments. `succ` renvoie la liste des sucesseurs d'un sommet et `vertices` renvoie la liste des sommets du graphe.

```
module type G = sig
  type t
  module V : Sig.COMPARABLE
  val vertices : t -> V.t list
  val succ : t -> V.t -> V.t list
end
```

Après passage au paradigme fonctionnel, le module principal de l'algorithme devient :

```
module Make(G : G) = struct
  module H = Map.Make(G.V)

  let rec loop f c = function
      | [] -> c
      | h : :l -> loop f (f c h) l
  let scc g =
    let rec pop hashcomp numcomp x = function
      | (y, w) : : l when y > x ->
          let nhcomp = (H.add w numcomp hashcomp) in
          (pop nhcomp numcomp x l)
      | l -> (hashcomp,l)
    in
    let rec visit (hashcomp,root,stack,numcomp,numdfs) v =
      if not (H.mem v root) then
        begin
          let numdfs = numdfs + 1 in
          let n = numdfs in
          let root = H.add v n root in
          let (nhcmp,nroot,ns,nnc,nnd) = loop
            (fun c w ->
              let (h,r,n1,n2,n3) = visit c w in
```

```
       if not (H.mem w h) then
         let r = (H.remove v r) in
         let r = (H.add v (min (H.find v r) (H.find w r)) r) in
         (h,r,n1,n2,n3)
       else
         (h,r,n1,n2,n3)
      ) (hashcomp,root,stack,numcomp,numdfs) (G.succ g v)
     in
     if H.find v nroot = n then
      let nhcmp = H.add v nnd nhcmp in
      let (nhcmp,ns) = pop nhcmp nnc n stack in
      (nhcmp,nroot,ns,nnc+1,nnd)
     else
      (nhcmp,nroot,(n,v) : :ns,nnc,nnd)
    end
   else (hashcomp,root,stack,numcomp,numdfs)
  in
  let (resmap,_,_,_,_) = loop
   visit
   (H.empty,H.empty,[],1,0)
   (G.vertices g)
  in
  fun v -> H.find v resmap

 module Int = struct
  type t = int
  let compare : int -> int -> int = compare
 end
 module M = Map.Make(Int)
 let scc_list g =
  let fscc = scc g in
  let tbl = M.empty in
  let hres = loop
   (fun tbl v ->
    let n = fscc v in
    if M.mem n tbl then
     let l   = M.find n tbl in
     let tbl = M.remove n tbl in
     M.add n (v : :l) tbl
    else
     M.add n [v] tbl
   ) tbl (G.vertices g)
  in
  M.fold (fun _ comp l -> comp : :l) hres []
end
```

2.3.3 Les limites de garantie d'Objective Caml

Objective Caml offre certaines garanties : son typeur assure l'utilisateur que son programme est bien typé. De plus, le langage ne propose de pointeurs que sous la forme de références qui sont sûres à l'emploi. Ainsi, l'utilisateur peut se sentir à l'abri des bugs : il est en effet à l'abri de certains types de bugs (très courants en C par exemple).

Malgré tout, certaines questions restent posées :

Le programme termine-t'il ? Le problème de l'arrêt de la machine de Turing étant indécidable, on sait qu'Objective Caml ne peut pas, par lui-même, déterminer si le programme s'arrête. On pourrait peut-être donner à Objective Caml une preuve que ce programme termine, mais cela sort du scope du langage : il en est incapable. Il n'y a donc aucune garantie que le programme termine, et c'est au soin du programmeur de s'en assurer (ce qui peut s'avérer être une tâche fastidieuse à la main).

Le programme est-il correct vis-à-vis de sa spécification ? Le programme fait-il ce qu'il est censé faire ? Là encore, Objective Caml n'offre aucune garantie : il ne sait pas ce qu'un programme est censé faire, et encore moins s'il le fait bien ou pas !

On va donc voir comment obtenir ces garanties pour nos algorithmes grâce à Coq.

Chapitre 3

Implémentation en Coq

3.1 Présentation de l'assistant de preuve Coq

3.1.1 L'automatisation de la preuve

Au début du vingtième siècle, l'automatisation des preuves mathématiques est devenue un grand défi pour mathématiciens et logiciens confondus.

Hélas, en 1931, le théorème d'incomplétude de Gödel coupe court aux spéculations en prouvant qu'il est indécidable de prouver une proposition dans l'arithmétique de Peano et que le système reste complet et consistant. Ce faisant il sera impossible de créer un programme qui prouve toutes les propositions mathématiques automatiquement. C'est le départ d'un nouveau domaine de recherche : les preuves assistées par ordinateur.

La volonté d'automatiser dans un certain degré le processus de preuve des propositions mathématiques s'est traduite en 1936 par l'introduction des calculs de séquents. Le procédé est basé sur des règles de réécriture simples et facilement vérifiables par la machine.

La théorie a depuis évolué, notamment grâce à la découverte de l'isomorphisme de Curry-Howard qui est la base du système de types de l'assistant de preuve Coq.

3.1.2 L'assistant de preuve Coq

Coq est un assistant de preuve qui permet des vérifier qu'une preuve est correcte et aide donc à la réalisation de preuves mathématiques. En effet, le noyau dur de Coq est son vérificateur de types qui décide si un terme est bien typé. Grâce à l'isomorphisme de Curry-Howard, le vérificateur de types vérifie aussi bien le type des fonctions que la validité les preuves. L'utilisateur guide le système vers la solution en sélectionnant une branche de l'arbre de preuve à l'aide de tactiques appropriées.

Ce système permet de minimiser les risques d'erreur humaine, et d'exhiber une preuve à l'utilisateur sceptique.

Coq ne se limite pas aux preuves mathématiques, mais permet aussi la preuve de programmes, c'est-à-dire prouver que ce dernier est conforme à sa spécification. Un programme prouvé en Coq peut être exporté vers d'autres langages de programmation.

3.1.3 Rappels de logique

Les règles d'inférence en logique sont à la base le fruit du raisonnement sur la « vérité » des philosophes puis des logiciens. Ainsi, la célèbre formule « Si Socrate est un homme et que les hommes sont mortels, alors Socrate est mortel » (appelée Modus Ponens) constitue la règle dite de « coupure » (*Cut*) qui

permet de découper un séquent qui a comme conclusion « Socrate est mortel » (c'est-à-dire la propriété $Mortel(Socrate)$ est vraie) en deux nouveaux séquents dont les conclusions sont $Homme(Socrate)$ est vraie et $\forall h, Homme(h) \rightarrow Mortel(h)$ (pour être précis, la coupure dit que Socrate est « quelque chose » de mortel, il faut donc trouver une propriété P telle que $P(Socrate)$ est vraie et $\forall x, P(x) \rightarrow Mortel(x)$: la propriété $Homme$ satisfait la propriété P car son type est réductible à P).

Un séquent est un couple contenant d'une part un ensemble de formules appelées hypothèses et d'autre part une formule appelée conclusion. Dans le système de déduction naturelle, un séquent est noté $H_1, \ldots, H_n \vdash f$ et affirme que f est vraie sous les hypothèses $H_1, \ldots, H_n$.

Une rêgle de déduction affirme que si les jugements $J_1, \ldots, J_n$ sont vrais, alors un jugement J est vrai. une telle règle est notée : $\dfrac{J_1, \ldots, J_n}{J}$ ou bien $J_1, \ldots, J_n \models J$.

Les règles de déduction naturelles pour la logique propositionnelle sont des règles de déductions sur les séquents. Le système de déduction naturelle est définit par les rêgles de déduction suivantes accompagnées de règles pour la conjonction et la disjonction non rappelées ici :

$$\textsc{Axiom}\ \frac{}{\Gamma, A \vdash A} \qquad \rightarrow_E \frac{\Gamma \vdash A}{\Gamma, A \rightarrow B \vdash B} \qquad \rightarrow_I \frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$$

$$\textsc{Cut}\ \frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash B} \qquad \neg_E \frac{\Gamma \vdash A}{\Gamma, \neg A \vdash C} \qquad \neg_I \frac{\Gamma, A \vdash False}{\Gamma \vdash \neg A}$$

L'assistant de preuve Coq utilise par défaut la déduction naturelle et logique intuitionniste pour déterminer si une propostion est vraie ou non. Ce système de déduction de la logique intuitioniste est correct, c'est-à-dire qu'il permet de prouver les formules intuitionistes mais il n'est pas complet du point de vue de la logique propositionnelle. En particulier, la tautologie $\neg\neg A \rightarrow A$ n'est pas prouvable sauf si l'on ajoute l'axiome du tiers exclu : $\dfrac{}{\Gamma \vdash A \vee \neg A}$.

