Ensembles nominaux dans Coq/SSreflect

Gabriel Lewertowski

Ensembles nominaux dans Coq/SSReflect

Gabriel Lewertowski,
sous la direction de Matthieu Sozeau et Nicolas Tabareau

21 août 2015

Introduction

Le contexte général

En raisonnant informellement sur des langages de programmation (λ-calcul, Système F, ...), il est courant de travailler "à α-équivalence près", en utilisant la convention de Barendregt ([Bar81], p. 26) : "Les termes α-équivalents sont identifiés. On écrit donc $\lambda x.x \equiv \lambda y.y$, etc."

En particulier, pour implémenter une fonction de substitution sans capture des variables, il faut être capable de renommer les variables liées "à la volée" : supposons qu'on veuille substituer dans $t := \lambda x.(x\ y)$ la variable y par $u := \lambda z.(x\ x)$, on doit dans un premier temps renommer x dans t en une variable fraîche, mettons w, puis faire la substitution pour obtenir $\lambda w.(w\ (\lambda z.(x\ x)))$. Si on ne renomme pas x avant de substituer, cette variable est capturée par le lieur λ.

Une autre utilisation de la convention de Barendregt consiste à supposer, dans une preuve par induction, qu'une variable liée est distincte d'un certain contexte donné. Par exemple, pour prouver le lemme de substitution défini ci-dessous, on raisonne par induction sur M, et dans le cas $M := \lambda z.M_1$: "Par la convention de Barendregt, on peut supposer que $z \neq x, y$ et $z \# (N, L)$" [Bar81].

```
Lemma substitution_lemma x y L M N :
  x # (y, L) -> M{x:=N}{y:=L} = M{y:=L}{x:=N{y:=L}}.
```

Une représentation des variables liées qui permet ce type de manipulations est difficile à mettre en oeuvre dans les assistants de preuves formelles. Plusieurs solutions existent :

- Les indices de De Bruijn [Pie02], où une variable liée est représentée par un entier qui mesure le nombre de lieurs à traverser pour retrouver celui qui lie la variable en question. Par exemple, le terme $t := \lambda x.\lambda y.(y\ x)$ est représenté par $\lambda.\lambda.(0\ 1)$. Cette solution présente l'inconvénient d'imposer le recours à des lemmes de décalage d'indices qui alourdissent les preuves.
- La représentation Locally Nameless [Cha09], qui utilise les indices de De Bruijn pour les variables liées, et des noms pour les variables libres.
- Higher Order Abstract Syntax [PE88], basée sur le lambda-calcul simplement typé enrichi par les produits et le polymorphisme.

Le problème étudié

Dans ce travail, on s'intéresse aux ensembles nominaux, un autre formalisme pour représenter les variables liées introduit par Andrew Pitts [Pit13]. Son principal avantage est qu'il permet de raisonner dans des assistants de preuves formelles comme on le ferait informellement avec un papier, sans recourir à des manipulations techniques (décalages d'indices, ...) qui augmentent le risque d'erreurs.

Cette approche a été implémentée dans l'assistant de preuve Isabelle [Urb05] et a permis de prouver plusieurs résultats intéressants : théorème de Church-Rosser, Normalisation forte du lambda-calcul simplement typé, première partie du POPLmark Challenge [Ayd05].

Plusieurs tentatives ont été faites pour développer les mêmes outils dans Coq, la plus aboutie date de 2006 [ABW06] et permet d'utiliser l'approche nominale pour raisonner sur le lambda-calcul, sans toutefois permettre à l'utilisateur de définir des fonctions par récurrence sur les termes.

Dans ce rapport, on propose une formalisation des ensembles nominaux dans Coq qui permet

— De représenter un langage de programmation quelconque fourni par l'utilisateur. La partie 3 traite le cas de $F_{<:}$, le langage étudié dans le POPLmark Challenge.
— De raisonner par induction en supposant qu'une variable liée est distincte d'un contexte donné
— De définir des fonctions récursives grâce à un combinateur

Pour ce faire, on ne part pas de zéro : le code qui, en gros, correspond à la partie 1 a été repris puis adapté d'un travail antérieur de Cyril Cohen.

La contribution proposée

Après avoir défini les notions relatives aux ensembles nominaux (partie 1), on définit (partie 2) la relation d'α-équivalence (notée $=_\alpha$). On s'aperçoit que cette définition n'est pas propre au lambda-calcul, mais qu'elle s'applique à n'importe quel langage qu'on peut encoder comme un *Arbre de Syntaxe Abstraite (AST)*, pourvu qu'on sache quels constructeurs lient des variables. On définit alors la relation d'α-équivalence en toute généralité sur les AST, puis on construit le type des AST quotientés par $=_\alpha$. On prouve ensuite un principe d'induction pour raisonner sur les AST quotients, puis un combinateur permettant de définir des fonctions récursives. Enfin (partie 3), on montre comment les termes d'un langage particulier, $F_{<:}$, s'encodent comme des AST, ce qui permet d'appliquer à ce langage le principe d'induction et le combinateur.

Tout le travail présenté dans ce rapport a été formalisé dans Coq, en utilisant la librairie SSreflect. On essaie autant que possible d'expliciter la syntaxte propre à cette librairie. En cas de doute, consulter le manuel [GMT15].

Le code est disponible à l'adresse : **https ://github.com/lewer/Nominal-Coq**

Pour le tester, il suffit d'avoir Coq8.5-beta2, SSreflect et la librairie MathComp. L'annexe B donne le contenu détaillé des différents fichiers.

Les arguments en faveur de sa validité

Voir annexe B

Le bilan et les perspectives

Ce travail pose les bases d'un plugin pour Coq, permettant à l'utilisateur de raisonner sur un langage dont il spécifie la grammaire. Pour devenir un plugin effectif, il lui manque essentiellement de l'automatisation pour :

— éviter à l'utilisateur d'avoir à énoncer et prouver (de manière immédiate) certains résultats. Par exemple, en définissant un langage L, l'utilisateur doit avoir *automatiquement* à disposition le principe d'induction *L_nominal_ind* correspondant (comme c'est le cas lorsqu'on définit un type inductif dans Coq). Pour cela, il faut écrire un plugin OCAML.
— prouver des conditions de bord (*fraîcheur* et *équivariance*, définies dans la partie 1).

Ces deux points sont discutés plus en détails dans la conclusion.

Remerciements

Je remercie mes directeurs de stage, Matthieu Sozeau et Nicolas Tabareau, qui m'ont souvent permis d'avancer par leurs réponses simples à mes questions compliquées. Je remercie également Cyril Cohen et Assia Mahboubi pour les nombreux conseils qu'ils ont pris le temps de me donner.

1 Ensembles nominaux

1.1 Atomes et permutations

Soit $\mathbb{A}$ un ensemble infini dont les éléments sont appelés *atomes*. Cette appellation signifie que la structure d'un atome n'a pas d'importance ; un atome est un atome et rien de plus. Pour pouvoir distinguer deux atomes différents, on suppose que l'égalité sur $\mathbb{A}$ est décidable. On note {**finperm** $\mathbb{A}$} le type des permutations sur $\mathbb{A}$ dont le support est fini, c'est à dire les fonctions $\pi : \mathbb{A} \to \mathbb{A}$ bijectives telles que l'ensemble $\{a \in \mathbb{A} \mid \pi a \neq a\}$ est fini. Pour manipuler de telles permutations dans Coq, on utilise la librairie **finmap.v** de Cyril Cohen, qui formalise à la fois le type {**fset** A} des ensembles finis d'éléments du type A, et {**fmap** $A \to B$}, le type des fonctions définies sur un sous-ensemble de A vers B permettant les opérations ensemblistes.

