RÉALISATION MÉCANISÉE D'INTERPRÉTEURS ABSTRAITS

Stage de DEA réalisé par David Monniaux sous la direction de Patrick Cousot au laboratoire d'informatique de l'École normale supérieure, Paris, 1998.

Ce rapport est présenté pour l'obtention du diplôme d'études approfondies (DEA) *sémantique, preuves, programmation*, cohabilité par les universités Paris VI (Pierre et Marie Curie), Paris VII (Denis Diderot), Paris Sud-XI, l'École normale supérieure de Paris (Ulm) , l'École Normale Supérieure de Cachan, l'École Polytechnique et le CNAM, valant troisième année du magistère d'informatique de l'École normale supérieure de Lyon, l'université Claude Bernard - Lyon I et l'université Joseph Fourier - Grenoble I.

Ce stage s'est déroulé dans l'équipe *sémantique et inteprétation abstraite* du laboratoire d'informatique de l'École normale supérieure, Paris (LIENS, URA CNRS 1327), sous la direction du professeur Patrick COUSOT, responsable de cette équipe.

Remerciements à Patrick et Radhia Cousot et Arnaud Venet pour leurs conseils et leur patience et à l'équipe Coq et en particulier Christine Paulin pour la *hotline* « un bug de Coq par jour ». Remerciements également aux auteurs de LaTeXet des logiciels et styles annexes[1] , en particulier XYpic, dvips et Ghostscript.

[1] À ce propos, pour les correspondances de Galois, j'ai utilisé le style `galois.sty` de P. Cousot, disponible sur `http://www.dmi.ens.fr/~cousot/Dossier_TeX/galois.sty/`.

Plan

Résumé

Les méthodes d'interprétation abstraite permettent d'obtenir des propriétés sémantiques de programmes informatiques de façon entièrement statique et automatisée. La réalisation d'interpréteurs abstraits est cependant complexe, ce qui peut mettre en doute leur fiabilité. Parallèlement, les techniques de preuves formelles permettent, au prix d'un travail humain important, de produire des programmes certifiés conformes à leur spécification. Nous nous proposons de produire ainsi des interpréteurs abstraits dont la conformité a été montrée de manière formelle.

Partie 1

Problèmes scientifiques

1.1 Analyse de programmes

Afin de produire des logiciels sûrs, on veut montrer formellement qu'il vérifient certaines spécifications. Ainsi, un logiciel de pilotage d'aiguillages de chemin de fer ne devra jamais envoyer deux rames l'une sur l'autre ; un logiciel de pilotage de fusée spatiale ne devra pas tomber en panne lors de son décollage et entraîner sa destruction. Sans aller jusqu'à de tels logiciels critiques, on veut souvent s'assurer que des parties compliquées de logiciels fassent bien ce que l'on attend d'elles.

On peut diviser les techniques formelles mécanisées en trois catégories :

- l'**analyse statique**, où un programme déjà réalisé est soumis à un logiciel qui l'analyse et donne des résultats sur ses exécutions possibles en machine ; notons les techniques
  - d'**interprétation abstraite**, sur des sémantiques approchées
  - de *model-checking* , sur des modèles finis
- la **preuve assistée**, dans laquelle un humain analyse le code du programme avec l'aide d'un logiciel ; souvent, on opère sur une version simplifiée du programme, développée dans un langage fonctionnel ;
- l'**extraction de programme**, qui rejoint en grande partie la méthode précédente, mais où le programme est développé conjointement avec la preuve de son bon fonctionnement - le programme, débarassé des preuves, peut alors être extrait pour être utilisé.

Le problème fondamental est que, bien évidemment, toute propriété sémantique non uniformément vraie ou fausse est indécidable[1] ; dans le premier cas, on obtiendra des propriétés approchées, c'est-à-dire des propriétés vraies, mais non nécessairement exhaustives.

[1] Il s'agit d'un résultat bien connu de calculabilité, le théorème de Rice : dans un système acceptable de programmation [MY78, §3.1, pp. 93–99], tel que les machines de Turing, si φ note la fonction qui au code d'un programme associe sa sémantique dénotationnelle, c'est à dire la fonction récursive partielle qu'il calcule, alors les seules images réciproques par φ qui soient récursives sont l'ensemble de tous les codes sources et l'ensemble vide.

Sans chercher des problèmes d'exposition compliquée, l'analyse du très simple programme suivant est très difficile :

$x :=$ lire_valeur_entière ;
while $x > 1$
    if impair(x)
    then $x := 3x+1$
    else $x := x/2$

La conjecture dite de Syracuse est que ce programme termine toujours, quelle que soit la valeur entrée. Ainsi, un programme d'analyse suffisamment puissant pour trouver ce résultat serait plus puissant que des milliers de mathématiciens compétents !

Toute la difficulté réside alors en l'obtention de propriétés les plus exactes possibles (après tout, le système peut répondre « pas d'information », ce qui est une propriété exacte, mais de peu d'intérêt !). Dans les autres cas, c'est l'ingéniosité de l'analyste-programmeur qui est mise en jeu ; la difficulté réside alors dans l'automatisation du système de preuve afin de rendre la tâche moins fastidieuse.