Il est intéressant de remarquer que sans Axiom et Cut, les règles de déduction sont réparties entre celles qui permettent l'introduction d'un connecteur $(\rightarrow, \wedge, \vee, \neg, ..)$ et celles qui permettent son elimination.

3.1.4 Spécification et propositions

Les principaux types de Coq sont *Type*, *Set* et *Prop*. *Set* et *Prop* sont eux mêmes de type *Type*, et distinguent deux types de termes fondamentaux : les spécifications et les propositions.

Les termes de type *Set* sont appelés des spécifications, et les instances de ces types sont des programmes.

```
Definition natop : Set := nat -> nat -> nat.
```

natop est un exemple de spécification pour un opérateur sur le type *nat* des entiers naturels. Cette spécification est bien de type *Set*, et définit une contrainte sur les arguments ainsi que sur le type retourné, à savoir prendre deux arguments de type *nat* et retourner un terme de type *nat*.

Un opérateur vérifiant cette spécification pourrait être l'opération *maximum* qui renvoie le plus grand de deux entiers.

```
Definition max natop := fun (m n :nat) =>
   if lt_nat_dec m n then
      n
   else
      m.
```

Il est à noter que ce qu'on appelle *spécification* en COQ est en fait une contrainte sur les types (et non pas ce qui est habituellement compris : une contrainte sur la correction d'un programme). Ces contraintes sont de type $Prop$, et correspondent aux prédicats de la logique propositionnelle.

Dans l'exemple précédent, max correspond bien au type $natop$, mais nous avons bien peu d'informations sur la sémantique de la fonction voulue. Nous allons donc augmenter la spécification de la fonction voulue avec une proposition.

```
Definition max_spec : Prop :=
   forall a b :nat, (a <= b -> max a b = b) /\ (a >= b -> max a b = a).
```

On définit ici un prédicat sur la fonction max, lequelle exprime que l'on veut une fonction qui renvoie le maximum des deux arguments.

3.1.5 Les preuves

Pour le moment nous nous sommes uniquement occupés de définir une fonction et un prédicat qui définit la sémantique de la fonction. Ceci pourrait être fait dans n'importe quel langage de programmation avec des assertions, mais l'intérêt de COQ est de pouvoir prouver statiquement que le programme se conforme à sa spécification.

De la même manière que les termes de type Set sont des type des programmes, les termes de type $Prop$ sont des types de preuves. Une fois une proposition définie, tout terme qui a pour type cette proposition est une preuve de cette proposition.

Soit la tautologie suivante :

```
Definition trivial_tautology : Prop :=
   forall P Q R :Prop, (P -> Q)->(Q->R)->P->R.
```

Prouver cette proposition revient à exhiber un terme du type `simple_tautology`.

```
Lemma trivial_tautology_true : trivial_tautology.
Proof fun (P Q R :Prop) (f1 :P->Q) (f2 :Q->R) (p :P) =>
     f2 (f1 p).
```

Le terme `trivial_tautology_true` constitue une preuve de la proposition `trivial_tautology`.

3.1.6 Mécanismes des preuves

Un aspect intéressant de COQ est que le système de typage qui vérifie qu'une fonction est bien typée est le même que celui qui vérifie qu'une preuve est correcte. La notation utilisée pour représenter les types correspond volontairement au sens habituel utilisé en logique propositionnelle.

Il y a équivalence entre les règles de déduction naturelle et les règles de typage de COQ, par exemple dans le cas du modus ponens, la règle de logique Cut correspond à la règle d'application d'une fonction App.

$$\textsc{Cut}\ \frac{\Gamma \vdash P \qquad \Gamma, P \vdash Q}{\Gamma \vdash Q} \qquad\qquad \textsc{App}\ \frac{E, \Gamma \vdash t : P \qquad E, \Gamma \vdash t' : P \rightarrow Q}{E, \Gamma \vdash t'\ t : Q}$$

la règle App se formule de la manière suivante. Si t est une preuve de P est t' une preuve de $P \rightarrow Q$, alors l'application de t' à t est une preuve de Q.

Les variables déclarée dans l'environnement et le contexte sont respectivement des axiomes et des hypothèses qui peuvent être utilisées au cours d'une preuve avec la règle Var.

$$\text{AXIOM}\ \frac{}{\Gamma \vdash P} \qquad\qquad \text{VAR}\ \frac{(x,p) \in E \cup \Gamma}{E, \Gamma \vdash x : P}$$

La rêgle d'introduction de l'implication est disponible avec *Lam*.

$$\rightarrow_I \frac{E, \Gamma, P \vdash Q}{E, \Gamma \vdash P \rightarrow Q} \qquad\qquad \text{LAM}\ \frac{E, \Gamma, (H : P) \vdash t : Q}{E, \Gamma \vdash fun\, H : P \Rightarrow t : P \rightarrow Q}$$

Prouver $P \rightarrow Q$ c'est trouver un terme de type $P \rightarrow Q$. Autrement dit, c'est créer une fonction qui prenne une preuve de P en argument et renvoie une preuve de Q. On cherche donc à obtenir une preuve de Q en supposant que l'on dispose d'une preuve de P.

3.1.7 Utilisation de tactiques

L'utilisation de tactiques permet de faire des preuves de manière naturelle sans se préoccuper de la recherche du terme approprié. Les deux rêgles énoncées précédement se traduisent par l'introduction des tactiques de base `apply` pour *App*, `assumption` pour *Var* et `intro` pour *Lam*.

Nous reprenons la tautologie présentée plus haut, et la prouvons maintenant avec les tactiques présentées.

```
Lemma trivial_tautology_true : (P -> Q)->(Q->R)->P->R.
```

En mode interactif, ou avec CoqIde[1], un séquent est présenté qui nous permet de suivre l'évolution de la preuve. Les séquents sont notés avec le contexte au dessus du terme à prouver.

Le séquent $E, \Gamma \vdash P$ est donc affiché de la manière suivante : $\frac{E, \Gamma}{P}$

Dans toute la suite on considère que l'environnement est vide, c'est à dire que $E = \emptyset$

Le séquent initial est le suivant :

```
   ============================
   forall P Q R : Prop, (P -> Q) -> (Q -> R) -> P -> R
```

On applique la tactique `intros`, qui applique *Lam* autant de fois que necessaire pour définir les hypothèses données.

```
intros P Q R f g p.

   1 subgoal
   P : Prop
   Q : Prop
   R : Prop
   f : P -> Q
   g : Q -> R
   term : P
   ============================
   R
```

Il ne reste plus qu'à appliquer f puis g à *term* pour obtenir R en hypothèse.

[1]L'environnement de développement intégré pour Coq

```
apply f in term.

   1 subgoal
   P : Prop
   Q : Prop
   R : Prop
   f : P -> Q
   g : Q -> R
   term : Q
   ============================
   R

apply g in term.

   1 subgoal
   P : Prop
   Q : Prop
   R : Prop
   f : P -> Q
   g : Q -> R
   term : R
   ============================
   R
```

À ce stade, la proposition à prouver est en hypothèse, il ne reste qu'à terminer la preuve.

```
assumption.

  Proof Completed
```

L'arbre de la peuve réalisée est le suivant :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, (term : R) \vdash R}\ \text{ASSUMPTION}}{\Gamma, (Q : Prop), (R : Prop), (g : Q \to R), (term : Q) \vdash R}\ \text{APPLY G}}{\Gamma, (P : Prop), (Q : Prop), (f : P \to Q), (term : P) \vdash R}\ \text{APPLY F}}{\Gamma \vdash \forall P, Q, R : Prop, (P \to Q) \to (Q \to R) \to P \to R}\ \text{INTROS}$$

Les tactiques `intros`, `apply`, et `assumption` que nous venons de présenter constituent un système complet du point de vue de la logique minimale. Cependant de nombreuses autres tactiques existent, qui permettent de prouver de manière concise les propositions les plus variées. Le tableau suivant donne les tactiques correspondant à l'introduction et l'élimination des connecteurs logiques.