On note :

— $id_{\mathbb{A}}$ la permutation identité

— $\pi \circ \pi'$ la composée des deux permutations π et π'

— π^{-1} l'inverse de la permutation π

— $(a\ b)$ la transposition qui échange les atomes a et b, et laisse fixes les autres atomes

La *conjugaison* de la permutation γ par la permutation π est la permutation $\pi \circ \gamma \circ \pi^{-1}$. Dans le cas où γ est la transposition $(a\ b)$, sa conjugaison par π est la transposition $((\pi\ a)\ (\pi\ b))$, ce qu'on énonce sous la forme :

```
Lemma tfinperm_conj (a b : 𝔸) (π : {finperm 𝔸}) :
  π ∘ (a b) = ((π a) (π b)) ∘ π.
```

1.2 Définition et exemples

Le type {**finperm** $\mathbb{A}$} est un groupe pour la composition, d'élément neutre $id_{\mathbb{A}}$. On peut alors définir un permType comme un type sur lequel {**finperm** $\mathbb{A}$} agit, avec les axiomes usuels des actions de groupe.

```
Record permType (X : Type) := PermType {
act : {finperm 𝔸} -> X -> X;
  _ : forall x, act id𝔸 x = x;
  _ : forall π π' x, act (π ∘ π') x = act π (act π' x)
}.

Notation "π•x" := act π x.
```

Soit X : *permType* , et $x : X$. Un ensemble fini d'atomes A supporte x ssi :

$$\forall \pi : \{\textbf{finperm } \mathbb{A}\},\ (\forall a, a \in A \to \pi\ a = a) \to \pi \bullet x = x$$

Un *ensemble nominal* est un *permType* dont tous les éléments ont un support fini.

```
Record nominalType (X : choiceType) := NominalType {
  perm_type_of : permType X;
  support : X -> {fset 𝔸};
  _ : forall (x : X), (support x) \supports x.
```

Point technique

Pour pouvoir, par la suite, quotienter X par la relation d'α-équivalence, on requiert une structure de choiceType sur X, c'est à dire un opérateur de choix sur X (ce qui implique entre autres que X est muni d'une égalité décidable). En particulier, un `choiceType` est un ensemble au sens de la théorie des types homotopiques [Uni13].
Tout type qui peut s'encoder comme un arbre dont les noeuds sont étiquetés par des éléments d'un `choiceType C` est également un `choiceType`. Ceci suffit pour montrer que tous les types exhibés dans ce rapport sont effectivement des `choiceType`.

Exemple 1 (Ensemble nominal des atomes)**.**
Le groupe {**finperm** $\mathbb{A}$} agit naturellement sur $\mathbb{A}$:

```
Definition atomact (π : {finperm 𝔸}) (a : 𝔸) := π a.
```

Pour cette action, le singleton $\{a\}$ est un support fini pour a. Ainsi, $\mathbb{A}$ est un ensemble nominal.

Exemple 2 (Ensemble nominal trivial)**.**
Tout `choiceType` X peut être muni d'une structure d'ensemble nominal en faisant agir {**finperm** $\mathbb{A}$} trivialement sur X :

```
Definition trivialact {X : choiceType} (π : {finperm 𝔸}) (x : X) := x
```

L'ensemble vide $\emptyset$ est un support fini pour tout $x : X$.

Exemple 3 (Lambda-calcul)**.** On définit le lambda-calcul non typé de la manière suivante :

```
Inductive rterm :=
 |rVar of 𝔸
 |rApp of rterm & rterm
 |rLam of 𝔸 & rterm.
```

Remarque

On préfixe avec un "r" (pour *raw*) les types et les constructeurs qui correspondent aux termes *purs*, non quotientés par la relation d'α-équivalence.

Une permutation π : {**finperm** $\mathbb{A}$} agit sur un terme t en permutant chaque nom qui apparaît dans t :

```
Fixpoint rtermact (π : {finperm 𝔸}) (t : rterm) :=
 match t with
  |rVar a => rVar (π a)
  |rApp t1 t2 => rApp (rtermact π t1) (rtermact π t2)
  |rLam x t => rLam (π x) (rtermact π t)
 end.
```

L'ensemble des atomes qui apparaissent dans un terme t, défini ci-dessous inductivement sur la structure de t, est un support fini de t.

L'ensemble des termes `rterm` est donc un ensemble nominal.

```
Fixpoint rterm_support (t : rterm) :=
 match t with
  |rVar a => {a}
  |rApp t1 t2 => (rterm_support t1) ∪ (rterm_support t2)
  |rLam x t => {x} ∪ (rterm_support t)
 end.
```

Plus généralement,

Proposition 1. Soit X et Y deux ensembles nominaux et I un type fini,

- Le produit cartésien $X \times Y$ est un ensemble nominal pour l'action $\pi \bullet (x, y) = (\pi \bullet x, \pi \bullet y)$. Si S_x (resp. S_y) est un support de x (resp. y), alors $S_x \cup S_y$ est un support de (x, y).
- **List** X, le type des listes d'éléments de type X est un ensemble nominal pour l'action $\pi \bullet l =$ `map` $(\pi \bullet)$ l. On obtient un support d'une liste l en prenant l'union des supports des éléments de l.
- Le type des fonctions $I \to X$ est un ensemble nominal pour l'action $(\pi \bullet f)\ x = \pi \bullet (f\ x)$. Si pour $i : I$, S_i est un support de $f\ i$, alors $\cup_{i:I} S_i$ est un support de f.

1.3 Fraîcheur

L'exemple 3 donne une intuition de ce que représente `support x` : il s'agit des atomes "qui comptent" pour x.

La notion duale est celle de *fraîcheur*. Un atome a est *frais* dans x, ce qu'on note $a \# x$ si a "ne compte pas pour x". Dans le cas de l'exemple 3, $a \# t$ signifie que a n'apparaît pas dans t.

Dire que $a \# x$ n'est pas tout à fait synonyme de $a \notin$ `support x`, car la fonction `support` donnée pour construire un ensemble nominal n'est pas canonique. Cependant, on peut montrer que x possède un support minimal pour l'inclusion, `min_support`, obtenu par intersection de tous ses supports, en remarquant au préalable que la propriété "être un support" est close par intersection.

Une possibilité est alors de définir a # x := a $\notin$ `min_support x` ([Pit13], chap. 3), mais la condition de minimalité est délicate à manipuler en mathématiques constructives. On préfère donner une définition (équivalente) alternative :

```
Definition fresh {X : nominalType} (a : 𝔸) (x : X) :=
  exists S, S \supports x /\ a ∉ S.

Notation "a # x" := (fresh_in a x).
```

> **Remarque**
> On peut prouver $a \notin$ `support x` $\to a \#$ `x` mais pas la réciproque.

On termine cette sous-section par deux lemmes importants pour la suite.

Pour le premier, on repart de l'exemple 3 qui donne l'intuition de ce que signifie l'action d'une transposition sur un élément : $(a\ b) \bullet x$ se lit comme "renommer a en b et b en a dans x". Si a et b sont tous deux frais dans x, cette opération ne fait rien :

```
Lemma act_fresh (a b : 𝔸) (x : X) :
     a # x -> b # x -> (a b)•x = x.
```

Le deuxième lemme exprime la compatibilité de l'opérateur # avec l'action des permutations.

```
Lemma fresh_equiv (x : X) (a : 𝔸) (π : {finperm 𝔸}) :
  (a # x) <-> (π a) # (π•x).
```

Ce motif est un cas particulier *d'équivariance*, que l'on étudie dans la prochaine sous-section.

1.4 Équivariance et opérateur New

La "bonne" notion de fonction entre ensembles nominaux est celle de fonction *équivariante*, définie par :

```
Definition equivariant (f : X -> Y) :=
  forall (π : {finperm 𝔸}) (x : X), π•(f x) = f (π•x).
```

Ce qui induit la définition d'une proposition équivariante :

```
Definition equivariant_prop (R : X -> Prop) :=
  forall (π : {finperm 𝔸}) (x : X), R (π•x) <-> R x.
```

La proposition 1 qui montre que le produit cartésien d'ensembles nominaux est un ensemble nominal permet, par curryfication, d'appliquer le prédicat d'équivariance à une fonction qui prend plusieurs arguments. Dans la suite, si f est une fonction de plusieurs arguments, par exemple $f : X \to Y \to Z$, on se permettra d'écrire *equivariant* f pour signifier

$$equivariant\ (fun\ xy : (X * Y) \Rightarrow f\ (xy.1)\ (xy.2))$$

Par exemple :

```
Lemma transp_equivariant {X : nominalType} :
  equivariant (fun (a b : 𝔸) (x : X) => (a b)•x).
```

Les propositions équivariantes sont utiles lorsqu'on raisonne avec des atomes frais, car elles permettent la généralisation suivante :

```
Lemma ex_all (R : 𝔸 -> X -> Prop) (x : X) :
  equivariant_prop R ->
  (exists (a : 𝔸), a # x /\ R a x) -> forall (a : 𝔸), a # x -> R a x.
```

Démonstration.
Soit R une propriété équivariante, et a tel que $a \# x$ et $R\ a\ x$.
Soit b tel que $b \# x$, on veut prouver $R\ b\ x$.
Il suffit de prouver, $R\ ((a\ b)•b)\ ((a\ b)•x)$ par application de `equivariant_prop R`.
Or $(a\ b)•b = a$, et $(a\ b)•x = x$ d'après le lemme `act_fresh`. □

