


Générateur de code Coq

Travaux encadrés de recherche

Licence 3 informatique

Laboratoire : Institut de Recherche en Informatique de Toulouse

Maître de stage : Prof. Jean-Paul Bodeveix

Étudiant : Joas Kinouani

Année universitaire 2012/2013

Table des matières

Chapitre 1

Introduction

Ce document expose les activités effectuées dans le cadre des travaux encadrés de recherche (TER). Il s'agit d'une unité d'enseignement du sixième et dernier semestre de la licence informatique de l'université Paul Sabatier Toulouse 3. Son but est d'apprendre aux étudiants à intégrer une équipe de recherche, et à produire un travail conforme aux attentes de celle-ci [1].

Ce TER a eu lieu à l'Institut de recherche en informatique de Toulouse (IRIT). Il a été réalisé en binôme avec M. Abakar Mahamat Sougui. En tant qu'étudiants, nous avons été encadrés par Pr. Jean-Paul Bodeveix, membre permanent de l'équipe d'Assistance à la certification d'applications distribuées et embarquées (ACADIE) de l'IRIT.

L'IRIT est une unité mixte de recherche où travaillent du personnel du centre National de la Recherche scientifique (CNRS), de l'Institut National Polytechnique de Toulouse (INPT), de l'université Toulouse 1 Capitole, de l'université Toulouse 2 Le Mirail, et de l'université Paul Sabatier Toulouse 3 [2]. Dirigée depuis juillet 2011 par Michel Daydé, elle compte en son sein 600 personnes dont 250 chercheurs et enseignants-chercheurs, 244 doctorants, 14 post-doctorants et chercheurs contractuels, et 43 ingénieurs et administratifs. Ses 19 équipes de recherches se répartissent sept thèmes d'étude : (1) l'analyse et la synthèse de l'information ; (2) l'indexation et la recherche d'information ; (3) l'interaction, l'autonomie, le dialogue et la coopération ; (4) le raisonnement et la décision ; (5) la modélisation, les algorithmes et le calcul haute performance ; (6) l'architecture, les systèmes et les réseaux ; (7) la sûreté de développement du logiciel. Les équipes fédèrent leurs efforts autour de quatre axes stratégiques correspondant aux défis contemporains : (1) l'informatique pour la santé ; (2) les masses de données et le calcul ; (3) les systèmes sociotechniques ambiants ; (4) les systèmes embarqués critiques [3]. Le laboratoire a été noté A+ par l'Agence d'évaluation de la Recherche et de l'Enseignement **s**upérieur (AÉRES) [4].

L'équipe ACADIE a pour objectif de contribuer à l'amélioration des méthodes et outils de certification des logiciels distribués et embarqués. Son thème de recherche est la sûreté de développement du logiciel. Y travaillent actuellement 17 membres permanents, dont le responsable, M. Mamoun Filali, chargé de recherche au CNRS, et notre encadrant, Pr. Jean-Paul Bodeveix, professeur à l'université Paul Sabatier Toulouse 3. S'y trouvent aussi 23 membres non-permanents, parmi lesquels un étudiant, un contractuel, des doc-

1. *Licence informatique troisième année. Présentation des enseignements 2012-2013*. 2012. URL : http://licinfo.deptinfo.fr/file/Syllabus_L3_Info_2012_2013.pdf (visité le 29/05/2013), p. 16.

2. « Institut de recherche en informatique de Toulouse ». Dans : *Wikipedia* (21 mar. 2013). URL : http://fr.wikipedia.org/wiki/Institut_de_recherche_en_informatique_de_Toulouse (visité le 29/05/2013).

3. Michel Daydé. *Le mot du directeur*. URL : http://www.irit.fr/Le-mot-du-directeur,1155?lang=fr (visité le 29/05/2013).

4. « Institut de recherche en informatique de Toulouse », cf. note 2.

torants, des post-doctorants, et des chercheurs invités [5].

Ce TER porte sur la génération de code Coq. Il s'appuie sur des connaissances fraîchement acquises au cours de l'année, à savoir la théorie des langages et la programmation fonctionnelle avancée. Étant donné, dans un certain langage, un type inductif, on doit générer en Coq ses observateurs, ses accesseurs et la relation de sous-terme avec la preuve de sa transitivité. Nous expliquerons ces concepts dans le chapitre 2. Le chapitre 3 se concentrera sur leur implémentation.

Ce travail n'existerait pas si les enseignants de l'IRIT ne m'avait formé pour. Je tiens à les remercier pour la qualité de leur pédagogie. Ma reconnaissance va particulièrement à Pr. Jean-Paul Bodeveix qui a chaleureusement accueilli notre candidature spontanée. Merci aussi à M. Abakar Mahamat Sougui ; la bonne humeur qui régnait entre Pr. Jean-Paul Bodeveix, M. Abakar Mahamat Sougui et moi, a rendu ce TER vraiment très agréable.

5. *Équipe ACADIE*. URL : `http://www.irit.fr/-Equipe-ACADIE-?lang=fr` (visité le 29/05/2013).

Chapitre 2

Projet et outils

Nous pouvons résumer ce travail comme le développement d'un mini-compilateur, baptisé Tifc, traduisant un programme Tif en un programme Coq [1] (figure 2.1). La compréhension de son code source, écrit en Ocaml, nécessite donc la connaissance des bases de ces deux langages. Ce chapitre est là pour les présenter (§2.a, §2.c). Il parle aussi des des logiciels utilisés (§2.b) et du processus de développement appliqué (§2.d). Le chapitre suivant documente les modules et les fonctions du code source.

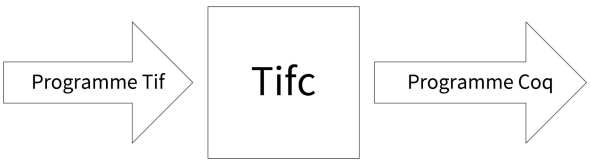

Figure 2.1 — Tifc. Tifc (nom signifiant ***Tif c**ompiler*, c.-à-d. « compilateur Tif ») prend en entrée un programme Tif et délivre en sortie un programme Coq.