Connecteur	Introduction	Élimination (H :P en hypothèse)
$False$		ELIM H
$A \to B$	INTRO	APPLY H
$A \wedge B$	SPLIT	ELIM H
$A \vee B$	LEFT, RIGHT	ELIM H
$\neg A$	INTRO	APPLY H

3.1.8 Constructions inductives

COQ est un language purement fonctionnel qui dispose de possibilités de recursion basées sur le calcul des constructions inductives. Il est possible de définir des types récursifs représentants des ensembles infinis, puis des fonctions récursives sur ces types.

La définition d'un type inductif se fait en donnant les constructeurs de ce type, ceux ci prenant au besoin des arguments.

Les entiers naturels sont définis de la manière suivante dans COQ :

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

L'ensemble des entiers naturels est défini récursivement à partir de zéro et de l'opération S, qui renvoie le successeur d'un entier.

Un principe d'induction est créé par COQ en même temps que le type inductif pour permettre de raisonner par récurrence.

Des fonctions récursives peuvent être définies dans COQ pour travailler sur les types inductifs. Les calculs sont alors réalisés avec les règles de $\iota-conversion$. À titre d'exemple, on donne la définition de l'opérateur *plus* sur les entiers naturels.

```
Fixpoint plus (n m :nat) {struct n} : nat :=
   match n with
   | O => m
   | S p => S (plus p  m)
   end.
```

plus est définie comme une fonction recursive sur les entiers naturels. Il est a noter que l'argument n joue un role particulier dans la définition de cette fonction, nous aborderons cet aspect dans une prochaine partie.

3.1.9 Les foncteurs dans COQ

Tout comme OBJECTIVE CAML et de nombreux langages fonctionnels, COQ permet d'utiliser des foncteurs. La syntaxe est similaire, avec quelques nuances. La principale différence est la possibilité d'ajouter des hypothèses dans les signatures que les modules doivent prouver à travers un lemme.

A titre d'exemple, nous présentons les modules et foncteurs pour le chemin le plus court de Dijkstra.

Voici la signature de *WEIGHT* en COQ :

```
Module  Type WEIGHT.
  Parameter label : Set.
  Parameter t : Set.
  Parameter weight : label -> t.
  Parameter zero : t.
  Parameter add : t -> t -> t.
  Parameter compare : t -> t -> comparison.
End WEIGHT.
```

On remarque que la notation *val* en OBJECTIVE CAML devient *Parameter* en COQ, et qu'il faut bien préciser qu'on est dans le type *Set* ou *Prop*. Voici une implémentation W de *WEIGHT* :

```
Module W < : WEIGHT.
 Parameter label : Set.
 Definition t : Set := nat.
 Definition weight (l : label) := 1.
 Definition zero := 0.
 Definition add (w1 : t) (w2 : t) := w1+w2.
 Definition compare  (w1 : t) (w2 : t) := nat_compare w1 w2.
End W.
```

Cette implémentation utilise les entiers naturels comme poids. La fonction *nat_compare* compare deux naturels. Ici dans l'implémentation, tous les poids pèsent 1 mais l'utilisateur est libre de gérer les poids.

Tout comme dans OCAMLGRAPH, nous créons une signature des types modules comparables, ici pour les sommets des graphes.

```
Module Type COMPARABLE.
 Parameter t : Set.
 Parameter compare : t -> t -> comparison.
End COMPARABLE.
```

Nous définissons maintenant la signature pour un arc. Elle déclare un module V comparable (le module des sommets) pour pouvoir utiliser leur type.

```
 Module Type EDGE.
    Declare Module V : COMPARABLE.
    Parameter t : Set.
    Parameter label_t : Set.
    Parameter label : t -> label_t.
    Parameter dst : t -> V.t.
    Parameter src : t -> V.t.
    Hypothesis no_self_loop  : forall e :t,  V.compare (dst e) (src e) <> Eq.
 End EDGE.
```

Afin d'illustrer la possibilité d'ajouter des preuves aux modules, nous avons ajouté une condition dans la signature d'*EDGE* : que le module qui l'implémente ne permette pas de créer des « self-loops » sur les sommets. Nous verrons plus loin comment se passe la preuve de l'hypothèse.

Voici la signature du graphe (*VSet* étant un ensemble de *COMPARABLE.t*) :

```
Require Import List.
Module Type GRAPH.
  Parameter t  : Set.
  Declare Module V : COMPARABLE .
  Declare Module E : EDGE.
  Parameter succ_e : t ->  V.t -> list E.t.
  Parameter get_vertices : t -> VSet.t.
  Parameter get_edges : t -> list E.t.
End GRAPH.
```

La signature du graphe ne précise ici que ce dont a besoin l'algorithme de Dijkstra : la fonction *succ_e* qui retourne la liste des arcs vers les successeurs d'un noeud.

Nous définissons désormais une implémentation des sommets :

```
Module VERTEX < : COMPARABLE.
 Definition t : Set := nat.
 Definition compare (v1 : t) (v2 : t) := nat_compare v1 v2.
End VERTEX.
```

Puis des arcs :

```
Module EDGE_IMP < : EDGE with Module V := VERTEX.
   Module V := VERTEX.
   Import V.
   Definition t : Set := prod V.t V.t.
   Definition label_t : Set := nat.
   Definition label (e : t) := 1.
   Definition dst (e : t) := snd e.
   Definition src (e : t) := fst e.
   Lemma no_self_loop  : forall e :t,  V.compare (dst e) (src e) <> Eq.
```

Avant de pouvoir terminer cette implémentation, il faudrait donc prouver le lemme *no_self_loop* sur notre structure de données, ce qui est impossible dans notre cas car les arcs sont des paires de sommets, et rien dans la construction des paires en Coq (type *prod*) ne garantit cette propriété. On illustre ici une façon qu'aurait un algorithme fonctorisé d'établir des pré-conditions si il en a besoin pour garantir sa correction : ce n'est pas le cas de l'algorithme de Dijkstra qui fonctionne avec les « self-loops », l'hypothèse et le lemme *no_self_loop* n'ont donc rien à faire ni dans la signature, ni dans l'implémentation !

Nous présentons enfin une implémentation de la signature du graphe :

```
Module G < : GRAPH with Module V := VERTEX with Module E := EDGE_IMP.
    Module V := VERTEX.
    Module E := EDGE_IMP.

    Definition t : Set := prod (VSet.t) (list E.t).
    Definition get_vertices (G :t) := fst G.
    Definition get_edges (G :t) := snd G.
    Definition succ_e (G :t) (v :V.t) : list E.t :=
    let succerec := fix succerec (sle :list E.t) (sv :V.t) {struct sle} : list E.t :=
      match sle with
      | nil => nil
      | e : :t => match V.compare (E.src e) sv with
                 | Eq => e : :(succerec t sv)
                 | _ => succerec t sv
                 end
      end
      in
      succerec (snd G) v.
     Check succ_e.
End G.
Definition g := G.t.
```

On a donc ici réussi à garantir statiquement, avec la notation « with Module », que le type des sommets traités par V dans G est le même que celui des sommets traités par E : la preuve, on peut utiliser la fonction de comparaison $V.compare : V.t \rightarrow V.t \rightarrow comparison$ sur un $V.t$ et un $E.V.t = V.t$.

Par ailleurs, on a vu comment les foncteurs pouvaient être utilisés pour garantir la sémantique de la signature (l'implémenter comme on le voulait) de la façon optimale en spécifiant les pré-requis des algorithmes qui travailleront sur le module.