Une proposition équivariante vraie pour un atome frais est donc vraie pour presque tous les atomes frais. Cette notion de "vraie en dehors d'un ensemble fini d'atomes", sera utile par la suite ; on l'exprime grâce au quantificateur *new*, noté И, défini de la manière suivante :

```
Definition new (P : 𝔸 -> Prop) :=
  exists A : {fset 𝔸}, forall a, a # A -> P a.

Notation "И a , P" := (new (fun a : 𝔸 => P))
```

La proposition $Иx, (Px)$ signifie : il existe un ensemble d'atomes S tel que $P\ a$ est vérifiée pour tout atome $a \# S$. Cette formulation est équivalente à celle donnée dans le lemme `ex_all` :

```
Theorem some_any (R : 𝔸 -> X -> Prop) (x : X) :
  equivariant_prop R ->
  (exists (z : 𝔸), z # x /\ R z x) <-> И z, R z x.
```

Démonstration. L'implication directe est une conséquence du lemme `ex_all`. Pour la réciproque, étant donné A : {fset 𝔸} tel que pour tout $z \# A$, $R\ z\ x$ est vérifiée. N'importe quel atome frais à la fois dans x et dans A convient. □

La preuve précédente montre qu'on a parfois besoin, étant donné un ou plusieurs éléments d'ensembles nominaux, de construire un atome frais dans chacun de ces éléments. Tout $x : X$ ayant un support fini, et l'ensemble des atomes étant infini, on peut construire une fonction qui à chaque $x : X$ renvoie un atome frais dans x :

```
Definition fresh_atom : X -> 𝔸 := ...
Lemma fresh_in (x : X) : (fresh_atom x) # x.
```

En utilisant la proposition 1 qui montre que le produit cartésien d'ensembles nominaux est un ensemble nominal, on peut utiliser la fonction `fresh_atom` pour construire un nom frais dans plusieurs éléments à la fois : si $(x_1 : X_1), \ldots, (x_n : X_n)$ avec $X_1, \ldots, X_n$ des ensembles nominaux, *fresh_atom* $(x_1, \ldots, x_n)$ est un atome frais dans tous les x_i.

2 α-équivalence pour rNAST

2.1 La relation d'α-équivalence

Dans cette sous-section, on cherche à définir la relation d'α-équivalence (notée $=_\alpha$), qui exprime le fait que deux termes sont égaux, au renommage près de leurs variables liées. Commençons par le lambda-calcul :

- Deux variables `rVar x` et `rVar y` sont α-équivalentes ssi `x = y` (ces expressions ne contiennent pas de variable liée)
- `rApp t1 t2` $=_\alpha$ `rApp u1 u2` ssi `t1` $=_\alpha$ `u1` et `t2` $=_\alpha$ `u2`.
- `rLam x t` et `rLam y u` sont α-équivalents lorsqu'en prenant une variable fraîche z, qui n'apparaît ni dans `t` ni dans `u`, et en substituant x par z dans t et y par z dans u, les deux termes obtenus sont α-équivalents. Dans le paradigme des ensembles nominaux, substituer x par z dans t revient exactement à faire agir la transposition $(x\ z)$ sur t.

```
Inductive alpha_rec (n : nat) (t1 t2 : rawterm) : bool :=
 match n, t1, t2 with
  |_, rVar x, rVar y => x == y
  |S n, rApp t1 t2, rApp u1 u2 => (alpha_rec n t1 u1) && (alpha_rec n t2 u2)
  |S n, rLam x t, rLam y u =>
    let z := fresh_atom (x, y, t, u) in
    (alpha_rec n (x z)•t (y z)•u
 end.

Definition alpha t1 t2 := alpha_rec (raw_depth t1) t1 t2.
```

Point technique

- L'appel récursif à `alpha_rec` dans le cas `rLam` n'est pas *structurellement* plus petit : $(x\ z)\bullet t$ n'est pas un sous-terme de `rLam x t`. C'est la raison pour laquelle on définit `alpha_rec` inductivement, non pas sur la structure des termes, mais sur leur hauteur.
- Le but de cette partie est de définir le type quotient `term` $=$ `rawterm/`$=_\alpha$, encodé dans SSReflect comme un Σ-type. Afin que chaque classe d'équivalence ne contienne qu'un seul représentant, il faut qu'une preuve d'α-équivalence `t` $=_\alpha$ `u` soit unique [Coh13], ce qui n'est pas le cas si cette preuve est de type `Prop` sans utiliser l'axiome *UIP* [HS96]. En définissant `alpha` comme un prédicat booléen, puis $t =_\alpha u :=$ (`alpha` $t\ u = true$), on s'assure qu'une preuve $t =_\alpha u$ est unique, car le type **bool** vérifie *UIP*.

La définition précédente du prédicat `alpha` n'est pas spécifique au lambda-calcul : elle s'applique à n'importe quel langage de programmation dont les termes peuvent être représentés par un *Arbre de Syntaxe Abstraite (AST)* dont les feuilles sont des atomes, et dont les constructeurs sont de trois sortes différentes :

- Un constructeur *Var*, qui représente les variables.
- Un ou plusieurs constructeurs *Cons* qui ne lient pas de variable (comme le constructeur *rApp* du lambda-calcul). Ces constructeurs ont une arité fixée (2 pour *rApp*) et peuvent éventuellement prendre des annotations.
- Un ou plusieurs constructeurs *BCons* qui lient une variable (à l'instar du constructeur *rLam*). Ces constructeurs ont une arité et peuvent également prendre des annotations (par exemple,

pour le lambda-calcul simplement typé, le constructeur *rLam* prend comme annotation le type de la variable liée).

On appelle *NAST* ("N" pour Nominal) un langage de programmation de la forme décrite ci-dessus. On cherche à donner une signature abstraite d'un *NAST*, ce qui permet dans la suite d'énoncer des lemmes non seulement sur le lambda-calcul, mais sur tout langage (lambda-calcul typé, Système F, ...) qui peut s'encoder comme un *NAST*.

Un *NAST* peut avoir un nombre de constructeurs arbitraire. Pour modéliser ce fait, on implémente *Cons* et *BCons* comme des familles de constructeurs, indexées par un *label* qui identifie chaque constructeur. Pour décider si deux labels sont égaux, le type qui les représente doit être muni d'une structure de `eqType` qui indique que l'égalité est décidable.

Une fonction d'arité donne pour chaque label le nombre de fils attendus.

```
Inductive rNAST (cons_label : eqType)
                (bcons_label : eqType)
                (cons_annot : cons_label -> eqType)
                (bcons_annot : bcons_label -> eqType)
                (cons_arity : cons_label -> nat)
                (bcons_arity : bcons_label -> nat) :=
|rVar of 𝔸
|rCons : forall (c : cons_label),
          cons_annot c -> ('I_(cons_arity c) -> rNAST) -> rNAST
|rBCons : forall (c : bcons_label) (x : 𝔸),
           bcons_annot c -> ('I_(bcons_arity c) -> rNAST) -> rNAST.
```

Point technique

Le type `'I_n` désigne dans SSReflect l'intervalle d'entiers $[0; n-1]$.

Avant de faire quelques remarques, on montre comment le lambda-calcul simplement typé est une instance de *rNAST*.

```
Inductive typ :=
  |SimpleType of nat
  |ArrowType of typ & typ.

Inductive cons_label := app.
Inductive bcons_label := lam.
Definition cons_anot (c : cons_label) := unit. (* app ne prend aucune annotation *)
Definition bcons_annot (c : bcons_label) := typ. (* lam prend en annotation le type de la
    variable liee *)
Definition cons_arity (c : cons_label) := 2. (* arite de app *)
Definition bcons_arity (c : bcons_label) := 1. (* arite de lam *)

Definition simply_typed_lambdac := rNAST cons_label bcons_label
                                    cons_annot bcons_annot
                                    cons_arity bcons_arity.
```

Remarque. Le type `'I_n -> rNAST` qui apparaît dans la définition de `rNAST` est une fonction qui à chaque entier $i \in [0; n-1]$ renvoie un élément de type `rNAST`. Les constructeurs `Cons` et `BCons` prennent donc en argument une fonction f, qui appliquée en l'entier i renvoie le i^{e} fils de l'arbre. C'est équivalent à donner une liste d'éléments de type `rNAST` de taille fixée n, dont le type est noté {`n.-tuple rNAST`} dans SSReflect. Par rapport au type `'I_n -> rNAST`, le type

{`n.-tuple rNAST`} présente l'avantage de vérifier *l'égalité extensionnelle* : deux `n.-tuple` sont égaux ssi ils contiennent les mêmes éléments. Pour avoir le même résultat avec l'encodage `'I_n -> rNAST`, il faut supposer *l'axiome d'extensionalité*, au moins pour les fonctions dont le domaine est un type fini :

```
Axiom funext {I : finType} (X : Type) (f g : I -> X) :
  forall i, f i = g i -> f = g.
```

Cependant, la définition de `rNAST` dans laquelle on remplace `'I_n -> rNAST` par {`n.-tuple rNAST`} n'est pas validée par Coq , à cause de la condition de stricte-positivité requise pour définir un type inductif [Cas07].