2.a Types inductifs en Tif

Le langage Tif (mot-valise signifiant « **t**ype induct**if** ») sert à décrire des types inductifs ou récursifs.

Définition 2.1 — type. Disons qu'un *type* `t` s'interprète comme un ensemble de *valeurs*. Il s'appuie sur un jeu de *constructeurs* donnant, à eux-seuls, le moyen d'atteindre chacune d'entre elles.

Exemple 2.2 — booléens. Un booléen (angl. *boolean*) est soit vrai (angl. *true*) soit faux (angl. *false*). Pour atteindre toutes les valeurs de type `boolean`, nous n'avons donc besoin que de deux constructeurs : `True` et `False`. En Tif, il faut écrire :

```
type boolean = True | False
```

Exemple 2.3 — jours de la semaine. Une semaine est faite de sept jours (angl. *days*). Pour atteindre toutes les valeurs de type `day`, nous avons donc besoin de sept constructeurs :

1. On peut s'amuser à lui ajouter d'autres langages cibles comme Ocaml, Java ou C. Nous avons brièvement discuté avec notre encadrant comment y parvenir (cf. §2.e.

```
type day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

Définition 2.4 — type inductif ou récursif. Un type est *inductif* ou *récursif* lorsqu'il fait référence à lui-même dans sa définition. Il s'agit en général de types dont l'ensemble de valeurs est infini.

Exemple 2.5 — entiers naturels. Un entier naturel (angl. *natural number*) est soit 0, soit le successeur (angl. *successor*) d'un autre entier naturel n). En Tif cela s'écrit :

```
type natural =
     Zero
   | Successor (n : natural)
```

Remarque 2.6 — entiers naturels. La théorie des ensembles recourt au même procédé pour définir l'ensemble $\mathbb{N}$ des entiers naturels. Elle pose $0 \stackrel{\text{def}}{=} \emptyset \in \mathbb{N}$, et, pour tout n, si $n \in \mathbb{N}$ alors $n+1 \stackrel{\text{def}}{=} n \cup \{n\} \in \mathbb{N}$. 0 correspond à `Zero`, et $n+1$ à `Successor`(n). Par exemple : $1 = 0+1 \stackrel{\text{def}}{=} 0 \cup \{0\} = \emptyset \cup \{\emptyset\} = \{\emptyset\}$; $2 = 1+1 \stackrel{\text{def}}{=} 1 \cup \{1\} = \{\emptyset\} \cup \{\{\emptyset\}\} = \{\emptyset, \{\emptyset\}\}$; etc.

Exemple 2.7 — piles. Une pile (angl. *stack*) d'objets de type `t` est soit vide (angl. *empty*), soit le résultat de l'empilement (angl. *push*) d'un objet `top` de type `t`, au sommet d'une autre pile `pop` d'objets de type `t`. Voici la déclaration du type `stack` en Tif :

```
type stack (t : Type) =
     Empty
   | Push (top : t) (pop : pile t)
```

Il s'agit d'un type *générique* c.-à-d. paramétré par un autre type `t`.

Nous donnons ci-dessous la grammaire du langage Tif en ne mentionnant que les règles de production. Les symboles terminaux sont écrits entre guillemets avec une fonte à chasse fixe (les mots clés ayant davantage de graisse), et les symboles non terminaux en italique.

La racine *déclaration* correspond à une déclaration de type. En Tif, elle débute par le mot clé **`type`**. Vient ensuite l'identificateur du type (commençant par une minuscule), ses éventuels paramètres, le symbole `=`, puis ses constructeurs (au moins un) séparés par des '`|`' :

déclaration → '**`type`**' *identificateurMin paramètresOpt* '`=`' *constructeurs*
paramètresOpt → ϵ | *paramètre paramètresOpt*
constructeurs → *constructeur* | *constructeur* '`|`' *constructeurs*

Pour un paramètre, le compilateur s'attend à lire une parenthèse ouvrante, suivie d'un identificateur (commençant par une minuscule), d'un deux-points, d'un typage, puis d'une parenthèse fermante :

paramètre → '`(`' *identificateurMin* '`:`' *typage* '`)`'

Pour un constructeur, il veut un identficateur (commençant par une majuscule) et des arguments éventuels ; ces derniers ayant la même syntaxe que les paramètres :

constructeur → *identificateurMaj argumentsOpt*

argumentsOpt → ϵ | *argument argumentsOpt*
argument → '(' *identificateurMin* ':' *typage* ')'

Un typage consiste soit en un nom de type (éventuellement générique), soit en un produit de types, soit en une signature de fonction ; on peut se servir de parenthèses :

typage → *type* | *parenthésé* | *produit* | *signatureDeFonction*
type → *identificateur généricité*
généricité → ϵ | *identificateur généricité*
parenthésé → '(' *typage* ')'
produit → *typage* '*' *typage* | *typage* '*' *produit*
signatureDeFonction → *typage* '->' *typage*

La signature de fonction est associative à droite. Pour terminer la grammaire, il ne reste que les identificateurs :

identificateur → *identificateurMin* | *identificateurMaj*
identificateurMin → *lettreMin chaîne*
lettreMin → 'a' | ... | 'z'
identificateurMaj → *lettreMaj chaîne*
lettreMaj → 'A' | ... | 'Z'
chaîne → ϵ | *caractère chaîne*
caractère → *lettreMin* | *lettreMaj* | *chiffre* | *autre* | '_' | '''
chiffre → '0' | ... | '9'

Le non terminal *autre* renvoie aux caractères comme `é` ou `à` dont la disponibilité dépend du système d'exploitation. Voici des exemples supplémentaires de déclarations Tif :

Exemple 2.8 — couleurs. Une couleur (angl. *color*) est caractérisée par ses composantes rouge (angl. *red*), verte (angl. *green*) et bleue (angl. *blue*) ; toutes étant des entiers naturels :