Notons que ces techniques peuvent se combiner ; ainsi on pourra utiliser des techniques de preuve formelle pour se ramener à un système fini, puis appliquer le *model-checking*. C'est ainsi que les prouveurs de théorèmes se voient intégrés à des *model-checkers* [ORS97, HS96].

1.2 Interprétation abstraite

1.2.1 Introduction

Motivations

L'interprétation abstraite remplace l'usage de la sémantique exacte, qui malheureusement est indécidable, par une sémantique approchée. Cette dernière ne rendra pas compte de toutes les propriétés observables sur la sémantique exacte, mais donnera un calcul en temps fini et si possible d'une complexité raisonnable. Nous ne détaillerons pas trop les résultats mathématiques de cette approche de la sémantique ; le lecteur se réfèrera à la thèse de P. Cousot [Cou78] ou à [CC98, CC92].

Prenons un premier exemple, un système de transitions. Celui-ci est donné par un ensemble, dit **domaine concret**, X, une relation $\rightarrow \in X \times X$ et un ensemble d'états initiaux possibles X_0. Ce que nous voulons, c'est calculer l'ensemble des états accessibles, c'est-à-dire $X_a = \{x \in X \mid \exists x_0 \in X_0 \; x_0 \rightarrow^* x\}$. Or, suivant le système, cet ensemble peut être très difficile, voir impossible à calculer (par exemple, si l'on prend le système de transitions correspondant à la sémantique à petits pas d'un programme...).

Ce que nous voulons calculer est

$$X_a = \bigcup_{i=0}^{\infty} \Psi^i X_0$$

où Ψ est la fonction :

$$\begin{array}{rccl} \Psi : & \wp(X) & \to & \wp(X) \\ & S & \mapsto & \{s' \subseteq X \mid \exists s \in S \; s \to s'\} \end{array}$$

Remarquons que la difficulté vient du fait que l'espace $\wp(X)$ peut être infini. Nous voulons nous ramener à un espace plus simple.

Propriétés à démontrer

Toujours sur cet exemple, nous pouvons nous intéresser à deux catégories de propriétés :

- *liveness properties*, où l'on veut s'assurer qu'à un moment, le système passera par certains états X_t ;
- *safety properties*, où l'on veut s'assurer qu'à aucun moment, le système ne passera par certains états X_t.

Dans le premier cas, nous pourrons conclure à l'aide d'une approximation inférieure de X_a : si $X'_a \subset X_a$ et $X_t \cap X'_a \neq \emptyset$, alors a fortiori $X_t \cap X_a \neq \emptyset$. Si l'approximation est trop imprécise, c'est-à-dire que X'_a est « trop inférieure » à X_a, nous ne pourrons pas conclure. Dans le second cas, nous utiliserons une approximation supérieure de X_a : si $X_a \subset X'_a$ et $X_t \cap X'_a = \emptyset$, alors a fortiori $X_t \cap X_a = \emptyset$. Si l'approximation est trop imprécise, c'est-à-dire que X'_a est « trop supérieure » à X_a, nous ne pourrons pas conclure.

1.2.2 Définitions

Treillis et opérateurs

Nous opérons dans des **treillis complets**, c'est à dire des ensembles munis d'ordres partiels (souvent notés $\sqsubseteq$) tels que tout sous-ensemble A ait une borne supérieure $\sqcup A$ et une borne inférieure $\sqcap A$. Par exemple, $\wp(X)$ muni de l'ordre partiel $\subseteq$ est un treillis complet, dont l'opération de borne supérieure est l'union et l'opération de borne inférieure l'intersection. Remarquons que cela implique que le treillis ait un plus petit élément, souvent noté $\bot$ (infimum ou *bottom*) et un plus grand élément, souvent noté $\top$ (supremum ou *top*).

Une fonction d'un treillis dans un autre, ou opérateur, est **monotone** si c'est un morphisme pour l'ordre ; il est **continu** si l'image de la borne supérieure de toute chaîne croissante (sous-ensemble sur lequel l'ordre est total) est la borne supérieure des images ; il est **ω-continu** si l'image de la borne supérieure de toute suite croissante est la borne supérieure des images. Remarquons les implications suivantes :

morphisme de bornes supérieures

$\Rightarrow$ continu

$\Rightarrow$ ω-continu

$\Rightarrow$ monotone

Nous nous intéresserons souvent aux point fixes de fonctions d'un treillis dans lui-même.

Théorème 1 (Tarski) *L'ensemble des points fixes d'un opérateur monotone est un treillis complet.*

Le théorème suivant a l'avantage d'être calculatoire.