Pour éviter d'utiliser l'axiome *funext*, on peut dans la définition du type rNAST indiquer que *Cons* et *BCons* attendent une liste de fils, sans spécifier sa taille. Pour éviter que puissent exister des passagers clandestins (comme $App\ t_1\ t_2\ t_3$), il faut ensuite filtrer le type rNAST par un prédicat `well_formed` qui spécifie les termes bien formés. Cette approche a été tentée (voir fichier **ast.v**) mais elle est plus compliquée à mettre en oeuvre que l'approche finalement choisie.

Le prédicat d'α-équivalence se définit sur le type `rNAST` de la même manière que pour le lambda-calcul, à une difficulté technique près : étant donné deux termes, `Cons c1 a1 f1` et `Cons c2 a2 f2`, on exprime le fait qu'ils sont α-équivalents en disant qu'ils ont même constructeur (`c1 == c2`), même annotation (`a1 == a2`), et des termes-fils deux à deux α-équivalents :
`forall i, i < cons_arity c1 -> f1 i` $=_\alpha$ `f2 i`.

Mais dans un terme, le type de l'annotation dépend du label du constructeur : `a1` a type `cons_annot c1`, et `a2` a type `cons_annot c2`, si bien qu'à priori, l'égalité `a1 == a2` n'est pas licite puisque `a1` et `a2` ne sont pas du même type. Pour comparer `a1` et `a2`, il faut au préalable avoir une preuve `p : c1 = c2`, et transporter `a1` le long de l'égalité `p`. La définition exacte du prédicat `alpha` est disponible à l'annexe A. On préfère donner ici une définition informelle mais plus lisible.

```
Fixpoint alpha_rec (n : nat) (t1 t2 : rNAST ) :=
  match n, t1, t2 with
    |_, rVar x, rVar y => x == y
    |S n, rCons c1 a1 f1, rCons c2 a2 f2 =>
        (c1 == c2) && (a1 == a2) &&
        [forall i < cons_arity c1, alpha_rec n (f1 i) (f2 i)]
    |S n, rBCons c1 x1 a1 f1, rBCons c2 x2 a2 f2 =>
        (c1 == c2) && (a1 == a2) &&
        let z := fresh_atom (x1, x2, f1, f2) in
        [forall i < bcons_arity c1, alpha_rec n (x1 z)•(f1 i) (x2 z)•(f2 i)]
    |_, _,_ => false
  end.

Definition alpha (t1 t2 : rNAST) := alpha_rec (raw_depth t1) t1 t2.

Lemma alpha_recE (n : nat) (t1 t2 : rNAST) :
  (rW_depth t1 <= n) -> alpha_rec n t1 t2 = alpha t1 t2.
```

Remarque
D'après la proposition 1, le type $I_n \to rNAST$ est un ensemble nominal. La notation *fresh_atom* f_1 a donc bien du sens, et désigne un atome frais dans $\cup_{i<n} f_1\ i$.

Pour construire le quotient `term` = `rawterm/`$=_\alpha$, il faut montrer que `alpha` est une relation d'équivalence, autrement dit qu'elle est reflexive, symmétrique et transitive. Si les deux premières propriétés ne posent pas de problème, on a besoin pour la troisième de savoir que `alpha` est une relation équivariante.

```
Lemma alpha_equivariant (t1 t2 : rNAST ) (π : {finperm 𝔸}):
  alpha (π•t1) (π•t2) = alpha t1 t2.
```

Démonstration. On raisonne par induction sur la hauteur de `t1`.

— Si `t1` est de hauteur 0, c'est une variable. Si `t2` n'est pas une variable, les membres gauche et droit de l'égalité qu'on cherche à démontrer s'évaluent à `false`. Si `t2` est une variable, on veut démontrer `(rVar (π • x) == rVar (π • y)) = (rVar x == rVar y))`, ce qui est vrai parce que π est une fonction injective.

— On suppose vrai le résultat lorsque `t1` est de hauteur inférieure ou égale à n, et on considère `t1` et `t2` deux termes, avec `t1` de hauteur $n+1$.

— Si `t1` est de la forme `Cons c a1 f1`, on peut supposer que `t2` est de la forme `Cons c a2 f2` (sinon, les deux membres de l'égalité à démontrer s'évaluent de nouveau en `false`). On veut alors montrer que pour `i < cons_arity c`,
`alpha (π • (f1 i)) (π • (f2 i)) = alpha (f1 i) (f2 i)`, ce qui est vrai par hypothèse d'induction.

— Si `t1` est de la forme `BCons c x1 a1 f1`, on se restreint (pour la même raison que précedemment) au cas où `t2` est de la forme `BCons c x2 a2 f2`, et on veut montrer que pour tout `i < bcons_arity v`,

$$alpha\ ((\pi\ x_1)\ z)\bullet(\pi\bullet(f_1\ i))\ ((\pi\ x_2)\ z)\bullet(\pi\bullet(f_2\ i)) = alpha\ (x_1\ z')\bullet(f_1\ i)\ (x_2\ z')\bullet(f_2\ i)$$

où `z := fresh_name (π x1,π x2,π • f1,π • f2)` et `z' := fresh_name (x1,x2,f1,f2)`. On réécrit successivement le membre gauche de l'égalité :

$$\begin{aligned}
&alpha\ ((\pi\ x_1)\ z)\bullet(\pi\bullet(f_1\ i))\ ((\pi\ x_2)\ z)\bullet(\pi\bullet(f_2\ i)) \\
&= alpha\ ((\pi\ x_1)\ z)\circ\pi)\bullet(f_1\ i))\ ((\pi\ x_2)\ z)\circ\pi)\bullet(f_2\ i)) && \text{(axiome des actions de groupe)} \\
&= alpha\ (\pi\circ(x_1\ (\pi^{-1}z))\bullet(f_1\ i))\ (\pi\circ(x_2\ (\pi^{-1}z))\bullet(f_2\ i)) && \text{(lemme \texttt{tfinperm_conj})} \\
&= alpha\ (x_1\ (\pi^{-1}z))\bullet(f_1\ i)\ (x_2\ (\pi^{-1}z))\bullet(f_2\ i) && \text{(hypothèse d'induction)}
\end{aligned}$$

ce qui est égal au membre droit de l'égalité dans lequel z' a été remplacé par $\pi^{-1}z$. On utilise à présent une manipulation fréquente dans les preuves sur les ensembles nominaux : pour remplacer la transposition $(x_1\ (\pi^{-1}\ z))$ par $(x_1\ z')$, il faut la conjuguer par la transposition $((\pi^{-1}\ z)\ z')$: $((\pi^{-1}\ z)\ z')\circ(x_1\ (\pi^{-1}\ z))\circ((\pi^{-1}\ z)\ z') = (x_1\ z')$. On continue la réécriture du membre gauche de l'égalité :

$$\begin{aligned}
&alpha\ (x_1\ (\pi^{-1}z))\bullet(f_1\ i)\ (x_2\ (\pi^{-1}z))\bullet(f_2\ i) \\
&alpha\ (((\pi^{-1}\ z)\ z')\circ(x_1\ (\pi^{-1}z)))\bullet(f_1\ i) && \text{(hyptohèse d'induction)} \\
&\qquad (((\pi^{-1}\ z)\ z')\circ(x_2\ (\pi^{-1}z)))\bullet(f_2\ i) \\
&= alpha\ ((\pi^{-1}\ z)\ z')\circ(x_1\ (\pi^{-1}z))\circ((\pi^{-1}\ z)\ z')\bullet(f_1\ i) \\
&\qquad ((\pi^{-1}\ z)\ z')\circ(x_2\ (\pi^{-1}z))\circ((\pi^{-1}\ z)\ z')\bullet(f_2\ i) \\
&= alpha\ (x_1\ z')\bullet(f_1\ i)\ (x_2\ z')\bullet(f_2\ i) && \text{(lemme \texttt{tfinperm_conj})}
\end{aligned}$$

l'avant-dernière égalité étant obtenue en remarquant que $(f_1\ i) = ((\pi^{-1}\ z)\ z')\bullet(f_1\ i)$ d'après le lemme `act_fresh`, le fait $z'\ \#\ (f_1\ i)$, et le fait $(\pi^{-1}\ z)\ \#\ (f_1\ i)$ obtenu avec le lemme `fresh_equiv`.