```
type color = Color (red : natural) (green : natural) (blue : natural)
```

Exemple 2.9 — arbres binaires. Un arbre (angl. *tree*) d'objets de type `t` est soit une feuille (angl. *leaf*), soit un nœud (angl. *node*) ⟨`left`, `root`, `right`⟩ où `left` est le sous-arbre gauche, `root` est un objet, et `right` le sous-arbre droit :

```
type tree (t : Type) =
   Leaf
 | Node (left : tree t) (root : t) (right : tree t)
```

Exemple 2.10 — graphes. Un graphe (angl. *graph*) d'objets de type `t` est la donnée de sommets (angl. *vertices*) et d'arcs (angl. *edges*) les reliant. Un graphe est soit vide (angl. *empty*), soit le résultat de l'ajout d'un sommet (angl. *vertex*) ou d'un arc entre deux sommets. En Tif, le type `graph` se déclare donc comme suit :

```
type graph (t : Type) =
     Empty
   | AddVertex (vertex : t) (subgraph : graph t)
   | AddEdge (start : t) (end : t) (subgraph : graph t)
```

2.b

Analyse syntaxique avec Camlp5

Camlp5 est un préprocesseur pour Ocaml [2]. On peut, dans du code Ocaml, lui réserver des instructions ; il les exécutera juste avant le compilateur Ocamlc. Il propose, entre autres, des fonctions d'analyseur syntaxique que nous avons utilisé pour le langage Tif (figure 2.2).

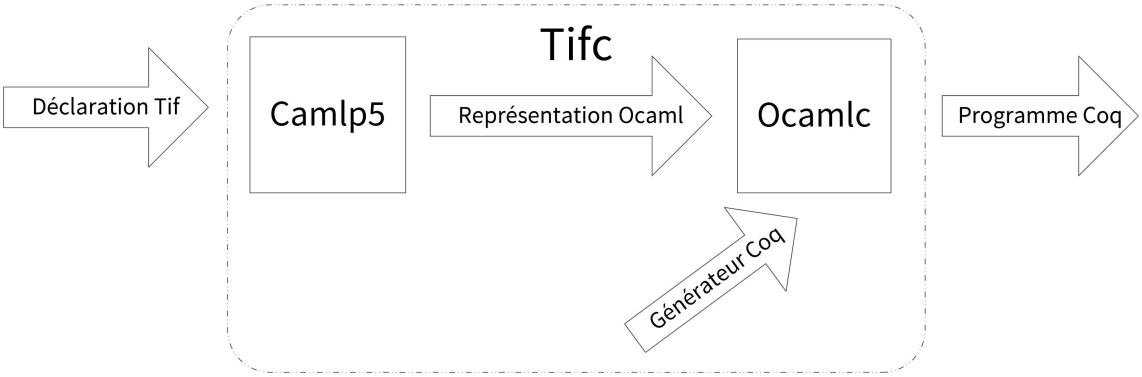

Figure 2.2 — Fonctionnement de Tifc. Tifc fait appel à Camlp5 pour vérifier la syntaxe du programme Tif en entrée. Camlp5 génère alors une représentation Ocaml du type inductif spécifié par ce dernier. Tifc intègre aussi un générateur de code Coq qui, à partir de la représentation Ocaml du type inductif, génère un programme Coq. Le tout est passé à Ocamlc pour compilation.

Camlp5 utilise la structure de donnée suivante pour stocker en mémoire le type inductif spécifié en Tif (cf. §3.a) :

```
type expression_t =
     Type of string * string list
   | Parenthesized of expression_t
   | Product of expression_t list
   | FunctionSignature of expression_t * expression_t
type argument_t = Argument of string * expression_t
type constructor_t = Constructor of string * argument_t list
type parameter_t = Parameter of string * expression_t
type declaration_t = Declaration of string * parameter_t list * constructor_t list
```

La similitude avec la grammaire de Tif est frappante.

Exemple 2.11 — arbres binaires. À la lecture du type `tree` de l'exemple 2.9, Camlp5 génère en mémoire :

```
Declaration ("tree",
   [Parameter ("t", Type ("Type", []))],
   [Constructor ("Leaf", []);
    Constructor ("Node",
```

2. Daniel de Rauglaudre. *Camlp5 - Reference Manual. Version 6.00*. Institut National de Recherche en Informatique et Automatique, 2010. URL : `http://pauillac.inria.fr/~ddr/camlp5/doc/pdf/camlp5-6.00.pdf` (visité le 31/05/2013).

```
[Argument ("left", Type ("tree", ["t"]));
 Argument ("root", Type ("t", []));
 Argument ("right", Type ("tree", ["t"]))])])
```

2.c L'assistant de preuve Coq

L'assistant de preuve Coq permet d'écrire des types inductifs, ainsi que des fonctions et des preuves sur ceux-ci. Tif s'inspire beaucoup de la syntaxe de Coq. En effet, pour passer de Tif à Coq, il suffit de remplacer `type` par **`Inductive`**, `=` par `:=`, et terminer la déclaration par un point.

Exemple 2.12 — abres binaires. En Coq, le type `tree` de l'example 2.9 s'écrit :