Théorème 2 (Kleene) *Le plus petit point fixe d'un opérateur ω-continu f est*

$$\mathrm{lfp}\, f = \bigsqcup_{i=0}^{\infty} f^i(\bot).$$

Si nous sommes dans un treillis verifiant la **condition de chaîne ascendante**, il est clair que le théorème précédent nous donne un algorithme pour calculer le plus petit point fixe d'un endoopérateur sur un treillis de hauteur finie :

$x := \bot$
while $x < f(x)$ (ou, ce qui est équivalent, $x \neq f(x)$)
    $x := f(x)$
return x

Appellant **hauteur** d'un treillis la borne supérieure dans $\mathbb{N} \cup \{\infty\}$ des longueurs des chaînes strictement croissantes[2] , un treillis de hauteur finie vérifie forcément la condition de chaîne ascendante. Les treillis abstraits utilisés ici seront tous de hauteur finie.

Correspondances de Galois

On appelle **correspondance de Galois** entre deux treillis $X^\flat$ et $X^\sharp$ une paire de fonctions α, dite abstraction, et γ, dite concrétisation, vérifiant

$$\forall x \in X^\flat \forall y \in X^\sharp \ \alpha(x) \sqsubseteq^\sharp y \iff x \sqsubseteq^\flat \gamma(y).$$

Intuitivement, on perd de l'information au cours de l'abstraction. On note une telle correspondance

$$X^\flat \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} X^\sharp.$$

[2]Nous pourrions définir les hauteurs de parties dirigées de cardinal quelconque et obtenir ainsi un ordinal donnant la hauteur. On obtient alors le plus petit point fixe par récurrence transfinie. Cela n'a évidemment aucun intérêt calculatoire.

1.2.3 Approximations (supérieures)

Si nous avons un treillis T, une fonction $f : T \to T$, et un treillis $T^\sharp$ tel que $T \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} T^\sharp$, nous appelons **abstraction** ou **approximation** de f une fonction $f^\sharp$ telle que

$$\forall t \in T \ \alpha(f(t)) \sqsubset^\sharp f^\sharp(\alpha(t))$$

. Si nous avons une égalité $\forall t \in T \ \alpha(f(t)) = f^\sharp(\alpha(t))$ c'est que $f^\sharp$ est une **abstraction exacte**. Si $f^\sharp$ est le minimum des abstractions de f pour l'ordre produit sur $T^\sharp \to T^\sharp$, f est la **meilleure abstraction** [3] et $f^\sharp = \alpha \circ f \circ \gamma$[4].

On définit de même la notion d'abstraction pour les constantes ($\alpha(C) \sqsubset^\sharp C^\sharp$) et les fonctions à plusieurs arguments ($\forall x,y \in T \alpha(f(x,y)) \sqsubset^\sharp f^\sharp(\alpha(x),\alpha(y))$).

1.3 Le logiciel Coq et le calcul des constructions inductives

1.3.1 Le calcul des constructions λC

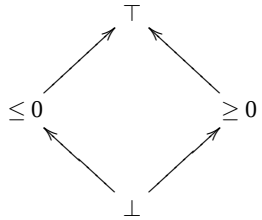
Expressivité

Le Calcul des Constructions (CoC) de G. Huet et T. Coquand est le plus puissant des systèmes de types du λ-calcul pur du cube. Il permet d'exprimer des notions telles que le type des tableaux, paramétré par la nature des éléments à y stocker et le nombre d'élements (on a ainsi une véritable fonction Types $\to \mathbb{N} \to$ Types). Cette souplesse lui donne une forte expressivité lorsqu'à travers d'un isomorphisme de Curry-Howard on l'utilise comme logique d'ordre supérieur (fig. 1.2).

Extraction vers F_ω

L'utilisation du calcul des constructions pour définir à la fois des fonctions et des propositions conduit à un mélange de types « concrets » (par exemple, le type des entiers naturels) et de types propositionnels, dont le contenu est l'ensemble des λ-termes codant des démonstrations de la proposition logique codée par le type.

[3] Il n'existe pas nécessairement de meilleure abstraction si on n'a pas de correspondance de Galois. Par exemple, le treillis



ne donne pas de meilleure approximation de la constante 0.

[4] Il faut bien noter que la formule $\alpha \circ f \circ \gamma$ n'a en général aucun contenu calculatoire (voir 2.2.4).

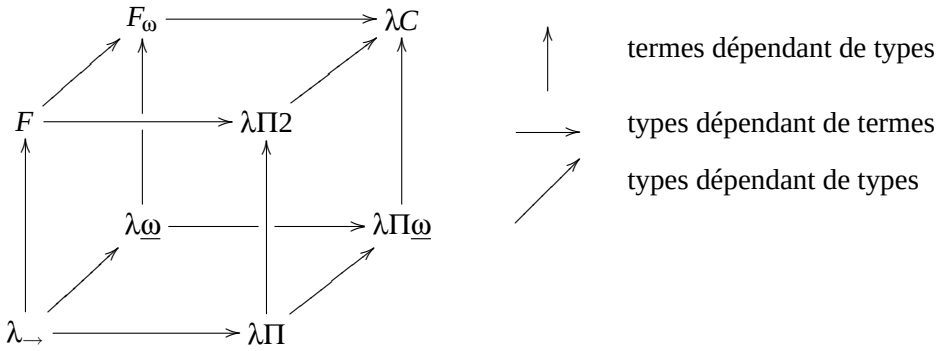


FIG. 1.1: Le cube de Barendregt des systèmes de types purs. Notons le système F de Girard, le système F_ω vers lequel on peut extraire les programmes du calcul des constructions λC.