3.2 Terminaison des programmes

Pour prouver la terminaison de nos algorithmes, nous devons guider Coq.

Remarque : En Coq, une fonction n'est définie (et donc utilisable) que si elle termine (c'est-à-dire que Coq dispose d'une preuve qu'elle termine).

3.2.1 Récurrence structurelle

On sait que certains programmes terminent : ceux qui n'ont aucune itération. La question de la terminaison ne se pose donc que pour les fonctions récursives en Coq (puisque c'est le seul moyen de faire des itérations en Coq). Coq n'a bien entendu aucune difficulté pour savoir si une fonction est récursive ou non, il suffit de regarder si sa définition contient un appel récursif.

Il faut donc une manière de prouver qu'une fonction récursive (ou itérative en général) termine. On raisonne ainsi : si cette fonction termine à un moment donné, alors c'est qu'il existe un invariant I vrai durant l'itération et faux une fois l'itération terminée (en Coq comme l'itération s'exprime avec des fonctions récursives, la fonction termine quand l'itération termine). Prouver la terminaison, c'est prouver que l'invariant va devenir faux quelle que soit l'entrée.

En Coq, beaucoup d'objets manipulés sont des structures récursives définies par induction comme les listes ou les entiers naturels vus précédemment. Il est facile pour Coq de prouver la terminaison des récursions sur de tels objets car un invariant est facile à trouver : $I \neq cas_base$ (où cas_base est la valeur de base de l'induction, 0 pour les entiers, *nil* pour les listes) à condition que chaque appel récursif se fasse sur un « sous-terme » de l'objet sur lequel on récurse. Notons par ailleurs qu'en Coq, tous les tests *(if-then-else*, *match*, ...) sont complets grâce au typeur qui l'exige, donc pour tout terme inductif on est garanti d'avoir le cas de base. C'est donc une preuve par récurrence structurelle que Coq fait.

Par exemple, Coq prouve seul la terminaison de la fonction *map* vue au premier chapitre :

```
Fixpoint map (A B :Set) (l :list A) (f : A->B) {struct l} : list B :=
match l with
| nil => nil
| hd : :tl => (f hd) : :(map tl f)
end.
```

On remarque qu'il faut spécifier la variable sur laquelle on fait la récurrence structurelle dans une enclosure `{struct} pour que` Coq soit capable de faire les vérifications dessus (il peut y avoir plusieurs arguments dans la fonction mais pas dans le bloc `{struct}` donc il faut bien choisir la bonne variable). Ces définitions de fonctions à un point fixe sont nommées conséquemment *Fixpoint* en Coq.

Hélas, toutes les fonctions ne font pas de récursion sur un seul argument donc il est impossible de prouver leur décroissance structurelle ainsi. C'est le cas des algorithmes du plus court chemin de Dijkstra et des composantes fortement connexes.

3.2.2 Nos algorithmes en Coq et leur terminaison

3.2.2.1 Le plus court chemin de Dijkstra

L'algorithme en Coq

L'algorithme en Coq est exactement le même qu'en Objective Caml (avec la syntaxe de Coq) mais a du subir quelques modifications pour les besoins de la preuve de la terminaison. Tout d'abord, pour établir une mesure bien fondée sur le déroulement de l'algorithme, et comme celle-ci ne peut s'appliquer que sur un seul argument du fait de limitations de la notation *Function*, les arguments « visited » et « queue » de l'algorithme fonctionnel pur en Objective Caml ont été regroupés en un argument sous forme de paire. La paire est donc cassée au début de l'algorithme. Un test inutile à l'algorithme (est-ce qu'un sommet appartient bien au graphe !) est ajouté pour les besoins de la preuve pour qu'elle dispose de l'hypothèse « le sommet v est bien dans le graphe ». Enfin, on utilise le foncteur *set_list* de la bibliothèque Coccinelle[2] pour définir l'ensemble des sommets visités sans redondance (module *VSet*). Notons que la variable g est accessible car définie de type $G.t$ dans la section.

[2] `http://www.lri.fr/~contejea/Coccinelle/`

```
Function loop  (v2 : V.t) (vq : VSet.t * (list QueueElt.t))
{ wf om vq } : option (prod (list E.t) W.t) :=
match vq  with
| (visited, queue) =>
   match queue with
   | nil => None
   | (w,v,p) : :tlqueue =>
           match V.compare v v2 with
           | Eq =>  Some((List.rev p), w)  (* algo fini *)
           | _ =>
           (
              let (new_visited,new_queue) :=
              if     (VSet.mem_dec v visited
              then  (
                    (visited,tlqueue)
               )
               else (* sinon continuer *)
               (
                  if (VSet.mem_dec v (G.get_vertices g)) then (
                     let nqueue :=
                       Queue.add
                        (List.map
                          (fun (e :E.t) =>
                            (W.add w (W.weight (E.label e)),
                              E.dst e, e : : p))
                      (G.succ_e g v)) tlqueue
                     in
                     (VSet.add v visited,nqueue)
                   )
                   else
                      (visited,tlqueue)
              )
              in
              loop v2 (new_visited, new_queue)
))
   end
end.


Definition shortest_path (g :G.t) (v1 :V.t) (v2 :V.t) :=
   let visited_start := VSet.empty  in
    loop v2 (visited_start,((0, v1, nil) : :nil)).
```

Nous allons donc voir plus précisément à quoi correspond la notation *Function* ainsi que l'enclosure `{ wf om vq }`.

La notation *Function* et les relation bien-fondées

Il existe plusieurs méthodes pour définir une fonction récursive qui ne peut s'écrire en *Fixpoint* en Coq : on a décidé au cours du stage qu'on écrirait l'algorithme du plus court chemin de Dijkstra avec la notation *Function*. Cette notation cherche à reprendre la syntaxe de *Fixpoint*, c'est-à-dire qu'il y a une enclosure `{}` pour dans laquelle on choisit comment on souhaitera prouver la terminaison parmi trois possibilités :

– `{struct l}` utilise la meme tactique que pour *Fixpoint*. Dans ce cadre, Function équivaut à *Fixpoint* (il faut donc bien entendu que l'argument récursif soit un sous-terme de l, la structure sur laquelle on récurse)

– `{measure f l}` est une méthode qui exige une fonction f qui prend en paramètre l le (ou les, sous forme de n-uplets) arguments et demande, pour prouver la terminaison de prouver $\forall entree, corps_fonction \rightarrow f(param_l_appel_recursif) < f(param_l_appel_courant)$. Voici une présentation *measure* en reprenant un exemple de la fonction *map* légèrement modifié pour qu'il ne passe plus avec *Fixpoint* (car Coq ne reconnaît plus que l'appel récursif se fait sur un sous-terme de la variable de récursion) :

```
Function map (l :list nat) {measure length l} : list nat :=
match (l++nil) with
| nil => nil
| hd : :tl => (hd) : :(map tl)
end.
```

On voit qu'on utilise ici la mesure évidente « length » qui renvoie la longueur de la liste l. On utilise cette fonction car on devine qu'on peut prouver que $length(tl) < length(l)$. Elle génère donc pour but :

```
1 subgoal
============================
forall (l : list nat) (hd : nat) (tl : list nat),
l ++ nil = hd : : tl -> length tl < length l
```

– `{wf f l}` est une méthode qui exige une fonction f qui prend en paramètre deux arguments, ici des listes, du type de l : l'argument qui est paramètre de l'appel recursif qui correspond à l'argument l, appelons-le tl, et l'argument l de l'étape courante. Cette méthode est plus générale que *measure* : on peut ici choisir la fonction de comparaison « • » pour prouver $f(tl) \bullet f(l)$ alors que pour *measure,* • correspond à <. Il faut néanmoins prouver que • est une relation bien-fondée (d'où le nom *wf* pour *well-founded*). Pour prouver la terminaison en utilisant *wf,* il faut donc prouver $\forall entree, code_fonction \rightarrow f(param_l_appel_recursif, param_l_appel_courant) \wedge well_founded\, f$. Voici le même exemple de *map* avec *wf* :

```
Function map (l :list nat)
{wf (fun (a :list nat) (b :list nat) => length a  < length  b)  l} : list nat :=
match (l++nil) with
| nil => nil
| hd : :tl => (hd) : :(map2 tl)
end.
```