```
Inductive tree (t : Type) :=
    Leaf
  | Node (left : tree t) (root : t) (right : tree t).
```

Pour être précis, le code ci-dessus est écrit en Gallina, le langage de programmation fonctionnelle sur lequel s'appuie Coq. Parmi ses types prédéfinis se trouvent `bool` (similaire au type `boolean` de l'exemple 2.2, mais avec `True` et `False` respectivement renommés `true` et `false`) ; `nat` (similaire au type `natural` de l'exemple 2.6, mais avec `Zero` et `Successor` respectivement renommés `0` et `S`) ; `Prop`, le type des propositions logiques ; `Set`, le type des types de données de base (comme `bool` et `nat`) ; et `Type`, le type de tous les types.

La syntaxe de Gallina s'explique par sa philosophie.

Tout d'abord, pour lui, les fonctions et les types sont, au même titre que les données de base (les types de type `Set` comme `bool` et `nat`) des *valeurs*, c.-à-d. des informations pouvant servir d'arguments à d'autres fonctions. Les langages fonctionnels comme Ocaml ne vont pas aussi loin, ils se limitent aux fonctions. L'interpréteur interactif pour Coq, Coqtop, fournit la commande **`Check`** qui renseigne sur le type d'une valeur connue par le système. Voyez plutôt (`Coq <` est l'invité de commande ; les réponses de Coqtop sont en italique grisé ; et la notation `v : t` signifie que la valeur `v` est de type `t`) :

```
Coq < Check true.
true : bool
Coq < Check false.
false : bool
Coq < Check S (S (S 0)).
3 : nat
Coq < Check bool.
bool : Set
Coq < Check nat.
nat : Set
Coq < Check Prop.
Prop : Type
Coq < Check Set.
Set : Type
```

```
Coq < Check Type.
Type : Type
```

Pour Gallina, `tree` est une fonction de `Type` dans `Type`. Il considère que si `t` est un type, `tree t` l'est aussi. Ainsi de `tree bool`, `tree nat` et de `tree Type` qui sont respectivement les types des arbre de booléens, d'entiers naturels et de types :

```
Coq < Check tree.
tree : Type -> Type
Coq < Check tree bool.
tree bool : Set.
Coq < Check tree nat.
tree nat : Set
Coq < Check tree Type.
tree Type : Type
```

Deuxièmement, lorsqu'ils n'ont pas d'arguments, les constructeurs du type inductif sont eux-même vus commes des fonctions :

```
Coq < Check 0.
0 : nat
Coq < Check S.
S : nat -> nat
```

De plus, les paramètres du type inductif sont, pour ses constructeurs, des arguments quantifiés :

```
Coq < Check Leaf.
Leaf : forall t : Type, tree t
Coq < Check Leaf nat.
Leaf nat : tree nat
Coq < Check Leaf Type.
Leaf Type : tree Type.
Coq < Check Node.
Node: forall t : Type, tree t -> t -> tree t -> tree t
Coq < Check Node nat (Leaf nat) 397 (Leaf nat).
Node nat (Leaf nat) 397 (Leaf nat) : tree nat
Coq < Check Node Set (Leaf Set) bool (Leaf Set).
Check Node Set (Leaf Set) bool (Leaf Set) : tree Set
```

Les mots clés **`Definition`** et **`Fixpoint`** de Gallina jouent respectivement le même rôle que les `let` et `let rec` d'Ocaml ; ils servent à définir une valeur respectivement non récursive et récursive. On retrouve aussi le **`match ... with ... end`**.

Exemple 2.13 — Factorielle. En Gallina, la factorielle se définit ainsi :

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

La commande **`Compute`** demande à Coqtop d'effectuer un calcul :

```
Coq < Compute factorial 3.
 = 6 : nat
```

En vérité, le **`Fixpoint`** de Gallina n'est pas exactement le **`let rec`** d'Ocaml. **`Fixpoint`** rend obligatoire le respect du schéma d'induction associé à un type inductif. Ce schéma d'induction exige : (1) que la fonction prévoit un traitement pour chaque constructeur (Ocaml se contente d'avertir de la présence d'aiguillages incomplets) ; (2) que ses appels récursifs se fassent sur des données « plus petites » ou « moins complexes » que l'argument en cours de traitement.

Exemple 2.14. Comparez les réponses de Coq et d'Ocaml.

```
Ocaml # let rec redo n = redo n;;
val redo : 'a -> 'b = <fun>
Coq < Fixpoint redo (n : nat) := redo n.
Erreur : impossible d'inférer redo
Ocaml # let rec zeroOnly n = match n with 0 -> n;;
Attention : l'aiguillage n'est pas exhaustif
Coq < Fixpoint zeroOnly (n : nat) := match n with 0 => n end.
Erreur : l'aiguillage n'est pas exhautif
Ocaml # let rec neverDecrease n =
   match n with
     0 -> 0
   | n -> neverDecrease n;;
val neverDecrease : int -> int = <fun>
Coq < Fixpoint neverDecrease (n : nat) :=
   match n with
     0 => 0
   | S k => neverDecrease (S k) end.
Erreur : définition récursive de neverDecrease mal formée
```

Coq veut interdire la non-terminaison des programmes.

Exemple 2.15 — Observateurs d'un type inductif. Soit `t` un type inductif et `v` une valeur de ce type. Un *observateur* est une fonction qui permet de savoir lequel des constructeurs de `t` a été utilisé pour obtenir `v`. Il y a autant d'observateurs que de constructeurs. Le type `tree` a par exemple deux constructeurs : `isLeaf` et `isNode`.

```
Definition isLeaf {t : Type} (x : tree t) : Prop :=
   match x with
     Leaf => True
   | _ => False
   end.
Definition isNode {t : Type} (x : tree t) : Prop :=
   match x with
     Node _ _ _ => True
   | _ => False
   end.
Coq < Compute isLeaf (Leaf bool).
 = True : Prop
Coq < Compute isLeaf (Node bool (Leaf bool) true (Leaf bool)).
 = False : Prop
Coq < Compute isNode (Leaf nat).
 = False : Prop
```

```
Coq < Compute isNode (Node nat (Leaf nat) 397 (Leaf nat)).
 = True : Prop
```

`True` et `False` correspondent respectivement aux propositions toujours vraie et toujours fausse de la logique. L'argument `t : Type` est optionnel (Gallina peut l'inférer au moment de l'appel à partir de l'argument obligatoire `x : tree t`), c'est pourquoi on le met entre accolades.

Exemple 2.16 — Accesseurs d'un type inductif. Soit `t` un type inductif, `v` une valeur de ce type, et `C` le constructeur de `t` utilisé pour obtenir `v`. Un *accesseur* est une fonction permettant d'accéder à l'un des arguments donné à `C` lors de la construction de `v`. Un constructeur donné a autant d'accesseurs que d'arguments. Le constructeur `Leaf` du type `tree` n'a aucun accesseur. Par contre, son constructeur `Node` en a trois : `getLeftSubtree`, `getRoot` et `getRightSubtree`.