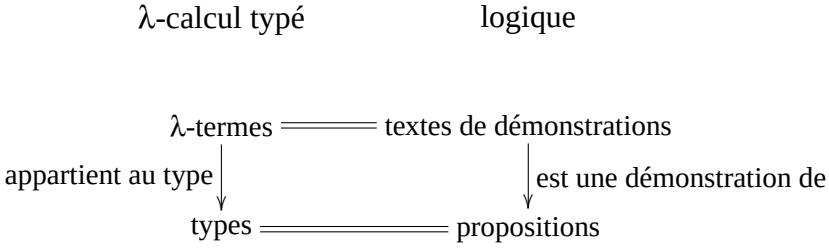


FIG. 1.2: Isomorphisme de Curry-Howard.

Les types ont donc été séparés en deux classes, Set et Prop, avec un typage ad hoc.[5] Tout le contenu calculatoire des programmes est ainsi reporté dans Set ; si l'on veut se contenter d'exécuter, on peut jeter ce qui correspond à Prop dans le terme du programme, ce qui revient à procéder à une extraction [PM89] de son contenu calculatoire.

Les systèmes de type des langages fonctionnels comme ML [Ler] sont encore largement insuffisants pour représenter les termes ainsi obtenus. En particulier, on ne peut pas y paramétrer les types par des termes (en Caml, il existe un type « tableau » polymorphe en le type des éléments, mais non paramétrable par la longueur). On est donc amené à encore extraire, pour obtenir des termes de F_ω. Bien que le système de type de ML, prédicatif, soit insuffisant pour représenter F_ω, impredicatif, on peut s'en contenter ; au besoin, on insèrera des coercions forcées.

1.3.2 Les types inductifs

Le système F est capable de représenter les types inductifs [GLT89]. Cette représentation est malheureusement très lourde, en particulier par les programmes extraits qu'elle produit, inefficaces (par exemple, une définition naïve de la fonction prédécesseur sur les naturels définis par zéro et successeur opérera en temps linéaire en ce naturel, alors qu'on pourrait s'attendre à un temps unitaire). Une représentation interne des types inductifs a donc été ajoutée. On obtient ainsi le calcul des constructions inductives (λC^i). L'extraction donne des programmes de F_ω^i ; en cas d'exportation vers ML, les types inductifs du langage sont utilisés.

Cette représentation permet de définir tous les combinateurs logiques usuels ($\wedge$, $\vee$, $\exists$) comme des types inductifs, dont les constructeurs et les destructeurs correspondent aux opérations d'introduction et d'élimination habituelles.

1.3.3 L'assistant à la preuve Coq

Le logiciel Coq permet l'entrée de spécifications dans le langage Gallina ; les preuves à fournir, fastidieuses dans le cas de preuves logiques, peuvent être construites à partir d'un assistant de preuve.

[5] Remarquons que ce typage séparant Prop de Set peut être vu comme une interprétation abstraite de la $\beta\delta\iota$-réduction, approximant la propriété de n'être pas utilisé dans le résultat final.

Partie 2

Travaux

2.1 Preuves formelles en théorie de l'interprétation abstraite

Afin de pouvoir aborder les problèmes concrets d'analyse de programme, j'ai d'abord réalisé une formalisation de la plupart des aspects théoriques de l'interprétation abstraite.

2.1.1 Modèle de base : le treillis complet

Nous avons pris comme structure mathématique de base le treillis complet ; cette structure, qui malheureusement exclut certains développements, comme les polyhèdres convexes[1] était cependant la plus indiquée pour les exemples envisagés. Le problème est qu'il est malaisé, dans l'état actuel des structures de Coq, d'affaiblir les hypothèses de façon non homogène à travers les définitions (définir une certaine opération pour les ordres partiels, une autre pour les sup-demi-treillis complets, une autre pour les treillis) tout en gardant une facilité d'utilisation suffisante — voir le 3.2. Ainsi, les constructions de cette partie gagneraient pour beaucoup à voir leurs hypothèses affaiblies. De même, nous nous sommes restreints aux correspondances de Galois, alors que des formes plus faibles [2] d'abstractions existent.

Le treillis est donné comme la structure :

```
Record lattice : Type := {
```

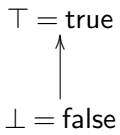
[1] Les polyhèdres convexes d'un espace de dimension d, munis de l'enveloppe convexe de l'union comme borne supérieure et de l'intersection comme borne inférieure, ne forment pas un treillis complet à partir de la dimension 2 ; ainsi, en dimension 2, l'union infinie de la famille indexée par n des polygones réguliers à n côtés de centre O et de rayon R est le disque de centre O et de rayon R au bord près, ce qui ne peut pas être donné par un polygone convexe. En revanche, c'est un treillis.