□

On peut désormais prouver que `alpha` est une relation d'équivalence.

Démonstration. On donne la démonstration de la transitivité : soient t_1, t_2 et t_3 des termes, on suppose $t_1 =_\alpha t_2$ et $t_2 =_\alpha t_3$, on veut montrer $t_1 =_\alpha t_3$.

On raisonne par induction sur la taille de t_1, et on traite le cas compliqué : t_1 est de la forme $BCons\ c\ x_1\ a\ f_1$, t_2 de la forme $BCons\ c\ x_2\ a\ f_2$, et t_3 de la forme $BCons\ c\ x_3\ a\ f_3$. La définition de `alpha` donne deux variables fraiches z et z' telles que pour tout $i <$ `bcons_arity c`,

$$(x_1\ z)\cdot(f_1\ i) =_\alpha (x_2\ z)\cdot(f_2\ i) \tag{1}$$
$$(x_2\ z')\cdot(f_2\ i) =_\alpha (x_3\ z')\cdot(f_3\ i) \tag{2}$$

et on veut montrer $(z''\ x_1)\cdot(f_1\ i) =_\alpha (z''\ x_3)\cdot(f_3\ i)$ pour $z'' := fresh\_atom(x_1, x_3, f_1, f_3)$. La difficulté réside dans le fait que les équations 1 et 2 ne sont valables que pour des valeurs particulières de z et z', à savoir $z := fresh\_atom(x_1, x_2, f_1, f_2)$ et $z' := fresh\_atom(x_2, x_3, f_2, f_3)$. On peut démontrer que ces équations sont en fait presque toujours vraies :

```
Lemma alpha_BConsP (c : bcons_label) (x1 x2 : 𝔸) a f1 f2 :
  reflect (let n := bcons_arity c in
           ∀z, (forall i, alpha (x1 z)•(f1 i) (x2 z)•(f2 i)))
          (alpha (rBCons c x1 a f1) (rBCons c x2 a f2)).
```

Point technique

La proposition `reflect P b` avec `P : Prop` et `b : bool` se lit : `P <-> b = true`. C'est une construction standard dans SSReflect, appelée la *réflexion booléenne* [GMT15].
Deux exemples :

```
Lemma andP (b1 b2 : bool) : reflect (b1 /\ b2) (b1 && b2)
Lemma eqP (T : eqType) (x1 x2 : T) : reflect (x1 = x2) (x1 == x2)
```

Démonstration. C'est une application du théorème `some_any`, en remarquant que la proposition `P := (fun (z x1 x2: 𝔸) (t1 t2: rNAST) => alpha (x1 z)•t1 (x2 z)•t2)` est équivariante en tant que composée de fonctions équivariantes (`alpha_equivariant, transp_equivariant`). □

Le lemme `alpha_BConsP` donne alors deux ensembles S_1 et S_2 tels que l'équation 1 (resp. 2) est vraie pour tout $z \notin S_1$ (resp. $z' \notin S_2$). On montre ensuite, en utilisant l'hypothèse d'induction, que

$$(x_1\ z)\cdot(f_1\ i) =_\alpha (x_3\ z)\cdot(f_2\ i) \text{ pour tout } z \notin (S_1 \cup S_2)$$

ce qui prouve que $BCons\ c\ x_1\ a\ f_1 =_\alpha BCons\ c\ x_3\ a\ f_3$, à nouveau par le lemme `alpha_BConsP`. □

2.2 Quotient

Ayant prouvé dans la section précédente que `alpha` est une relation d'équivalence décidable, SSReflect permet de construire `NAST`, le type quotient du type `rNAST` par la relation `alpha` ([Coh13]).

```
Definition NAST := {eq_quot alpha}.
```

Cette définition fournit deux fonctions : `pi : rNAST -> NAST`, la projection dans le quotient et `repr : NAST -> rNAST`, qui renvoie pour chaque terme du quotient un représentant. La fonction `repr` est un inverse droit de la fonction `pi` :

```
Lemma reprK : cancel repr pi. (* pi (repr x) = x *)
```

Les constructeurs Var, $Cons$ et $BCons$ sont obtenus par *relèvement* des constructeurs $rVar$, $rCons$ et $rBCons$:

```
Definition Var (x : 𝔸) : NAST := pi (rVar x).

Definition Cons (c : cons_label) (a : cons_annot c)
  (f : 'I_(cons_arity c) -> NAST) : NAST :=
pi (rCons c a (fun i => repr (f i))).

Definition BCons (c : bcons_label) (x : 𝔸) (a : bcons_annot c)
  (f : 'I_(bcons_arity c) -> NAST) : NAST :=
pi (rBCons c a (fun i => repr (f i))).
```

On peut définir une action de {**finperm** $\mathbb{A}$} sur $NAST$ qui en fait un ensemble nominal. C'est un résultat général : tout ensemble nominal peut être muni d'une relation d'α-équivalence, et le quotient par cette relation est également un ensemble nominal (appelé l'ensemble nominal des *abstractions de noms* [Pit13]). Une permutation agit sur le représentant d'un terme, et le résultat est projeté dans le quotient :

```
Definition termact (π : {finperm 𝔸}) (t : NAST) := pi (π • (repr t)).
```

Le support du représentant est toujours un support pour le terme :

```
Definition repr_support (t : NAST) : support (repr t) \supports t.
```

Il peut cependant arriver que le support minimal d'un terme quotienté soit strictement inclus dans le support minimal de son représentant. En reformulant avec la notion de fraîcheur, un atome peut être frais dans le terme quotienté sans être frais dans le représentant. Ainsi, on n'a pas $x \# rBCons\ c\ x\ a\ f$ mais ceci devient vrai dans le quotient :

```
Lemma bname_fresh x c a f : x # (BCons c x a f).
```

2.3 Principe d'induction NAST_ind

Comme pour tout type inductif, Coq définit pour $rNAST$ un principe d'induction qui établit qu'une proposition qui dépend d'un $rNAST$ est vérifiée pour tous les termes si elle est vérifiée pour les variables, et pour les termes de la forme *Cons* ou *BCons* en supposant qu'elle est vérifiée sur leurs fils.

Le type quotient $NAST$, lui, n'est pas un type inductif ; aucun mécanisme ne donne automatiquement de principe d'induction pour raisonner sur les $NAST$, il faut l'écrire et le prouver *à la main*.

```
Lemma NAST_naive_ind (P : NAST -> Prop) :
  (forall x, P (Var x)) ->
  (forall c a f, (forall i, P (f i)) -> P (Cons c a f)) ->
  (forall c x a f, (forall i, P (f i)) -> P (BCons c x a f)) ->
  forall (u : NAST), P u.
```

Démonstration. Immédiat par induction sur le représentant de u, $repr\ u : rNAST$. □

Toutefois, dans les raisonnements inductifs sur les langages de programmation, il est courant de renommer "à la volée" les variables liées en utilisant la convention de Barendregt. Par exemple, pour prouver le lemme de substitution défini ci-dessous, on raisonne par induction sur M, et dans le cas $M := Lam\ z.M_1$: "Par la convention de Barendregt, on peut supposer que $z \neq x, y$ et $z \# (N, L)$" [Bar81].

```
Lemma substitution_lemma x y L M N :
  x # (y, L) -> M{x:=N}{y:=L} = M{y:=L}{x:=N{y:=L}}.
```

Grâce à la structure d'ensemble nominal sur le type *NAST*, on peut définir et prouver un principe d'induction qui permet le renommage à la volée : dans le cas d'un terme de la forme $BCons\ c\ x\ a\ f$, on peut supposer que la variable liée x est fraîche dans un certain contexte donné en paramètre :

```
Lemma NAST_ind {ctxt : nominalType} (C : ctxt) (P : NAST -> Prop) :
  (forall x, P (Var x)) -> (* HVar *)
  (forall c a f, (forall i, P (f i)) -> P (Cons c a f)) -> (* HCons *)
  (forall c x a f, x # C -> (forall i, P (f i)) -> P (BCons c x a f)) ->(*HBCons*)
  forall (u : NAST), P u.
```

Point technique

Un contexte est représenté par un tuple de termes. Par exemple, si dans le cas du constructeur `BCons`, on veut que la variable `x` soit fraîche à la fois dans les atomes `y, z` et dans les termes `N, L`, on définit `ctxt := 𝔸*𝔸*NAST*NAST` et `C := (y, z, N, L)`.