```
Definition getLeftSubtree {t : Type} (x : tree t) : isNode x -> tree t :=
  match x with
    Node leftSubtree _ _ => fun precondition => leftSubtree
  | _ => fun precondition => match precondition with end
  end.
Definition getRoot {t : Type} (x : tree t) : isNode x -> t :=
  match x with
    Node _ root _ => fun precondition => root
  | _ => fun precondition => match precondition with end
  end.
Definition getRightSubtree {t : Type} (x : tree t) : isNode x -> tree t :=
  match x with
    Node _ _ rightSubtree => fun precondition => rightSubtree
  | _ => fun precondition => match precondition with end
  end.
Coq < Definition node := Node nat (Leaf nat) 397 (Leaf nat).
node is defined
Coq < Check I.
I : True
Coq < Compute getLeftSubtree node I.
 = Leaf nat : tree nat
```

Toutes ces fonctions ont deux arguments : (1) la variable `x` de type `tree t` ; (2) la précondition `isNode x`, c.-à-d. la preuve que l'entrée `x` est un nœud (et non un arbre vide). Là encore, Coq se démarque des langages traditionnels. En Coq, la précondition d'une fonction fait partie de ses entrées ! Cela signifie que l'appel de la fonction n'est possible que si on fournit, en plus des arguments normaux, la preuve du respect de la précondition ! C'est ce que nous avons fait dans la dernière instruction de la session intéractive ci-dessus. Comme la valeur `node` est un `Node`, nous devons passer à `getLeftSubtree` la preuve de `True` ; laquelle, par définition, est `I`.

Exemple 2.17 — Types avec dépendances. Lors du TER, nous avons été confronté au problème des types ayant dans leur définition des dépendances. Prenons le type suivant :

```
Inductive dependancy (s : Type) (t : Type) :=
   Without (first : s) (second : t)
 | With (first : Type) (second : first).
```

Son constructeur `With` prend un type comme premier argument, puis l'utilise pour typer le second. L'accesseur ci-dessous, donnant l'argument `second` du constructeur `With`, sera rejeté par Coq :

```
Definition getSecondWith {s : Type} {t : Type} (x : dependancy s t) :
   isWith x -> first :=
      ...
```

En effet, il ne connaît pas le type `first`. Pour ce cas particulier, l'algorithme de génération de code produit ceci :

```
Definition getSecondWith {s : Type} {t : Type} (x : dependancy s t) :
   forall (precondition : isWith x), getFirstWith x precondition :=
      match x return
         forall (precondition : isWith x), getFirstWith x precondition with
              With _ second => fun precondition => second
            | _ => fun precondition => match precondition with end
      end.
```

où `getFirstWith` est l'accesseur sur l'argument `first` du constructeur `With`.

Exemple 2.18 — Relation de sous-terme. Soit `t` un type inductif ; soient `v` et `w` deux valeurs de ce type. `v` est un sous-terme de `w` si et seulement si, à un moment donné, `v` a été utilisé pour construire `w`. Par exemple, dans le cas du type `tree`, `Leaf nat` est un sous-terme de `Node nat (Leaf nat) 397 (Leaf nat)`.

On se souvient qu'en théorie des ensembles, une relation $\mathscr{R}$ sur un ensemble E est un ensemble de couples $\langle x, y\rangle \in E^2$; x et y sont dits *en relation* (notation : $x \mathscr{R} y$) si et seulement s'il existe un couple $\langle x, y\rangle \in \mathscr{R}$. Ce qui vaut pour les ensembles vaut aussi pour les types. Une relation `R` sur un type `t` est un type prenant deux paramètres : `x : t` et `y : t` ; `x` et `y` sont dits *en relation* si et seulement s'il existe une valeur `v : R x y`. Voici par exemple la relation de sous-terme pour le type `tree` :

```
Inductive subterm {t : Type} (x : tree t) : tree t -> Type :=
    SubSelf : subterm x x
  | SubLeft : forall leftSubtree root rightSubtree,
      subterm x (Node t leftSubtree root rightSubtree) ->
         subterm x leftSubtree
  | SubRight : forall leftSubtree root rightSubtree,
      subterm x (Node t leftSubtree root rightSubtree) ->
         subterm x rightSubtree.
```

`x : tree t` est un sous-terme de `y : tree t` si et seulement s'il existe une valeur `v : subterm y x`. Ce code définit en fait trois règles de déduction :

$$\frac{}{\texttt{subterm x x}}\ \texttt{SubSelf}$$

$$\frac{\texttt{subterm x (Node t leftSubtree root rightSubtree)}}{\texttt{subterm x leftSubtree}}\ \texttt{SubLeft}$$

$$\frac{\texttt{subterm x (Node t leftSubtree root rightSubtree)}}{\texttt{subterm x rightSubtree}}\ \texttt{SubRight}$$

La première, `SubSelf`, dit que tout arbre `x` est un sous-terme de lui-même. La seconde et la dernière expriment que si l'arbre `y = Node t lS r rS` est un sous-terme de `x`, ses sous-arbres `lS` et `rS` le sont aussi.

Montrons que `Leaf nat` est un sous-terme de lui-même. Nous devons construire une valeur `v : subterm (Leaf nat) (Leaf)` ; `v = SubSelf (Leaf nat)` convient :