On a utilisé ici une fonction anonyme qui correspond strictement à ce que fait *measure* puisque la fonction de comparaison • =< avec l'application de *length* comme dans l'exemple précédent. Voici les buts générés :

```
2 subgoals
============================
forall (l : list nat) (hd : nat) (tl : list nat),
l ++ nil = hd : : tl -> length tl < length l

============================
well_founded (fun a b : list nat => length a < length b)
```

Pour déterminer la méthode à employer, il faut analyser l'algorithme et trouver la plus appropriée : la plus simple qui permette de prouver la terminaison. Pour ce faire, on regarde si pour chaque pas de l'algorithme le ou les arguments ont bien une décroissance structurelle : il faut établir une relation bien fondée sur tous les arguments qui changent à chaque récursion. Dans le cas de Dijkstra, à chaque itération, la taille de *queue* diminue d'un élément, puis soit *visited* augmente de taille et dans ce cas *queue* aussi (autant que le nombre d'arcs sortants du noeud courant, soit de 0 à $nb_sommets\ (g) - 1$), soit *visited* n'augmente pas et dans ce cas on est sur que *queue* a diminué de taille. On sait donc que *Fixpoint* ne convient pas car on n'a pas de décroissance structurelle sur un argument unique, *measure* non plus, car il est impossible de définir une décroissance en utilisant une fonction sur les arguments traités. On doit donc utiliser *wf* en définissant comme fonction de comparaison la comparaison par ordre lexicographique

dont les éléments sont les paires $(cardinal(diff(sommets(g)\, visited)), length(queue))$ car on sait que *visited* ne fait que croître et à la fin de l'algorithme *queue* est vide : prouver la terminaison, c'est prouver qu'à chaque pas que la mesure sur l'ordre lexicographique décroît.

On a donc trouvé là (et non sans efforts) une décroissance qui nous permettra de prouver que notre algorithme termine. Il faut bien comprendre la démarche qu'on a suivie : on doit choisir une relation bien-fondée accompagnée d'arguments à lui passer qu'on est sûrs (dans la mesure du possible) de pouvoir prouver à partir du code de la fonction !

Preuve de terminaison

Une fois la fonction passée dans le compilateur, Coq nous génère donc ce but :

```
2 subgoals
g : graph
============================
forall (v2 : V.t) (vq : VSet.t * (list QueueElt.t))
  (visited : VSet.t) (queue0 : (list QueueElt.t)),
vq = (visited, queue0) ->
forall (p0 : W.t * V.t * list E.t)
  (tlqueue : list (W.t * V.t * list E.t)),
queue0 = p0 : : tlqueue ->
forall (p1 : W.t * V.t) (p : list E.t),
p0 = (p1, p) ->
forall (w : W.t) (v : V.t),
p1 = (w, v) ->
forall anonymous : v <> v2,
v == v2 = right (v = v2) anonymous ->
forall (new_visited : VSet.t)
  (new_queue : (list QueueElt.t)),
(if VSet.mem_dec v visited
 then (visited, tlqueue)
 else
  if VSet.mem_dec v (G.get_vertices g)
  then
   let nqueue :=
     Queue.add
        (List.map (fun e : E.t =>
          (W.add w (W.weight (E.label e)), E.dst e, e : : p))
             (G.succ_e g v)) tlqueue in
   (VSet.add v visited, nqueue)
  else (visited, tlqueue)) = (new_visited, new_queue) ->
om (new_visited, new_queue) (visited, (w, v, p) : : tlqueue)

============================
well_founded om
```

La fonction *om* est définie comme une fonction de comparaison qui utilise l'ordre lexicographique sur les paires $(cardinal(diff(sommets(g)\, visited)), length(queue))$. On doit donc prouver ici que le code de la fonction implique que l'ordre diminue et que *om* est un ordre bien défini pour prouver la terminaison de l'algorithme.

Donner la preuve n'a que peu d'intérêt car elle n'est compréhensible qu'avec les hypothèses à chaque pas, et elle est trop longue. Réaliser une premiere preuve de programme en Coq est une tâche ardue qui a plus d'intérêt, surtout pour une fonction dont on sait bien sûr qu'elle termine comme l'algorithme de Dijkstra, dans le fait qu'elle doive être prouvée pour être définie en Coq et utilisable dans une bibliothèque mais

aussi et surtout pour l'apprentissage et la compréhension du langage, de ses principes et des techniques à employer : il s'agit d'un passage obligé.

La preuve de la terminaison a donc bien été réalisée mais à cause d'un bug de l'implémentation de *Function* qui est assez récente (et qui n'a vraissemblablement pas été prouvée !), elle n'a pas pu être vérifiée par COQ à ce moment. On a donc repris la définition de *loop* pour faire le travail de *Function* manuellement et définir le but directement. Une description plus complète de la méthode sera donnée dans la section 3.2.2.2 avec la preuve de la terminaison de l'algorithme des composantes fortement connexes qui l'utilise aussi (afin de présenter les différentes méthodes, même si la *Function* s'avère en fait être un « raccourcis » pour l'autre méthode).

A l'aide de cette méthode, et en reprenant la même preuve (car c'est bien le même but), la preuve de terminaison de l'algorithme de Dijkstra a bien été vérifiée par le vérificateur de types de COQ : on a donc une nouvelle garantie sur cette algorithme, et il est bien défini en COQ.

3.2.2.2 La recherche des composantes fortement connexes

L'algorithme en COQ

La première chose que le lecteur pourra constater est que cet algorithme ne correspond pas à l'implémentation fonctionnelle de l'algorithme d'OCAMLGRAPH. Ce fut pourtant l'objet d'une première version de l'algorithme en COQ, mais les complication liés à l'utilisation de *Function* nous ont conduit à utiliser une version de l'algorithme plus simple à utiliser en fonctionnel. On peut en effet constater que l'algorithme fonctionnalisé d'OCAMLGRAPH n'est absolument pas conçu pour une approche de ce type, en particulier on notera qu'il passe l'intégralité du contexte d'execution aux fonctions appelées pour simuler des effets de bords.

Si ce manque de clarté de l'algorithme est sans grande importance dans un langage de programmation classique, il complique grandement la situation en COQ. En effet, faire une preuve sur une fonction que ce soit sur sa terminaison ou sa specification nécessite de raisonner sur le corps de la fonction. Il a donc été jugé qu'il est préférable dans le cas de l'algorithme de recherche des composantes connexes d'utiliser un algorithme purement fonctionnel présenté ci-dessous, et se basant sur la description générale de l'algorithme faite en section 2.1.2.2.