Démonstration. Soit P une proposition qui vérifie les trois hypothèses *Hvar*, *HCons* et *HBcons*, et soit $u : NAST$. On montre un résultat plus fort que $P\ u$, à savoir $\forall\ \pi : \{\textbf{finperm}\ \mathbb{A}\},\ P(\pi\ u)$, et ceci par induction sur u en utilisant le principe d'induction naïf `NAST_naive_ind`.

— Les cas où u est de la forme *Var* ou *Cons* correspondent exactement aux deux premières hypothèses *HVar* et *HCons*.

— Si $u := BCons\ c\ x\ a\ f$.
Soit $\pi : \{\textbf{finperm}\ \mathbb{A}\}$, on veut montrer $P(\pi \bullet BCons\ c\ x\ a\ f)$, c'est à dire $P(BCons\ c\ (\pi\ x)\ a\ (\pi \bullet\ f))$.
On commence par prouver que

$$BCons\ c\ (\pi\ x)\ a\ (\pi \bullet f) = BCons\ c\ y\ a\ (((\pi\ x)\ y) \bullet \pi \bullet f) \quad \text{avec } y := fresh\_atom(\pi\ x,\ f,\ C)$$

grâce au lemme :

```
Lemma eq_BCons c x y a f :
  y # (x, f) -> BCons c x a f = BCons c y a ((x y)•f).
```

On obtient alors le résultat par application de la troisième hypothèse *HBCons*.

□

2.4 Principle d'élimination NAST_rect

Dans le paragraphe précédent, on a remarqué que le type quotient *NAST* n'est pas un type inductif, et que par conséquent il fallait énoncer et prouver à la main un principe d'induction *NAST_ind* pour raisonner sur ce type. Pour la même raison, il n'est pas possible de définir une fonction sur le type *NAST* en utilisant les outils usuels de Coq, comme l'opérateur de point fixe `Fixpoint` ou le déstructeur `match ...with`. Dans ce paragraphe, on montre comment définir un éliminateur *NAST_rect* qui permet de construire une fonction sur le type *NAST*, et on l'illustre en définissant une fonction de substitution qui évite la capture des variables et vérifie les équations :

```
Context (x : 𝔸) (t : NAST).

Lemma subst_VarE y : (Var y){x := t} = (if x == y then t else Var y).

Lemma subst_ConsE c a f : (Cons c a f){x := t} = Cons c a ({x := t} ∘ f).

Lemma subst_BConsE c y a f :
  y # (x, t) -> (BCons c y a f){x := t} = BCons c y a ({x := t} ∘ f).
```

Idéalement, si on pouvait définir une fonction sur le type *NAST* avec l'opérateur `Fixpoint`, une telle définition serait de la forme :

```
Fixpoint some_function (t : NAST) :=
  match t with
    |Var x => f_var x
    |Cons c a f => f_cons c a f (fun i => some_function (f i))
    |BCons c x a f => f_bcons c x a f (fun i => some_function (f i))
  end.
```

On se donne alors trois fonctions `f_var`, `f_cons`, `f_bcons` comme ci-dessus, et on cherche à définir une nouvelle fonction, `NAST_rect` qui se comporte comme `some_function`.

```
Context (X : nominalType 𝔸)
        (f_var : 𝔸 -> X)
        (f_cons : forall (c : cons_label), (cons_annot c) ->
                                        ('I_(cons_arity c) -> W) ->
                                        ('I_(cons_arity c) -> X) -> X)
        (f_bcons : forall (c : bcons_label), 𝔸 ->
                                          (bcons_annot c) ->
                                          ('I_(bcons_arity c) -> W) ->
                                          ('I_(bcons_arity c) -> X) -> X)
```

Remarque

On doit supposer que X, le codomaine de la fonction qu'on cherche à définir, est un ensemble nominal. Ceci n'est pas restrictif car comme le montre l'exemple 2, tout type peut être muni de l'action triviale qui en fait un ensemble nominal.

Plusieurs conditions sont requises sur `f_var`, `f_cons` et `f_bcons` afin que la fonction *NAST_rect* se comporte correctement. Tout d'abord, il faut qu'existe un ensemble *Supp* en dehors duquel `f_var`, `f_cons` et `f_bcons` sont équivariantes, ce qu'on écrit (pour `f_var`) :

```
Hypothesis f_var_equivariant :
  forall (π : {finperm 𝔸}) (x : 𝔸),
    [disjoint (finsupp π) & Supp] -> π•(f_var x) = f_var (π•x).
```

Point technique

L'ensemble `finsupp` π désigne le *support* de la permutation π, c'est à dire $\{a \in \mathbb{A} \mid \pi\ a \neq a\}$. Si `A` et `B` sont deux ensembles, `[disjoint A & B]` est un prédicat booléen qui exprime le fait que `A` et `B` sont disjoints, *i.e* `[disjoint A & B] := (A ∩ B == ∅)`

Enfin, puisqu'on souhaite définir une fonction sur les termes, c'est-à-dire sur les classes d'équivalence, cette fonction doit respecter `alpha` (*i.e* ne pas dépendre du représentant choisi). On reformule plus simplement cette condition en disant que `f_bcons` doit vérifier la *Condition de fraîcheur pour les lieurs (Freshness Condition for Binders, FCB)* [Pit06] :

```
Hypothesis FCB_fbcons :
  forall x c a f f', x # Supp -> x # (f_bcons c x a f f').
```

Avec ces quatre hypothèses, on peut définir `NAST_rect` et prouver que cette fonction se comporte comme attendu.

Le destructeur `match ...with`, s'il ne peut être utilisé pour raisonner par cas sur un élément du type `NAST`, peut en revanche être utilisé sur le type inductif `rNAST`.

On définit alors `NAST_rect (t : NAST)` par induction sur la taille de `t`, puis en raisonnant par cas sur le représentant de `t` :

```
Fixpoint NAST_rect_rec (n : nat) (t : NAST) (dflt : X) :=
  match n, (repr t) with
    |_, rVar x => f_var x
    |S n, rCons c a f => f_cons c a (pi ∘ f) ((NAST_rect_rec n) ∘ (pi ∘ f))
    |S n, rBCons c x a f =>
     let z := fresh_atom (Supp, rBCons c x a f) in
     f_bcons c z a
            ((x z)•(pi ∘ f))
            ((NAST_rect_rec n)•(x z)•(pi ∘ f)))
    |_, _ => dflt
  end.

Definition NAST_rect t := NAST_rect_rec (depth t) t.
```

Point technique

Même si elle ne joue aucun rôle, il faut renvoyer une valeur par défaut `dflt` pour avoir une disjonction de cas exhaustive.

La raison pour laquelle, dans le cas `rBCons c x a f`, on génère un nom frais n'est pas apparente à première vue. Il s'agit d'avoir un nom `z` pour lequel on peut appliquer l'hypothèse `FCB_fbcons`, comme on va le voir dans les preuves qui suivent.

On termine ce paragraphe en montrant que `NAST_rect` se comporte comme attendu, c'est à dire que son résultat lorsqu'elle est appliquée en un terme de la forme `Var x`, `Cons c a f` ou `BCons c x a f` est bien celui donné par la fonction `f_var`, `f_cons` ou `f_bcons` :

```
Lemma NAST_rect_VarE x : NAST_rect (Var x) = f_var x.

Lemma NAST_rect_ConsE c a f :
  NAST_rect (Cons c a f) = f_cons c a f (NAST_rect ∘ f).

Lemma NAST_rect_BConsE c x a f :
  x # Supp ->
  NAST_rect (BCons c x a f) = f_bcons c x a f (NAST_rect ∘ f).
```

Les preuves de `NAST_rect_VarE` et `NAST_rect_BConsE` découlent de la définition de `NAST_rect`. En revanche, la preuve de `NAST_rect_BConsE` est plus délicate ; elle occupe la fin de cette section.