```
Coq < Check SubSelf (Leaf nat).
SubSelf (Leaf nat) : subterm (Leaf nat) (Leaf nat)
```

En fait, on voit bien comment définir une fonction prenant un arbre en entrée et retournant la preuve que celui-ci est un sous-terme de lui-même :

```
Definition reflexive {t : Type} (x : tree t) : subterm x x := SubSelf x.
Coq < Compute reflexive (Leaf nat).
 = SubSelf (Leaf nat) : subterm (Leaf nat) (Leaf nat)
Coq < Definition node := Node nat (Leaf nat) 397 (Leaf nat).
 node is defined
Coq < Compute reflexive node.
 = SubSelf node : subterm node node
```

2.d Processus de développement

Le TER s'est déroulé selon un processus itératif. Nous faisions le point chaque semaine avec l'encadrant pour contrôler l'ajout des nouvelles fonctionnalités.

Au début, le langage Tif ne supportait pas les types paramétrés. On ne pouvait écrire que des types non génériques comme `bool` (exemple 2.2) ou `nat` (exemple 2.6). Du 15 au 22 avril nous avons implémenté la prise en compte des paramètres. Du 22 avril au 6 mai, nous nous sommes occupés des observateurs. Du 6 au 13 mai, des accesseurs. Du 13 au 20 mai, nous avons résolu le problème des types avec dépendances et généré la relation de sous-terme. Enfin, du 20 au 27 mai, nous avons travailler sur la preuve de la transitivité de celle-ci.

2.e Les types inductifs dans différents paradigmes

Lors du TER, nous avons brièvement abordé le passage de Tif à Ocaml, Java ou C. L'objectif est de traduire dans ces trois langages, chacun préconisant un paradigme différent (fonctionnel pour Ocaml, objet pour Java, et procédural pour le C), le type `t` suivant :

```
type t (p1 : Type) (p2 : Type) ... (pi : Type) =
     C1 (a11 : ta11) (a12 : ta12) ... (a1j : ta1j)
   | C2 (a21 : ta21) (a22 : ta22) ... (a2k : ta2k)
   | ...
   | Cn (an1 : tan1) (an2 : tan2) ... (anp : tanp)
```

p*i*, C*i*, a*ij* et ta*ij* abrègent respectivement « paramètre *i* », « constructeur *i* », « argument *j* du constructeur *i* », et « type de l'argument *j* du constructeur *i* ». Nous illustrerons la méthode avec l'exemple 2.10 des graphes.

L'équivalent Ocaml du type `t` ne pose aucune difficulté :

```
type ('p1, 'p2, ..., 'pi) t =
     C1 of ta11 * ta12 * ... * ta1j
   | C2 of ta21 * ta22 * ... * ta2k
   | ...
   | Cn of tan1 * tan2 * ... tanp
```

Il faut seulement veiller à traduire correctement chaque ta*ij*.

Exemple 2.19 — graphes, cf. 2.10. En Ocaml le type `graph` devient :

```
type 't graph =
    Empty
  | AddVertex of 't * 't graphe
  | AddEdge of 't * 't * 't graphe
```

En Java, le type `t` devient l'interface `T` :

```
public interface T<P1, P2, ..., Pi> {
   //TODO Lister les méthodes du type T
}
```

Exemple 2.20 — graphes, cf. 2.10. Dans le cas du type `graphe`, on a :

```
public interface Graph<T> {
   //Exemples de méthodes possibles
   public boolean isEmpty();
   public boolean containsVertex(T vertex);
   public boolean containsEdge(T start, T end);
}
```

Ensuite, chaque constructeur C*n* est transformé en une classe implémentant l'interface `T` ; ses arguments devenant ses attributs :

```
public class Cn<P1, P2, ..., Pi> implements T<P1, P2, ..., Pi> {
   private tan1 an1;
   private tan2 an2;
   ...
   private tanp anp;
   public Cn(tan1 an1, tan2 an2, ..., tanp anp) {
      this.an1 = an1;
      this.an2 = an2;
      ...
      this.anp = anp;
   }
   //TODO Implémenter les méthodes de l'interface T
}
```

Exemple 2.21 — graphes, cf. 2.20. Dans le cas du type `graph`, on obtient :

```java
public class Empty<T> implements Graph<T> {
    public Empty () {
    }
    public boolean isEmpty() {
        // Un graphe vide est vide
        return true;
    }
    public boolean containsVertex(T vertex) {
        // Un graphe vide n'a pas de sommets
        return false;
    }
    public boolean containsEdge(T start, T end) {
        // Un graphe vide n'a pas d'arcs
        return false;
    }
}

public class AddVertex<T> implements Graph<T> {
    private T vertex;
    private Graph<T> subgraph;
    public AddVertex(T vertex, Graph<T> subgraph) {
        this.vertex = vertex;
        this.subgraph = subgraph;
    }
    public boolean isEmpty() {
        // Après ajout d'un sommet, un graphe n'est plus vide
        return false;
    }
    public boolean containsVertex(T vertex) {
        return this.vertex.equals(vertex)?
            // Cas de base : le sommet en argument est le sommet
            // ajouté lors de la construction
            true :
            // Cas récursif : regarder si le sommet en argument
            // se trouve dans le sous-graphe
            this.subgraph.containsVertex(vertex);
    }
    public boolean containsEdge(T start, T end) {
        // L'ajout d'un sommet ne modifie pas la disposition
        // des arcs
        return this.subgraph.containsEdge(start, end);
    }
}

public class AddEdge<T> implements Graph<T> {
    private T start;
    private T end;
    private Graph<T> subgraph;
    public AddEdge(T start, T end, Graph<T> subgraph) {
        this.start = start;
```