L'algorithme utilise la signature suivante pour le graphe :

```
Module Type GRAPH.
  Parameter t  : Set.
  Declare Module V : COMPARABLE.
  Declare Module E : EDGE.
  Parameter succ : t ->  V.t -> list V.t.
  Parameter get_vertices : t -> VSet.t.
End GRAPH.
```

COMPARABLE est une signature de sommet et *EDGE* celle d'une arête, comme vu en section 3.1.9. `succ` est une fonction qui renvoie la liste des successeurs d'un sommet. Enfin `get_vertices` renvoie l'ensemble des noeuds du graphe.

```
Definition path_exists_F :
   forall (next_blue :(list V.t)*VSet.t),
   (forall y, om y next_blue -> V.t -> bool) -> V.t -> bool.
refine
(fun next_blue path_exists_rec d =>
   match fst next_blue with
   | nil  => false
   | h : :t =>
      match V.compare h d with | Eq => true | _ =>
      if VSet.mem_dec  h (snd next_blue) then false else
```

```
      let (lsuc,_) := (G.succ g h,1) in
      let (res1,res2) := (
         path_exists_rec (lsuc,(VSet.add h (snd next_blue))) _ d,
         path_exists_rec (t,snd next_blue) _ d
      )  in
      andb res1 res2
      end
   end).
(* preuve de terminaison *)
[...]
Defined.

Definition path_exists_aux : forall (vq :list V.t*VSet.t),V.t->bool :=
   Fix wf_om (fun vq => (V.t->bool)) (path_exists_F).

Definition path_exists (v1 v2 :V.t) : bool :=
   path_exists_aux (v1 : :nil,VSet.singleton v1) v2.

Fixpoint components_loop2 (v1 :V.t) (l :list V.t) : VSet.t :=
   match l with
   | nil => VSet.empty
   | h : :t =>
      if andb (path_exists v1 h) (path_exists h v1) then
         VSet.add h (components_loop2 v1 t)
      else
         components_loop2 v1 t
   end.

Fixpoint components_loop1 (l :list V.t) (visited :VSet.t) {struct l} : list VSet.t :=
   match l with
   | nil => nil
   | h : :t =>
      if VSet.mem_dec h visited then
        components_loop1 t visited
      else
         let visited := VSet.add h visited in
         let currc := components_loop2 h ((G.get_vertices g).(VSet.support)) in
         currc : :components_loop1 t visited
   end.

Definition scc_list : list VSet.t :=
   components_loop1 ((G.get_vertices g).(VSet.support)) VSet.empty.
```

La fonction `scc_list` est la fonction principale. Les deux fonctions auxilliaires `components_loop1` et `components_loop2` sont utilisées pour faire une boucle imbriquée et ne présentent aucune difficulté. La fonction `path_exists` est uniquement utilisée pour initialiser les arguments de `path_exists_aux` correctement. En revanche, les fonction `path_exists_aux` et `path_exists_F` sont définies d'une manière peu conventionnelle. Cette façon de définir des fonctions utilisant une décroissance des arguments non triviale avec la commande *Fix* et la tactique `refine` est abordée dans la section suivante.

La notation *Fix*

Comme vu précedement, il est impossible de définir une fonction en Coq si sa terminaison n'est pas garantie. Nous avons déjà vu comment utiliser *Function* avec une mesure et nous allons maintenant utiliser les fonctionnalités sous-jacentes.

Tout comme l'algorithme de Dijkstra, celui des composantes connexes a besoin d'une mesure qui sur laquelle les arguments décroissent entre chaque appel. L'ordre lexicographique (donc bien-fondé) est défini sur des paires de type $(cardinal(diff(sommets(g), bluevertices)), length(next))$.

La commande *Fix* permet de replier une fonction récursive sur elle même, si preuve est faite qu'elle termine. En pratique, on va définir une fonction qui au lieu de s'appeler récursivement, va appeler une fonction passée en argument. Regardons de plus près comment celà peut être réalisé avec le cas de la fonction `path_exists_F`.

```
Definition path_exists_F :
   forall (next_blue :(list V.t)*VSet.t),
   (forall y, om y next_blue -> V.t -> bool) -> V.t -> bool.
```

On peut déjà constater que le type correspond à ce que nous venons d'énoncer. Une fonction va être passée en argument, et il devra être garanti que `(om y next_blue) = True`. Concrètement, pour chaque appel à celle fonction, il faudra prouver que les arguments ont diminué sur l'ordre `om`. La tactique `refine` est une tactique un peu spéciale qui tente de compléter ce qui n'est pas défini. Ici `refine` éssaie de faire correspondre le type ci-dessus avec la fonction anonyme définie plus bas.

```
refine
(fun next_blue path_exists_rec d =>
   match fst next_blue with
[...]
```

L'argument `next_blue` de cette fonction anonyme correspond à l'argument sur lequel l'ordre lexicographique est défini. Le deuxième argument est la fonction dont nous avons parlé plus haut, et le dernier argument est un sommet. Notons que les seuls éléments manquant sont la preuve pour chaque appel à `path_exits_rec` que l'argument à diminué. Grâce à la tactique `refine`, c'est ce que COQ va nous demander à la fin de la définition de la fonction en se mettant en mode preuve. Le terme à prouver est alors le suivant :

```
2 subgoals
g : G.t
next_blue : list V.t * VSet.t
path_exists_rec : forall y : list V.t * VSet.t,
                  om y next_blue -> V.t -> bool
d : V.t
h : V.t
t : list V.t
_ : h <> d
_0 : ~ VSet.mem h (snd next_blue)
lsuc : list V.t
_1 : nat
______________________________________(1/2)
om (fst next_blue, VSet.add h (snd next_blue)) next_blue
______________________________________(2/2)
om (lsuc, snd next_blue) next_blue
```

Ces goals correspondent comme prévu à une diminution de l'argument de récursion pour les appels : `(path_exists_rec (fst next_blue,(VSet.add h (snd next_blue))) _ d)` et `path_exists_rec (lsuc,snd next_blue) _ d`.

Une fois la preuve réalisée et la fonction `path_exists_F` bien définie, nous pouvons la replier sur elle même pour obtenir une récursion. il s'agit en fait pour la fonction de se passer elle même en argument et grâce à la preuve faite plus haut de garantir la terminaison.

La commande *Fix* va se charger de faire cette opération pour nous. Il suffit de lui passer en argument :

1. La preuve que l'ordre que nous avons défini est bien fondé (`wf_om`).

2. Une fonction qui renvoie le type de retour en fonction de l'argument (`fun _ => (V.t->bool)`).
3. La fonction qui doit être repliée sur elle même. et que nous avons déjà définie (`path_exists_F`).

On obtient donc la fonction récursive voulue avec la définition :

```
Definition path_exists_aux : forall (vq :list V.t*VSet.t),V.t->bool :=
  Fix wf_om (fun _ => (V.t->bool)) (path_exists_F).
```

Une fois la fonction récursive définie avec *Fix*, elle peut être utilisée comme toute autre fonction.

Chapitre 4

Propriétés de correction

4.1 Preuves de programmes en Coq

4.1.1 Spécification d'une fonction

L'intérêt de programmer en Coq est de l'utiliser pour démontrer qu'un programme est correct. Nous reprenons l'exemple de la fonction *max* qui renvoie le maximum de deux entiers dont nous avions donné une spécification. Voici le programme tel qu'il était donné dans le chapitre précédent :

```
Definition max : natop := fun( m n :nat) =>
     if lt_nat_dec m n then
        n
     else
        m.0

Definition max_spec : Prop :=
   forall a b :nat, (a <= b -> max a b = b) /\ (a >= b -> max a b = a).
```

La spécification du programme *max* est définie par *max_spec*. Il s'agit d'une propriété qui devrait toujours être vraie, quelque soient les arguments de la fonction *max*. Ici cette spécification est triviale : $\forall a, b \in \mathbb{N},\ (a \leq b\ \Rightarrow max(a, b) = b)\ \wedge\ (a \geq b\ \Rightarrow max(a, b) = a)$.

De manière générale, la spécification d'une fonction $f(a_1 : t_1, \ldots, a_n : t_n)$ s'énonce : $\forall a_1 \in t_1, \ldots, \forall a_n \in t_n$, $precondition(a_1, \ldots, a_n) \Rightarrow postcondition(f(a_1, \ldots, a_n))$. Dans ce cas particulier, la précondition est toujours vraie, la spécification se réduit donc à la postcondition.

4.1.2 Preuve

Nous avons défini une fonction *max*, sa spécification *max_spec* et nous avons maintenant les outils nécessaires pour prouver que la première se conforme à la seconde en toutes circonstances. Nous expliquons maintenant comment se passe la preuve de programme.