[2] On peut trouver des abstractions données uniquement par leur fonction de concrétisation. Ainsi, pour reprendre le même exemple que précedemment, les polyhèdres convexes sont une abstraction des parties de l'espace et se concrétisent en leur volume. En revanche, il n'y a pas de correspondance de Galois, parce qu'il n'y a pas de « meilleure abstraction » : un disque peut ainsi être approximé par excès de plus en plus finement par des polygones, mais l'intersection de ces polygones d'approximation va donner, modulo le bord, le disque lui-même, ce qui n'est pas un polygone.

```
  lat_T                          :> Set;
  lat_R                          : (relation lat_T);
  lat_R_order                    : (order lat_T lat_R);
  lat_least_upper_bound          : (Ensemble lat_T)->lat_T;
  lat_least_upper_bound_ok       : (pe: (Ensemble lat_T))
    (is_least_upper_bound lat_T lat_R
      (lat_least_upper_bound pe) pe);
  lat_greatest_lower_bound       : (Ensemble lat_T)->lat_T;
  lat_greatest_lower_bound_ok    : (pe: (Ensemble lat_T))
    (is_greatest_lower_bound lat_T lat_R
      (lat_greatest_lower_bound pe) pe)
}.
```

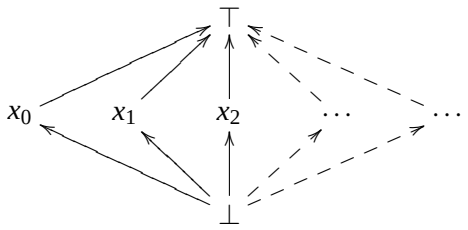
2.1.2 Treillis classiques de base

Booléens



Ce treillis ne pose pas de difficulté, à part que le fait qu'il soit complet ne puisse pas être prouvé en logique intuitionniste (voir 2.2.2). Nous introduirons donc les opérations de bornes inférieure et supérieure de sous-ensembles définis par des prédicats arbitraires comme des axiomes.

Treillis plats



2.1.3 Treillis de fonctions

Toutes les constructions ci-dessous sont prouvées entièrement formellement.

Définition

Pour X un ensemble et T un treillis, on se donne l'ordre $\leq_{(X \to T)}$:

$$f \leq_{(X \to T)} g \iff \forall x \in X\, (f\ x) \leq_T (g\ x)$$

Alors $\sqcup_{(X\to T)}S = \lambda x \in X.\sqcup_T \{(f\ x) \mid f \in S$ et $\sqcap_{(X\to T)}S = \lambda x \in X.\sqcap_T \{(f\ x) \mid f \in S$.
Pour $T = \mathbb{B}$, cela donne en particulier le **treillis des parties** de l'ensemble X vu au 1.2.2.

Abstractions classiques

Pour X et Y deux ensembles et T un treillis

1. $$((X \to Y) \to T) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (X \to (Y \to T))$$

   avec

   $$\begin{cases} \alpha = & \lambda P \in ((X \to Y) \to T).\lambda x \in X.\lambda y \in Y. \\ & \sqcup_T \{t \in T \mid \exists f \in (X \to Y)\,(P\ f) = t \wedge (f\ x) = y\} \\ \gamma = & \lambda f \in (X \to (Y \to T)).\lambda g \in (X \to Y) \\ & \sqcap_T \{t \in T \mid \exists x \in X\,(f\ x\ (g\ x)) = t\}. \end{cases}$$

   Pour $T = \mathbb{B}$ cela donne en particulier

   $$\wp((X \to Y)) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (X \to \wp(Y)).$$

2. Pour X un ensemble et T_1 et T_2 deux treillis tels que

   $$T_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} T_2,$$

   on a

   $$(X \to T_1) \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} (X \to T_2)$$

   avec $\alpha' = \lambda a \in (X \to T_1).\alpha \circ a$ et $\gamma' = \lambda b \in (X \to T_2).\gamma \circ b$.
3. Pour X et Y deux ensembles, $f : (X \to Y)$, T un treillis, on a

   $$(X \to T) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Y \to T)$$

   avec $\alpha = \lambda a \in (X \to T).\lambda y \in Y.\sqcup_T \{t \in T \mid \exists x \in X\,(f\ x) = y \wedge (a\ x) = t\}$ et $\gamma = \lambda b \in (Y \to T).b \circ f$. Pour $T = \mathbb{B}$ cela donne en particulier

   $$\wp(X) \underset{f}{\overset{f^{-1}}{\leftrightarrows}} \wp(Y),$$

   avec les fonctions image directe et réciproque.
4. Pour X un ensemble, $x_0 \in X$, on a

   $$(X \to T) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} T$$

   avec

   $$\begin{cases} \alpha = \lambda a \in (X \to T).\,(a\ x_0) \\ \gamma = \lambda t \in T.\lambda x \in X.\,[\text{si } x = x_0 \text{ alors } \top \text{ sinon } t]. \end{cases}$$

5. Pour X et Y deux ensembles, $f : (X \to Y)$, on a

$$(Y \to T) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (X \to T)$$

avec

$$\begin{cases} \alpha = \lambda a \in (Y \to T).a \circ f \\ \gamma = \lambda b \in (X \to T).\lambda y \in Y.\sqcap_T \{t \in T \mid \exists x \in X\,(f\ x) = y \wedge (b\ x) = t\}. \end{cases}$$

Pour $T = \mathbb{B}$ et f une injection, cela donne l'abstraction des fonctions sur leur restriction à une partie de leur domaine.