```
            this.end = end;
            this.subgraph = subgraph;
        }
        public boolean isEmpty() {
            // L'ajout d'un arc suppose un sous-graphe non vide
            return false;
        }
        public boolean containsVertex(T vertex) {
            // L'ajout d'un arc ne modifie pas la disposition
            // des sommets
            return this.subgraph.containsVertex(vertex);
        }
        public boolean containsEdge(T start, T end) {
            return this.start.equals(start) && this.end.equals(end)?
                // Cas de base : les sommets en arguments sont ceux de
                // l'arc que l'on a ajouté lors de la construction
                true :
                // Cas récursif : regarder si l'arc se touve
                // dans le sous-graphe
                this.subgraph.contains(start, end);
        }
    }
```

On construit un graphe `g` ayant pour sommets $\{1, 2, 3\}$ et pour arcs $\{\langle 1, 2\rangle, \langle 2, 3\rangle\}$ avec les instructions suivantes :

```
Graph<Integer> g = new Empty<>();
g = new AddVertex<>(1, g);
g = new AddVertex<>(2, g);
g = new AddVertex<>(3, g);
g = new AddEdge<>(1, 2, g);
g = new AddEdge<>(2, 3, g);
```

À titre comparatif, voici comment parvenir au même résultat en langage C :

Exemple 2.22 — graphes, cf. 2.10. On place l'interface dans un fichier `graph.h` :

```
#include <stdbool.h>
#include "t.h" /* Type T avec opération : bool equals(T t1, T t2) */
/* Cache l'implémentation */
typedef struct graph * Graph;
/* Constructeurs */
Graph empty();
Graph addVertex(T vertex, Graph subgraph);
Graph addEdge(T start, T end, Graph subgraph);
/* Opérations disponibles */
bool isEmpty(Graph graph);
bool containsVertex(Graph graph, T vertex);
bool containsEdge(Graph graph, T start, T end);
```

et l'implémentation dans un fichier `graph.c`

```
#include "graph.h"
/* Énumération des constructeurs */
typedef enum constructor {
   EMPTY,
   ADD_VERTEX,
   ADD_EDGE
} Constructor;
/* Objet construit par le constructeur EMPTY */
typedef struct empty {
   void * nothing;
} Empty;
/* Objet construit par le constructeur ADD_VERTEX */
typedef struct addVertex {
   T vertex;
   Graph subgraph;
} AddVertex;
/* Objet construit par le constructeur ADD_EDGE */
typedef struct addEgde {
   T start;
   T end;
   Graph subgraph;
} AddEdge;
/* Union discriminée : un graphe est soit vide, soit le résultat
 * de l'ajout d'un sommet ou d'un arc entre deux sommets
 */
typedef union value {
   Empty empty;
   AddVertex addVertex;
   AddEdge addEdge;
} Value;
/* Implémentation du type Graph */
struct graph {
   Constructor constructor;
   Value value;
}
/* Implémentation des constructeurs */
Graph empty() {
   Graph result = (Graph)(malloc(sizeof(struct graph)));
   result->constructor = EMPTY;
   result->value.empty.nothing = null;
   return result;
}
Graph addVertex(T vertex, Graph subgraph) {
   Graph result = (Graph)(malloc(sizeof(struct graph)));
   result->constructor = ADD_VERTEX;
   result->value.addVertex.vertex = vertex;
   result->value.addVertex.subgraph = subgraph;
   return result;
}
Graph addEdge(T start, T end, Graph subgraph) {
```

```
    Graph result = (Graph)(malloc(sizeof(struct graph)));
    result->constructor = ADD_EDGE;
    result->value.addEdge.start = start;
    result->value.addEdge.end = end;
    result->value.addEdge.subgraph = subgraph;
    return result;
}
/* Implémentation des opérations */
bool isEmpty(Graph graph) {
    switch (graph->constructor) {
        case EMPTY:
            return true;
        case ADD_VERTEX:
            return false;
        case ADD_EDGE:
            return false;
    }
}
bool containsVertex(Graph graph, T vertex) {
    switch (graph->constructor) {
        case EMPTY:
            return false;
        case ADD_VERTEX:
            return equals(graph->value.addVertex.vertex, vertex)?
                true :
                containsVertex(graph->value.addVertex.subgraph, vertex);
        case ADD_EDGE:
            return containsVertex(graph->value.addEdge.subgraph, vertex);
    }
}
bool containsEdge(Graph graph, T start, T end) {
    switch (graph->constructor) {
        case EMPTY:
            return false;
        case ADD_VERTEX:
            return containsEdge(graph->value.addVertex.subgraph, start, end);
        case ADD_EDGE:
            return equals(graph->value.addEdge.start, start) &&
                  equals(graph->value.addEdge.end, end)?
                     true :
                     containsEdge(graph->value.addEdge.subgraph, start, end);
    }
}
```

Chapitre 3

Documentation du projet

Le programme contient quatre modules (figure 3.1).

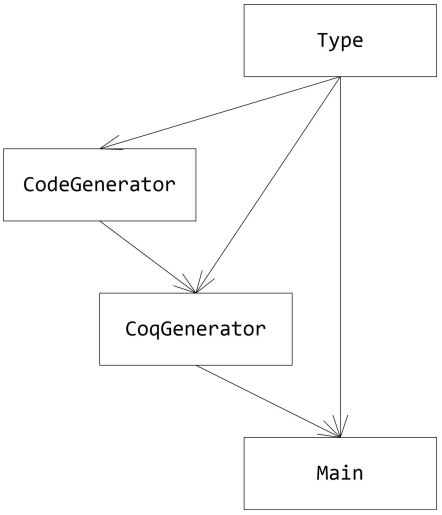

Figure 3.1 — Modules du programme. Ce diagramme montre les quatres modules du programme et leurs dépendances. Une flèche entre deux modules, par exemple entre `Type` et `CoqGenerator`, se lit et signifie « est appelé par » ; le module `Type` est appelé par le module `CoqGenerator`.

3.a Module Type

Le module `Type` fournit les outils nécessaires au stockage et au traitement d'un type inductif par Tifc.

3.a.i. Stockage d'un typage