La première chose à faire est de créer un lemme, qui reprenne notre spécification. Coq se met alors en mode preuve, affiche les hypothèses et le goal courant et nous pouvons commencer la preuve que la spécification est toujours vérifiée.

```
Lemma max_ok : max_spec.
Proof.
```

Le terme initial est :

```
1 subgoal
______________________________________(1/1)
max_spec
```

Le goal principal est notre spécification, et nous allons éxaminer sa définition pour pouvoir continuer.

```
  (* main goal *)
  unfold max_spec.

1 subgoal
______________________________________(1/1)
forall a b : nat, (a <= b -> max a b = b) /\ (a >= b -> max a b = a)
```

On retrouve la définition de notre spécification. la tactique `intros` permet d'introduire autant d'hypothèses que nécéssaire pour ne garder dans notre goal que la partie droite de l'implication.

```
  intros.

1 subgoal
a : nat
b : nat
______________________________________(1/1)
(a <= b -> max a b = b) /\ (a >= b -> max a b = a)
```

Prouver $a \wedge b$ revient à prouver a et b séparément. C'est ce que nous allons faire en utilisant la tactique `split` qui génère deux goals, l'un correspondant à la partie gauche de la conjonction, l'autre à la partie droite.

```
  split.

2 subgoals
a : nat
b : nat
______________________________________(1/2)
a <= b -> max a b = b
______________________________________(2/2)
a >= b -> max a b = a
```

Comme précédement, nous allons poser la partie gauche de l'implication en hypothèse en utilisant `intros` et en nommant notre hypothèse `H`. Nous pouvons même anticiper et remarquer qu'il faudra remplacer *max* par sa définition dans la suite en utilisant `unfold`. Nous omettons le deuxième goal dans la suite pour plus de clarté.

```
  (* subgoal 1 *)
     intro H.unfold max.

2 subgoals
a : nat
b : nat
H : a <= b
______________________________________(1/2)
(if lt_nat_dec a b then b else a) = b
```

Ici, la démonstration devient plus complexe, et il faut raisonner par cas. Ou bien `a` est supérieur à `b`, ou bien c'est `b` qui est supérieur à `a`. On va donc utiliser la tactique destruct qui décompose une construction. Comme nous savons déjà quels sont les différents cas à traiter, nous allons également les nommer, bien que ce ne soit aucunement nécessaire à la démonstration.

```
      destruct (lt_nat_dec a b) as [a_lt_b|a_ge_b].

3 subgoals
a : nat
b : nat
H : a <= b
a_lt_b : a < b
______________________________________(1/3)
b = b
```

Comme prévu, notre goal s'est décomposé en deux sous-goals. Pour le premier : `b = b`, il s'agit du premier cas que nous avions identifié en hypothèse : `a_lt_b : a < b`. Une fois que Coq connait la valeur `(lt_nat_dec a b)`, dans un cas `True`, dans l'autre `False`, celle-ci est directement utilisée pour simplifier les termes. Le goal est résolu trivialement par la tactique homonyme `trivial`.

```
      (* subgoal 1.1 : a_lt_b *)
         trivial.

2 subgoals
a : nat
b : nat
H : a <= b
a_ge_b : b <= a
______________________________________(1/2)
a = b
```

Le deuxième cas est beaucoup plus intéressant, et il est nécéssaire de parcourir la librarie standard pour trouver un lemme qui s'applique à notre cas. En effet, il est clair que si $a \leq b$ et $b \leq a$ alors $a = b$. Cette propriété correspond à un théorème qui se trouve dans la section `Le` de la librarie et s'énonce : `Theorem le_antisym : forall n m, n <= m -> m <= n -> n = m`. Il est important de remarquer que

```
      (* subgoal 1.2 : a_ge_b *)
         apply le_antisym.

3 subgoals
a : nat
b : nat
H : a <= b
a_ge_b : b <= a
______________________________________(1/3)
a <= b
______________________________________(2/3)
b <= a
```

Coq applique immédiatement le théorème pour résoudre notre goal, mais deux nouveau goals sont générés qui nous obligent à prouver que le théorème s'applique bien ici. Ceci n'est pas un problème, `a <= b` et `b <= a` sont en hypothèse, la tactique `assumption` est utilisée pour faire remarquer à Coq que le résultat est une hypothèse, et qu'il est donc correct.

```
         assumption. assumption.

  1 subgoal
a : nat
b : nat
______________________________________(1/1)
a >= b -> max a b = a
```

Il ne nous reste qu'à prouver le second goal d'une manière similaire.

```
    (* subgoal 2 *)
       intro H.
       unfold max.
       destruct (lt_nat_dec a b) as [a_lt_b|a_ge_b].
       (* subgoal 2.1 : a_lt_b *)
          apply lt_le_weak in a_lt_b.
          apply le_antisym ; assumption.
       (* subgoal 2.2 : a_gt_b *)
          trivial.

Proof completed.
```

La seconde partie de cette preuve est très proche de la première. Il faudra tout de même retourner dans la documentation de la librarie standard pour utiliser `lt_le_weak`. la preuve est maintenant terminée il suffit de la sauver pour continuer.

```
Qed.
```

Le lemme `max_ok` est maitenant défini, et l'intérêt est qu'il pourra servir dans tous les cas où l'on utilisera la fonction. Il suffira d'appliquer ce lemme pour immédiatement utiliser les propriétés de cette fonction dans une preuve, et cela sans avoir aucune connaissance de la fonction `max`.

4.2 Spécification du plus court chemin de Dijkstra

La fonction idéale est celle qui dispose de la meilleure complexité connue pour le problème (c'est le cas pour l'algorithme de Dijkstra), dont on sait qu'elle se termine et dont la correction est prouvée, c'est-à-dire qu'on est sûr qu'elle fait ce qu'elle doit faire. Nous avons déterminé que l'algorithme du plus court chemin se termine bien : il faut donc désormais prouver sa correction. Pour ce faire, il faut spécifier à Coq ce qu'on attend de la fonction.

L'idée générale est de trouver une spécification bien entendu correcte (ce qui n'est pas évident) mais pratique et surtout suffisante pour pouvoir prouver la fonction *shortest_path*. La fonction *shortest_path* retournant le chemin sous forme de *option* (*list E.t*) (rien ou une liste d'arcs), nous définissons le type *path* de la même façon :

```
Definition path : Set := option (list E.t).
```

Il nous faut donc trouver un moyen de dire maintenant : « *la fonction shortest_path renvoie bien un chemin valide du graphe, et c'est bien le (ou un des) plus court s'il existe, aucun s'il n'y a aucun chemin* ». Nous définissons ainsi deux fonctions *path_source* (resp. *path_source*) qui renvoient la source (resp. la destination) d'un chemin :

```
Definition path_source (p :path) : option V.t :=   match p with
    | None => None
    | Some(nil) => None
    | Some(elt : :_) => Some(fst elt)
    end
.

Definition path_target (p : path) : option V.t :=
    let sub_path_target := fix sub_path_target  (p :list E.t) : option V.t :=
    match p with
    | nil => None
    | elt : :nil =>  Some(fst elt)
```

```
    | _ : :tl => sub_path_target tl
    end
    in
    match p with
    | None => None
    | Some(wp) => sub_path_target wp
    end
.
```

Puis nous définissons ce qu'est un chemin bien défini dans un graphe $G(V, E)$ et un chemin p de la façon suivante : $\forall e_1, e_2 \in p : e_1, e_2 \in E \wedge ((e_1 :: e_2) \in p \rightarrow dst\, e_1 = src\, e_2)$ c'est-à-dire que tous les arcs d'un chemin sont bien dans le graphe, et si deux arcs e_1 et e_2 se suivent dans le chemin, alors la destination de e_1 est égale à la source de e_2. Voici le code en COQ :

```
Definition path_well_defined (p : path) : Prop :=
    let sub_path_well_defined :=
    fix sub_path_well_defined (p : list E.t) (last : option V.t)
                                        {struct p} : Prop := match p with
    | nil => True
    | hd : :tl =>
    match last with
        |None => (In hd (G.get_edges g)) /\ sub_path_well_defined tl (Some(snd hd))
        | Some(lastv) => if (lastv == (fst hd))
                        then
                            (In hd (G.get_edges g)) /\
                            sub_path_well_defined tl (Some(snd hd))
                        else False
                    end
    end
   in
    match p with
    | None => True
    | Some(p) => sub_path_well_defined p None
    end
.
```