2.1.4 Treillis numériques

Treillis des signes

Nous avons implémenté comme abstraction numérique le treillis des signes donné par

$$\wp(\mathbb{Z}) \underset{\text{sgn}}{\overset{\text{sgn}^{-1}}{\leftrightarrows\!\!\!\!\!\rightarrow}} \wp(\{-,0,+\})$$

où $\text{sgn}(x)$ est la fonction « signe » associant à un entier relatif son signe ($-$, 0 ou $+$).

Treillis des intervalles

Nous n'avons pu, faute de temps, implémenter le treillis des intervalles

$$\wp(\mathbb{Z}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows\!\!\!\!\!\rightarrow}} I_{\mathbb{Z}} = \{\bot_{I_{\mathbb{Z}}}\} \cup \{\langle x,y\rangle \mid x \in \mathbb{Z} \cup \{-\infty\} \wedge y \in \mathbb{Z} \cup \{+\infty\} \wedge x \leq y\}$$

où

$$\begin{cases} \gamma(\langle x,y\rangle) & = & [x,y] \cap \mathbb{Z} \\ \gamma(\bot_{I_{\mathbb{Z}}}) & = & \emptyset \\ \alpha(\emptyset) & = & \bot_{I_{\mathbb{Z}}} \\ \alpha(P) & = & \langle \inf P, \sup P\rangle \quad P \neq \emptyset \end{cases}$$

2.2 Constructivité

2.2.1 Indécidabilités

Voici deux exemples d'abstractions, la première étant extrèmement courante, où la meilleure approximation des constantes est indécidable.

Projection

Pour la correspondance $\wp(\mathbb{N}\times\mathbb{N}) \underset{\pi_1^2}{\overset{\pi_1^{2-1}}{\leftrightarrows}} \wp(\mathbb{N})$ où $\pi_1^2\langle x,y\rangle = x$, étant donné une partie récursive X de $\mathbb{N}\times\mathbb{N}$, $\pi_1^2(X)$ n'est en général pas récursive [3] ;autrement dit, on n'arrivera pas à associer de façon calculatoire à une partie de $\mathbb{N}\times\mathbb{N}$ sa première projection ! Lors de l'implémentation, cela correspondait à l'usage d'axiomes non réalisables (voir 2.2.2).

Abstraction vers un ensemble fini

Il est évident que le problème abstrait est indécidable en général si l'on ne place pas de conditions sur les opérateurs ou l'abstraction considérés. Nous voyons ici que si la fonction d'abstraction est non effective, il peut être impossible de donner effectivement des versions abstraites exactes des constantes du programme.

Soit W un ensemble non récursif[4]. Pour la correspondance $\wp(\mathbb{N}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{B}$ où

$$\alpha(X) = [\text{si } W\cap X \neq \emptyset \text{ alors } \top \text{ sinon } \bot]\,,$$

la meilleure approximation des constantes n'est pas calculable : calculer la meilleure approximation de n'importe quel singleton $\{x\}$ revient à décider $W(x)$.

Ainsi, on peut très bien avoir une abstraction vers un domaine fini où la meilleure approximation des constantes est indécidable !

Par contre, il faut noter que l'on a toujours une approximation supérieure, $\top$; tout le problème consiste à se placer entre celle-ci, qui n'a aucun intérêt, et la meilleure, éventuellement non calculable, et le plus près possible de cette dernière.

2.2.2 Ramener l'indécidabilité dans des axiomes

Pouvoir calculer $\sqcup_{\mathbb{B}}\{b\in\mathbb{B} \mid P(b)\}$ implique de pouvoir décider effectivement toute formule du système (calculer $\sqcup_{\mathbb{B}}\{b\in\mathbb{B} \mid P\wedge b=\top\}$), ce qui s'exprimerait en Coq par

```
(P: Prop) {P}+{~P}
```

C'est bien entendu un axiome très fort, qui implique d'ailleurs l'axiome classique $\forall P : \text{Prop}\ P\vee\neg P$.

Cependant, nous avions besoin de déclarer le fait que $\mathbb{B}$ est un treillis complet. Nous avons donc rajouté les opérations $P\mapsto\sqcup_{\mathbb{B}}\{b\in\mathbb{B} \mid P(b)\}$ et $P\mapsto\sqcap_{\mathbb{B}}\{b\in\mathbb{B} \mid P(b)\}$ comme des axiomes de la théorie. Ces axiomes étant non réalisables, ils ne devront pas être présents dans les programmes à exporter.