```
type expression_t =
     Type of string * string list
   | Parenthesized of expression_t
   | Product of expression_t list
   | FunctionSignature of expression_t * expression_t
```

Le type `expression_t` permet de stocker le typage des paramètres et des arguments qui interviennent dans un type inductif. Par exemple :

```
Type ("natural", [])
```

correspond au typage `natural` ;

```
Type ("stack", ["t"])
```

correspond au typage `stack t` ;

```
Product [Type ("natural", []), Type ("stack", ["t"]), Type ("boolean", [])]
```

correspond au typage `natural * stack t * boolean` ; et

```
FunctionSignature (
   Parenthesized (
      FunctionSignature (
         Type ("natural", []),
         Type ("stack", ["t"]))),
      Type ("tree", ["t"]))
```

correspond au typage `(natural -> stack t) -> tree t`.

```
val dataType : expression_t -> string
```

La fonction `dataType` retourne, si l'expression en argument décrit un type simple (c.-à-d. qui n'est ni un produit, ni parenthésée, ni une signature de fonction), le nom de celui-ci sans ses paramètres de généricité. Dans le cas contraire, elle lève l'exception `NotAType`. Par exemple :

```
dataType (Type ("stack", ["t"])) = "stack".
```

```
val genericity : expression_t -> string list
```

La fonction `genericity` s'occupe, quant à elle, des paramètres de généricité :

```
genericity (Type ("stack", ["t"])) = ["t"].
```

```
val parenthesized : expression_t -> expression_t
```

La fonction `parenthesizd` retourne, si l'expression en argument est parenthésée, celle qui résulte de la suppression des parenthèses. Dans le cas contraire, elle lève l'exception `NotParenthesized`. Par exemple :

```
parenthesized (Parenthesized (Type ("stack", ["t"])))
                                          = Type ("stack", ["t"]).
```

```
val product : expression_t -> expression_t list
```

La fonction `product` retourne, si l'expression en argument est un produit d'expressions, la liste de celles-ci. Dans le cas contraire, elle lève l'exception `NotAProduct`. Par exemple :

```
product (Product [Type ("stack", ["t"]); Type ("tree", ["t"])])
                        = [Type ("stack", ["t"]); Type ("tree", ["t"])].
```

```
val domain : expression_t -> expression_t
```

La fonction `domain` retourne, si l'expression en argument est une signature de fonction, le domaine de celle-ci. Dans le cas contraire, elle lève l'exception `NotAFunctionSignature`. Par exemple :

```
domain (FunctionSignature (Type ("natural", []), Type ("stack", ["t"])))
                                                = Type ("natural", []).
```

```
val codomain : expression_t -> expression_t
```

La fonction `codomain`, quant à elle, s'occupe du codomaine :

```
codomain (FunctionSignature (Type ("natural", []), Type ("stack", ["t"])))
                                                = Type ("stack", ["t"])
```

3.a.ii. Stockage des paramètres et des arguments

```
type argument_t = Argument of string * expression_t
type parameter_t = Parameter of string * expression_t
val argIdentifier : argument_t -> string
val argType : argument_t -> expression_t
val paramIdentifier : parameter_t -> string
val paramType : parameter_t -> expression_t
```

Les paramètres et les argument qui interviennent dans un type inductif sont entièrement caractérisés par leur identificateur (c.-à-d. une chaîne) et leur typage (c.-à-d. une expression). La structure de données ci-dessus permet de les stocker en mémoire. Les fonctions `argIdentifier`, `argType`, `paramIdentifier` et `paramType` retournent respectivement l'identificateur d'un argument, le type d'un argument, l'identificateur d'un paramètre et le type d'un paramètre.

3.a.iii. Stockage des constructeurs

```
type constructor_t = Constructor of string * argument_t list
val consIdentifier : constructor_t -> string
val arguments : constructor_t -> argument_t list
```

Un constructeur est caractérisé par son nom (c.-à-d. une chaîne) et la liste de ses arguments. La structure de données ci-dessus permet de les stocker en mémoire. Les fonctions `consIdentifier` et `arguments` retournent respectivement l'identificateur et les arguments d'un constructeur.

3.a.iv. Stockage d'une déclaration complète

```
type declaration_t = Declaration of string * parameter_t list * constructor_t list
val declaredType : declaration_t -> string
val parameters : declaration_t -> parameter_t list
val constructors : declaration_t -> constructor_t list
```

Une déclaration est caratérisée par le nom du type déclaré (une chaîne), la liste de ses paramètres et celle de ses constructeurs. La structure de données ci-dessus permet de stocker une déclaration en mémoire. Les fonctions `declaredType`, `parameters` et `constructors` retournent respectivement l'identificateur du type déclaré, ses paramètres et ses arguments.

3.b Module CodeGenerator

```
module type SIG = sig
   val of_expression: Type.expression_t -> string
   val of_argument: Type.argument_t -> string
   val of_constructor: Type.constructor_t -> string
   val of_parameter: Type.parameter_t -> string
   val of_declaration: Type.declaration_t -> string
end
```

Cette signature spécifie que tout module permettant de traduire un type inductif dans un langage particulier, doit implémenter les fonctions `of_expression`, `of_argument`, `of_constructor`, `of_parameter` et `of_declaration` ; retournant respectivement le code correspondant à une expression, un argument, un constructeur, un paramètre et une déclaration.

Le module `CoqGenerator` implémente cette signature pour le langage Coq.

Chapitre 4

Bilan

Ce TER a été très intéressant. Les objectifs posés au début du stage ont été atteints. Il nous a montré comment prouver, aidé par la machine, des propriétés sur un type inductif. L'assistant de preuve Coq, très impressionant, montre rapidement les limites des langages fonctionnels comme Ocaml.