Nous avons donc pouvons donc maintenant implémenter la fonction qui vérifie si un chemin est bien un chemin de s à t correctement défini (ou bien le chemin non-existant s'il n'existe aucun chemin de s à t). Cette fonction réutilise donc les trois fonctions définies ci-dessus. Voici le code en COQ de la fonction is_a_path qui prend un chemin p et deux sommets s et t :

```
Definition is_a_path (p : path) (s : V.t) (t : V.t) : Prop :=
   path_well_defined p /\
   (
       (p = None) \/
       (
          (~(p = None)) /\
          (path_source p = Some(s)) /\
         (path_target p = Some(t))
       )
   )
.
```

Nous disposons de la première partie de notre spécification puisqu'on sait dire si le chemin renvoyé est bien valide et dans le graphe. Nous devons maintenant vérifier qu'il est bien le (ou un des) plus court(s). Nous avons donc besoin d'une fonction qui donne le coût des chemins définie telle que :

```
Definition path_cost (p : path) : W.t :=
    let sub_path_cost := fix sub_path_cost (p : list E.t) : W.t :=
    match p with
    | nil => 0
    | hd : :tl => (W.weight (label hd)) +  sub_path_cost (tl)
    end
    in
    match p with
    | None => 0
    | Some(p) => sub_path_cost p
    end
.
```

Finalement, nous pouvons définir si un chemin p est le plus court : $is_a_path(p, s, t) \wedge (\forall p' : is_a_path(p', s, t) \rightarrow (p' = None \wedge p = None) \vee (pa \neq None \wedge path_cost(p) \leq path_cost(p'))$, c'est-à-dire soit le chemin p est videe, dans ce cas il n'existe pas de chemin du tout de s à t , soit p est un chemin dont le poids est inférieur ou égal au poids de tous les autres chemins de s à t.

```
Definition is_shortest_path (p : path) (s : V.t) (t : V.t) :=
is_a_path p s t /\ (
forall pa :path, is_a_path pa s t -> (pa = None /\ p = None) \/
               ((path_cost p <= path_cost pa) /\ ~(pa = None))
).
```

Nous pouvons maintenant bien spécifier la fonction *shortest _path* qui renvoie une optionnellement une paire $(path, cost)$: il faudra prouver le terme généré pour prouver sa correction. Voici le code de la spécification en Coq :

```
Lemma shortest_path_correction (s : V.t ) (t : V.t ) :
    forall s t : V.t, VSet.mem s (G.get_vertices g) /\
                      VSet.mem t (G.get_vertices g) ->
    match shortest_path g s t with
    | None => is_shortest_path None s t
    | Some(p, w) => is_shortest_path (Some(p)) s t /\ (path_cost p) = w.
```

On précise ici donc des pré-conditions (s et t sont bien dans le graphe) et les post-conditions (le chemin trouvé est bien le plus court). La correction de l'algorithme n'a pas été prouvée faute de temps mais elle a été commencée : les grands axes de cette preuve sont qu'il faut avant tout savoir prouver qu'un chemin de s à t existe (pour le cas où l'algorithme ne trouve aucun chemin), et dans le cas contraire prouver qu'on trouve forcément un chemin le plus court possible.

Conclusion

Nous avons, au cours de ce stage comme au long de ce rapport, suivi un processus de génie logiciel : de conception d'algorithmes génériques par les foncteurs et étude d'Ocamlgraph aux spécifications de la correction (et peut-être bientôt la preuve ?) en passant par la preuve de terminaison. Ce processus est bien celui qui nous offre le plus de garanties :

- choix d'algorithmes efficaces (études de complexité) : garantie de performance
- généricité des algorithmes : garantie réutilisabilité et pérennité du code
- typage dur statique, utilisation de langages avec un modèle mémoire sûr (pas de pointeurs, etc) : garantie contre une majorité de plantages et failles de sécurité (segfaults, buffer overflows, ...)
- preuve de terminaison : garantie que le programme s'arrête (contre les plantages type « boucle infinie »)
- preuve des spécifications : garantie que le programme fait bien ce qu'il doit faire, quelle que soit l'entrée

On obtient ainsi un programme dont la correction est mathématiquement prouvée, mais qui est en plus performant, réutilisable et pérenne. Bien qu'il offre toutes ces garanties, ce processus est encore difficile à aborder : la preuve de programmes assistée par ordinateur reste peu utilisée dans l'industrie, au profit de méthodes moins sûres comme les tests unitaires et la programmation par contrat qui n'offrent aucune garanties de correction (au mieux, des probabilités de garantie). Les raisons sont diverses : rapport « importance de la correction »/coût peu intéressant pour une majorité de logiciels commerciaux, manque de formation et de documentation sur les assistants de preuve, etc.

On peut néanmoins espérer que les preuves de programmes se généraliseront dans les années à venir : la course au produit le moins cher va être court-circuitée par le logiciel libre et celle des fonctionnalités va forcément s'arrêter au fur et à mesure que le temps passe et que moins d'idées sont à ajouter. Peut-être restera-t'il la course à la qualité du logiciel, et c'est là que peut-être la correction des programmes interviendra.

Encore mieux, peut-être un jour parviendra-t'on à obtenir les mêmes performance, réutilisabilité et pérennité en extractant le code à partir de la preuve.

En attendant, espérons que la fonction de pilote automatique des avions sur lesquels on embarque, et même bientôt de nos voitures, a été prouvée !

Conclusions personnelles

Simon Chemouil

Ce stage a été l'occasion pour moi de découvrir des domaines, outils et théories, mais aussi d'approfondir ma compréhension générale de l'informatique. En passant du travail de compréhension sur Ocamlgraph à l'étude de son design, aux rappels de graphes à la logique, j'ai beaucoup apprécié la diversité des sujets abordés, tout en restant dans une ligne directrice : la preuve assistée. Coq est un langage qui demande énormément de prérequis pour être utilisé, et sur lequel on est très souvent bloqué au début : il faut de la persévérance pour avancer. Cependant, une fois qu'on a compris son design, ses principes fondateurs et sa syntaxe, on ne peut qu'apprécier la puissance et la rigueur du système. J'ai le sentiment, grâce à ce stage, d'avoir donc pu faire le premier pas, d'avoir franchi la barrière d'entrée collossale de Coq. D'un point de vue éducatif donc, j'ai trouvé ce stage profondément enrichissant, et très intéressant.

D'un point de vue plus personnel, j'anticipais ce stage pour découvrir un peu le milieu de la recherche en informatique, puisque j'hésite toujours dans mon orientation professionnelle. J'ai apprécié la qualité des discussions et une certaine convivialité au sein de l'équipe Démons. Par ailleur, les sujets qui y sont traités m'intéressent beaucoup, et j'ai eu l'occasion de poser des questions ne touchant pas directement au sujet du stage.

En conclusion, je suis très satisfait du déroulement de ce stage : principalement pour avoir découvert, appris et compris et ensuite pour la gratification d'avoir produit (et de voir les termes « *Proof Completed* » s'afficher dans Coq, même si j'aurais aimé qu'ils s'affichent aussi pour la correction de l'algorithme, mais je ne désespère pas !). Je remercie Sylvain Conchon et Évelyne Contejean de nous avoir donné l'opportunité de réaliser ce stage, mais aussi et surtout pour leur aide et leurs nombreuses réponses !

Ludovic Arnold

Au début de cette année, nous avons contacté M. Conchon pour suivre un stage au sein de son équipe de recherche avec l'espoir de travailler sur un sujet plus intéressant que ceux qui étaient présentés aux étudiants. Je pense aujourd'hui que c'était le bon choix. Le sujet de la preuve assistée s'est révélé très enrichissant, et je suis convaincu que ce travail sur Coq constitue une expérience inestimable autant en mathématiques qu'en informatique. Bien que je doute que nos algorithmes soient réellement utilisés par la communauté, j'espère que ce document pourra donner envie a ceux qui le liront de se lancer dans le monde des assistants de preuve.

Je tiens à remercier toute l'équipe Démons et en particulier Sylvain Conchon et Évelyne Contejean pour leur acceuil chaleureux.