Démonstration. Soit $x \# Supp$. On veut montrer

$$NAST\_rect\ (BCons\ c\ x\ a\ f) = f\_bcons\ c\ x\ a\ f\ (NAST\_rect \circ f)$$

Le représentant de $BCons\ c\ x\ a\ f$ est de la forme $rBCons\ c\ y\ a\ f'$, et on peut montrer que pour $z := fresh\_atom\ (Supp, rBCons\ c\ y\ a\ f')$,

$$(x\ z)\bullet f = (y\ z)\bullet(pi \circ f') \tag{3}$$

pour tout $z \# (x, y, f, f')$. On a alors :

$$\begin{aligned} NAST\_rect\ (BCons\ c\ x\ a\ f) &= f\_bcons\ c\ z\ a\ ((y\ z)\bullet(pi \circ f'))\ (NAST\_rect((y\ z)\bullet(pi \circ f'))) \\ &= f\_bcons\ c\ z\ a\ ((x\ z)\bullet f)\ (NAST\_rect\ ((x\ z)\bullet f)) && \text{(par (3))} \\ &= (w\ z)\bullet f\_bcons\ c\ z\ a\ ((x\ z)\bullet f)\ (NAST\_rect\ ((x\ z)\bullet f)) \end{aligned}$$

avec w un atome générique, frais dans tous les termes qui apparaissent dans cette preuve. Cette dernière égalité est obtenue avec le lemme `act_fresh` (la condition de fraîcheur pour z est donnée

par l'hypothèse `FCB_fbcons`. C'est ici que l'on voit pourquoi, dans le cas `BCons` de la définition de `AST_rect`, il faut prendre un atome frais dans *Supp*). Par équivariance, on a ensuite :

$$\begin{aligned} NAST\_rect\ (BCons\ c\ x\ a\ f) &= f\_bcons\ c\ w\ a\ ((w\ z)\bullet(x\ z)\bullet f)\ ((w\ z)\bullet NAST\_rect\ ((x\ z)\bullet f)) \\ &= f\_bcons\ c\ w\ a\ ((w\ x)\bullet(w\ z)\bullet f)\ ((w\ z)\bullet NAST\_rect\ ((x\ z)\bullet f)) \\ &= f\_bcons\ c\ w\ a\ ((w\ x)\bullet f)\ ((w\ z)\bullet NAST\_rect\ ((x\ z)\bullet f)) \end{aligned}$$

La deuxième ligne étant obtenue en utilisant le lemme `tfinperm_conj`, et la troisième le lemme `act_fresh`. On effectue à présent la même manipulation sur le terme droit de l'égalité qu'on cherche à démontrer :

$$\begin{aligned} f\_bcons\ c\ x\ a\ f\ (NAST\_rect \circ f) &= (w\ x)\bullet f\_bcons\ c\ x\ a\ f\ (NAST\_rect \circ f) \\ &= f\_bcons\ c\ w\ a\ ((w\ x)\bullet f)\ ((w\ x)\bullet NAST\_rect \circ f) \end{aligned}$$

Pour avoir le résultat, il suffit donc de prouver l'égalité

$$((w\ z)\bullet NAST\_rect\ ((x\ z)\bullet f)) = ((w\ x)\bullet NAST\_rect \circ f)$$

Cette dernière égalité est une conséquence de l'équivariance de *NAST_rect*, et de l'égalité observée plus haut $(w\ z)\bullet(x\ z)\bullet f = (w\ x)\bullet f$.

On s'est ainsi ramené à prouver que *NAST_rect* est une fonction équivariante. Ce résultat est technique et ne présente pas d'intérêt particulier. Le lecteur intéressé peut trouver la preuve Coq dans le fichier **w.v**, lemme `NAST_rect_equivariant`. □

2.5 Substitution sans capture de variable

L'éliminateur *NAST_rect* de la sous-section précédente permet de définir des fonctions sur les *NAST*. On montre comment l'utiliser pour définir une fonction de substitution sans capture des variables.

```
Section SubstDef.

Context (x : 𝔸) (t : W). (* Context fait pareil que Variables *)
Local Notation dflt := (Var 0).
Local Notation Supp := ({x} ∪ (support t)).

(* substitution  de x par t *)

Definition subst_var (y : 𝔸) :=
  if x == y then t else Var y.

Definition subst_cons ...

Definition subst_bcons ...

Definition subst :=
  @NAST_rect _ subst_var subst_cons subst_bcons Supp dflt.

End SubstDef.

Notation "u { x := t}" := (subst x t u).
```

L'équivariance des fonctions `subst_var`, `subst_cons` et `subst_bcons` en dehors de `Supp` se prouve facilement, essentiellement en dépliant les définitions.

La condition *FCB* pour `subst_bcons`, quant à elle, s'énonce ainsi :

```
Lemma FCB_subst_bcons y c a f f' : y # Supp -> y # (subst_bcons c y a f f').
```

Par définition de `subst_bcons`, ceci revient à prouver `y # BCons c y a f'` ; il s'agit du lemme `bname_fresh` énoncé plus haut.

Toutes les conditions sont requises pour obtenir les équations de `subst` :

```
Lemma subst_VarE y : (Var y){x := t} = (if x == y then t else Var y).

Lemma subst_ConsE c a f : (Cons c a f){x := t} = (Cons c a ({x := t} ∘ f).

Lemma subst_BConsE c y a f :
  y # (x, t) ->
  (BCons c y a f){x := t} = (BCons c y a ({x := t} ∘ f).
```

3 Exemple d'application : le Système $F_{<:}$

Le système F [GTL89] est un lambda-calcul typé avec polymorphisme de types. Dans cette partie, on s'intéresse à une extension du système F, $F_{<:}$, qui introduit le sous-typage. Le système $F_{<:}$ présente l'intérêt d'être minimal par rapport aux langages de programmation réellement existants (Java, C,...) tout en étant suffisamment riche pour soulever des problèmes intéressants portant entre autres sur les variables liées, et les principes d'induction complexes. Pour cette raison, c'est un langage "de référence" pour experimenter des assistants de preuve et des techniques de représentation des variables liées. Sa syntaxe se définit en deux temps : d'abord les types et ensuite les termes.

$T ::=$		*Types*
	X	*variable de type*
	Top	*type maximum*
	$T \to T$	*type des fonctions*
	$\forall(X <: T).T$	*quantification bornée*
$t ::=$		*Termes*
	x	*variable de terme*
	top	*terme canonique de type Top*
	$\lambda(x : T).t$	*fonction*
	$t\ t$	*application*
	$\Lambda(X <: T).t$	*type borné des fonctions*
	$t[T]$	*application de type*

On commence par représenter les types comme une instance de $NAST$:

```
Inductive typ_cons_label := ttop | arrow.
Inductive typ_bcons_label := fAll.

Definition typ_cons_annot (c : typ_cons_label) := unit.
Definition typ_bcons_annot (c : typ_bcons_label) := unit.

Definition typ_cons_arity (c : typ_cons_label) :=
  match c with
    |ttop => 0
    |arrow => 2
```

```
  end.
Definition typ_bcons_arity (c : typ_bcons_label) := 1.

Definition typ := NAST typ_cons_label typ_bcons_label
                      typ_cons_annot typ_bcons_annot
                      typ_cons_arity typ_bcons_arity.
```

Remarque

Ici, `typ` est directement une représentation modulo la relation d'α-équivalence des types de $F_{<:}$.