[3] Il suffit pour s'en convaincre de considérer l'ensemble des couples d'entiers $X=\langle x,y\rangle$ où y est le code d'une preuve de la formule x pour un codage de Gödel des formules et des preuves de l'arithmétique de Péano, lequel est un ensemble récursif dont la première projection est l'ensemble des formules démontrables de l'arithmétique de Péano, lequel est non récursif.

[4] On prendra par exemple $W=\{\langle x,x\rangle \mid \varphi_u\langle x,x\rangle\neq\bot\}$ avec φ_u la fonction universelle d'un système acceptable de programmation, les machines de Turing par exemple. W est récursivement énumérable mais pas récursif.

2.2.3 La constructivité n'est pas évidente

Nous l'avons vu, le théorème de Kleene (2) nous fournit, dans les treillis vérifiant la condition de chaîne ascendante, un moyen simplement implémentable de calculer le point fixe de toute fonction monotone... Implémentable ? Nous faisons appel à deux opérations sur le treillis : l'infimum (ce qui ne pose en général pas de problème — toutes les constructions que nous faisons sur nos treillis de base donnent des infima d'expression très simple) et la comparaison d'éléments. Or, cette dernière n'est pas une opération anodine, elle peut même être très vite indécidable — ainsi la comparaison de deux fonctions $\mathbb{N} \to \mathbb{N}$ récursives l'est.

Il est ainsi fourni une définition constructive de l'égalité de deux fonctions $f : X \to Y$ avec Y un ensemble quelconque et X un ensemble fini ; celle-ci est définie par récurrence sur le cardinal de X. S'il y a besoin de prédicats d'égalité sur des ensembles arbitraires, on peut les introduire comme axiomes. Il faudra alors éviter la présence de ceux-ci dans le code exécutable.

La nécessité de ne garder que des termes réalisables dans le code à exporter a transparu dans les structures utilisées : nous avons ainsi défini la classe `clattice` des treillis complets calculatoires, dont les opérations d'infimum et de bornes supérieure et inférieure de paire d'éléments sont effectives ; des champs de cette structure établissent l'égalité entre ces éléments et leur définition en termes de bornes supérieures et inférieures de parties définies par des prédicats :

```
Record clattice : Type :=
  { clat_lattice :> lattice;
    clat_join: clat_lattice->clat_lattice->clat_lattice;
    clat_join_ok: (x,y: clat_lattice) (clat_join x y)=
      (lat_least_upper_bound clat_lattice
        (epair clat_lattice x y));
    clat_meet: clat_lattice->clat_lattice->clat_lattice;
    clat_meet_ok: (x,y: clat_lattice) (clat_meet x y)=
      (lat_greatest_lower_bound clat_lattice
        (epair clat_lattice x y));
    clat_bottom: clat_lattice;
    clat_bottom_ok: clat_bottom = (lat_bottom clat_lattice);
    clat_equal: (x,y: clat_lattice) {x=y}+{~ x=y}
}.
```

2.2.4 Égalité mathématique et contenu calculatoire

Lors des développements mathématiques effectués, il est apparu très vite la nécessité de bien faire apparaître des distinctions basées sur la calculabilité de termes mathématiquement égaux. Prenons un exemple : si $x, y \in \mathbb{B}$,

$$x \sqcup y = \bigsqcup \{t \in \mathbb{B} \mid t = x \vee t = y\}.$$

Or, le deuxième terme fait appel à un axiome non constructif, la prise de borne supérieure sur un sous-ensemble arbitraire de $\mathbb{B}$ défini par un prédicat, tandis que le premier est aisément définissable par distinction de cas.

De même, dans le cas d'une correspondance de Galois $T \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} T^\sharp$, d'une fonction $f : T \to T$, on peut prendre $f^\sharp : T^\sharp \to T^\sharp = \alpha \circ f \circ \gamma$, qui est la meilleure approximation de f. Malheureusement, en général, cela n'a pas de contenu calculatoire, α et γ pouvant très bien ne pas en avoir, par exemple en contenant des prises de bornes supérieures et inférieures par prédicats (exemples au 2.1.3).

Le point crucial est que des termes non $\beta\delta\iota$-équivalents peuvent avoir la même sémantique informelle alors que certains représentent des programmes et d'autres non. Il convient donc de faire très attention à établir l'interpréteur abstrait à partir de formules effectives — les plus simples possibles si l'on veut un code extrait efficace. Si l'interpréteur est effectivement égal à $\alpha \circ f \circ \gamma$, on doit montrer cette égalité.

2.2.5 Indécidabilité de formules

Nous sommes souvent confrontés à des preuves de théorèmes du premier ordre de la forme $A \Rightarrow B$, avec dans A et B comme seuls connecteur la conjonction, comme seul quantificateur le quantificateur universel, comme seul prédicat le prédicat d'ordre, et en admettant des symboles de fonctions sur une signature Σ.

Il serait donc tentant d'avoir une procédure de décision qui, en fonction d'une formule F de cette forme, dise si $(X, (f_i)_{i\in\Sigma}, \sqsubseteq) \Vdash F$ pour tout ensemble ordonné $(X, \sqsubseteq)$ et affectations des fonctions. Nous montrons que cela est malheureusement impossible.