On définit maintenant la syntaxe des termes :

```
Inductive term_cons_label := top | app | tapp.
Inductive term_bcons_label := lam | abs.

Definition term_cons_annot (c : term_cons_label) :=
  match c with
    |top => unit
    |app => unit
    |tapp => typ
  end.
Definition term_bcons_annot (c : term_bcons_label) := typ.

Definition term_cons_arity (c : term_cons_label) :=
  match c with
   |top => 0
   |app => 2
   |tapp => 1
  end.
Definition term_bcons_arity (c : term_bcons_label) := 1.

Definition term := NAST term_cons_label term_bcons_label
                      term_cons_annot term_bcons_annot
                      term_cons_arity term_bcons_arity.
```

La structure de *NAST* doit être invisible pour l'utilisateur. Il faut pour cela définir des constructeurs `Lam : 𝔸 -> term -> term`, `App : term -> term -> term`, et ainsi de suite, en fonction de `Var`, `Cons` et `BCons`. Par exemple :

```
Definition Lam (x : 𝔸) (T : typ) (t : term) :=
  BCons lam x T (fun (i : 'I_1) => t).

Definition Top := ...
Definition App := ...
Definition Abs := ...
Definition TApp := ...
```

On peut alors démontrer un principe d'induction pour raisonner sur les termes, qui se dérive automatiquement du principe d'induction général *NAST_ind* de la partie 2.3. Ce principe permet, comme attendu, de supposer que les variables liées qui apparaissent dans un `term` sont fraîches dans un certain contexte donné :

```
Theorem term_ind {ctxt : nominalType} (C : ctxt) (P : term -> Prop) :
  (P Top) ->
  (forall x, P (Var x)) ->
  (forall x T t, x # C -> P t -> P (Lam x T t)) ->
  (forall t1 t2, P t1 -> P t2 -> P (App t1 t2)) ->
  (forall X T t, X # C -> P t -> P (Abs X T t)) ->
  (forall t T, P t -> P (TApp t T)) ->
  forall (u : term), P u.
```

De plus, en tant qu'instances de `NAST`, `term` et `typ` sont automatiquement pourvus d'une fonction de substitution `{x:=t}` qui vérifie les équations attendues :

— `(Var y){x:=t} = if x == y then t else Var y`
— `(App t1 t2){x:=t} = App t1{x:=t} t2{x:=t}`
— `(Lam y u){x:=t} = Lam y u{x:=t}` si `y#(x, t)`

La représentation nominale du langage $F_{<:}$ permet des raisonnements formels très similaires aux raisonnements sur papier. Une illustration est la solution du POPLmark Challenge écrite avec Nominal Isabelle [UPV+06].

On souhaiterait naturellement proposer une solution du POPLmark Challenge utilisant la librairie définie dans ce rapport, afin de la tester sur un cas pratique et la comparer aux solutions existantes. Mais avant cela, il faut rendre cette librairie plus simple à utiliser ; en particulier, les exemples précédents ont montré que la syntaxe utilisée pour définir un ensemble nominal puis une fonction récursive sur cet ensemble est verbeuse et assez inélégante.

On termine en discutant quelques pistes d'amélioration.

Conclusion et perspectives

Le type générique `NAST` pose les bases d'un plugin Coq permettant à l'utilisateur de définir à partir d'une signature (constructeurs, arités, ...) d'un langage de programmation, une représentation nominale de ce langage où les termes α-équivalents sont égaux. Fournir une telle signature est laborieux ; il faudrait pouvoir spécifier le langage plus simplement, avec une syntaxe proche de celle utilisée pour définir un type inductif en Coq (comme dans [Urb05]).

Le principe d'induction et l'éliminateur permettant de raisonner sur les termes, bien qu'ils soient immediats à prouver à partir des principes généraux `NAST_ind` et `NAST_rect`, doivent être énoncés à la main. Il faudrait que ces preuves soient faites automatiquement (avec un plugin écrit en OCAML), comme c'est le cas pour les types inductifs. De plus, pour définir une fonction avec l'éliminateur `NAST_rect`, il faut prouver que les fonctions `f_var`, `f_cons` et `f_bcons` sont équivariantes, ce qui est le plus souvent immédiat. Pour prouver ces résultats d'équivariance automatiquement, on aimerait appliquer le théorème de paramétricité [Wad89] (grâce à un plugin **paramcoq** écrit par Marc Lasson) en exprimant l'équivariance comme une relation paramétrée.

Enfin, dans les preuves réalisées dans le cadre de ce travail, il est très fréquent de devoir prouver des résultats de fraîcheur de la forme $\mathit{fresh\_atom}\,(x_1, x_2, \ldots, x_n) \# x_i$, ou bien $\mathit{fresh\_atom}\,(\ldots, f\ i, \ldots) \# f$. Actuellement, ces buts sont résolus par une tactique Ltac, mais il serait intéressant d'utiliser le mécanisme d'inférence de type des *Typeclasses* ([SN08]) pour les prouver automatiquement.

A Définition du prédicat `alpha`

```
Fixpoint alpha_rec (n : nat) (t1 t2 : rNAST ) :=
  match n, t1, t2 with
    |_, rVar x, rVar y => x == y
    |S n, rCons c1 a1 f1, rCons c2 a2 f2 =>
     match (eq_comparable c1 c2) with
       |left p => (eq_rect c1
                  (fun c => forall (a2 : cons_annot c)
                                   (f2 : 'I_(cons_arity c) -> rNAST), bool)
                  (fun a2 f2 =>
                     (a1 == a2) &&
                     [forall i in 'I_(cons_arity c1), alpha_rec n (f1 i) (f2 i)]) c2
                        p)
                    a2 f2
       |right _ => false
     end
    |S n, rBCons c1 x1 a1 f1, rBCons c2 x2 a2 f2 =>
     match (eq_comparable c1 c2) with
       |left p =>
        (eq_rect c1
                (fun c =>
                   forall
                     (x2 : 𝔸)
                     (a2 : bcons_annot c)
                     (f2 : 'I_(bcons_arity c) -> rNAST), bool)
                (fun x2 a2 f2 =>
                   (a1 == a2) &&
                    let m := bcons_arity c1 in
                    let z := fresh_atom (x1, x2, f1, f2) in
                    [forall i < m, alpha_rec n (swap x1 z \dot f1 i)
                                               (swap x2 z \dot f2 i)]) c2 p)
          x2 a2 f2
       |right _ => false
     end
    |_, _,_ => false
  end.

Definition alpha t1 t2 := alpha_rec (raw_depth t1) t1 t2.
```

B Contenu des fichiers de code

Les fichiers de code sont disponibles sur le dépot **https ://github.com/lewer/Nominal-Coq**.

— **finmap.v** (auteur Cyril Cohen), ≈ 2300 lignes de code, 90kb.
Définition des ensembles finis sur un type infini dénombrable, puis des fonctions dont le domaine est un ensemble fini.

— **finsfun.v**, ≈ 300 ldc, 10kb.
Définition des fonctions à support fini.

— **finperm.v**, ≈ 400 ldc, 12kb.
Définition des permutations finies, qui sont des fonctions à support fini bijectives.
Définition des transpositions.
Lemmes divers sur les permutations.

— **nominal.v**, ≈ 1000 ldc, 31kb.
Définition des ensembles nominaux.
Lemmes basiques.
Définition d'ensembles nominaux particuliers (trivial, produit, list, ...)
Résultats de fraicheur.
Équivariance.
Théorèmes some-any.

— **w.v**, ≈ 1000 ldc, 31kb.
Définition des `rAST`.
Définition de l'α-équivalence.
Quotient par α-équivalence.
Principes d'induction.

— **ast.v**, ≈ 1000 ldc, 32kb.
Définition des `rAST` avec une liste de fils au lieu d'une fonction qui renvoie le i^e fils.
Définition de l'α-équivalence.
Quotient par α-équivalence.
Principes d'induction.

Références

[ABW06] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages :Theory and Practice (LFMTP)*, Seattle, WA, USA, August 2006.

[Ayd05] Brian E. Aydemir. Mechanized metatheory for the masses : The poplmark challenge. 2005.

[Bar81] Hendrik Pieter Barendregt. *The lambda calculus : its syntax and semantics*. Studies in logic and the foundations of mathematics. North-Holland, 1981.

[Cas07] Eduardo Giménez & Pierre Castéran. A tutorial on [co-]inductive types in Coq. 2007.

[Cha09] Arthur Charguéraud. The locally nameless representation. 2009.

[Coh13] Cyril Cohen. Pragmatic quotient types in coq. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 213–228. Springer Berlin Heidelberg, 2013.

[GMT15] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France, 2015.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[HS96] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[Pit06] Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3) :459–506, May 2006.

[Pit13] Andrew M. Pitts. *Nominal Sets*. Cambridge tracts in theoretical computer science. Cambridge University Press, 2013.

[SN08] Matthieu Sozeau and Oury Nicolas. First-Class Type Classes. *Lecture notes in computer science*, August 2008.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory : Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[UPV+06] Christian Urban, Benjamin Pierce, Dimitrios Vytiniotis, Stéphanie Weirich, and Steve Zdancewic. Poplmark, nominal everywhere, in isabelle contents. 2006.

[Urb05] Christian Urban. Nominal techniques in isabelle/hol. In *Proceedings of the 20th International Conference on Automated Deduction (CADE-20*, pages 38–53. Springer, 2005.

[Wad89] Philip Wadler. Theorems for free ! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.