Considérons une signature Σ et l'algèbre initiale I_Σ, ainsi qu'une théorie égalitaire E sur Σ (c'est à dire une conjonction de formules universellement quantifiées dont les fonctions sont dans la signature Σ) [Lal90, ch. 6], [JD90].

Toute formule F vraie sur I_Σ sera vraie sur tous les autres modèles de E. Remplaçons dans F toutes les égalités $x = y$ par des conjonctions $x \sqsubseteq y \wedge y \sqsubseteq x$ pour obtenir F'. La formule F' sera alors vraie sur tous les modèles $(X, (f_i)_{i\in\Sigma}, \sqsubseteq)$, puisque dans chacun de ces modèles $\forall x \forall y\, x = y \Leftrightarrow x \sqsubseteq y \wedge y \sqsubseteq x$.

Supposions que nous disposions d'une telle procédure de décision P. Si $\models F$ nous aurions alors $P(F') = \mathsf{true}$. Réciproquement, si $\not\models F$, comme $=_E$ est un ordre partiel, il existera au moins un modèle sur lequel F' sera fausse.

Nous obtenons ainsi une procédure de décision des théories équationnelles. Or cela est absurde ; il suffit de considérer la théorie des combinateurs S-K-I [Lal90], laquelle est indécidable.

2.3 Extraction d'interpréteurs abstraits

Mon choix de l'assistant Coq était entre autres guidé par le coté pratique qu'il y a à avoir une possibilité d'exportation du code spécifié vers un langage compilable. Ainsi, ici le code généré peut être exporté vers Objective Caml.

En raison de problèmes dans la procédure d'extraction de Coq, et notamment de son incapacité à $\beta\delta$-réduire certains fragments non typables pour Caml (plus par-

ticulièrement la fonction qui à un treillis associe son ensemble sous-jacent), nous n'avons pu extraire qu'un exemple relativement petit (les expressions régulières).

2.3.1 Points fixes d'opérateurs monotones

Un des points marquants de cette possibilité et de pouvoir faire intervenir dans la définition du programme des propriétés indispensables à la bonne définition de ce programme, mais sans contenu calculatoire ; les termes correspondant aux preuves de ces propriétés seront éliminés à l'extraction.

La procédure de calcul du plus petit point fixe d'un opérateur monotone dans un treillis vérifiant la condition de chaîne ascendante est de type :

```
iterate_fixpoint
     : (T:wflattice)
        (f:(T)->(T))
         (is_monotonic T (lat_R T) T (lat_R T) f)
          ->{nx:(nat*(T))|
               (omega_iterate T f (clat_bottom T) (Fst nx))=(Snd nx)
               & (f (Snd nx))=(Snd nx)}
```

(elle calcule en parallèle le nombre d'itérations). Cette fonction est basée sur la fonction

```
Acc_rec
     : (A:Set)
        (R:A->A->Prop)
         (P:A->Set)
          ((x:A)
            ((y:A)(R y x)->(Acc A R y))-
>((y:A)(R y x)->(P y))->(P x))
           ->(x:A)(Acc A R x)->(P x)
```

de récurrence bien fondée sur la définition inductive définition de la condition de chaîne ascendante :

```
Inductive  Acc : A -> Prop
  := Acc_intro : (x:A)((y:A)(R y x)->(Acc y))-
>(Acc x).
```

L'exportation de cette fonction produit

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

ce qui est un combinateur de point fixe universel !

Ainsi, la procédure d'extraction a jeté les paramètres de monotonie et de condition de chaîne croissante pour produire un sous-programme très simple.

2.3.2 Test d'égalité

Un autre exemple d'extraction est le test d'égalité. On lui donne un type `(x,y: T) {x=y}+{~x=y}`, ce qui veut dire que c'est une fonction à deux paramètres dans T qui retourne effectivement une valeur dans type à deux constructeurs, en fournissant dans le premier cas une preuve de $x = y$ et dans le deuxième une preuve de $x \neq y$. Les fonctions de test extraites retournent alors une valeur d'un type à deux éléments `left` et `right`, de même qu'habituellement on retourne `true` et `false`.

2.4 Exemples proposés

Les développements présentés ici et la théorie associée, formalisés sont Coq, font environ 10000 lignes de code.

Notre travail a été ralenti par la découverte de bugs dans Coq ; nous remercions d'ailleurs l'équipe de développement de ce logiciel d'avoir eu la patience de lire nos récriminations !

2.4.1 Expressions régulières

Soit un alphabet Σ et

$$\begin{array}{rcccl} \pi_1^l & : & \Sigma^* & \to & \Sigma \cup \{\varepsilon\} \\ & & h.t & \mapsto & h \\ & & \varepsilon & \mapsto & \varepsilon, \end{array}$$

Σ^* étant le monoïde des mots sur cet alphabet et ε le mot vide. On a la correspondance de Galois $\wp(\Sigma^*) \underset{\pi_1^l}{\overset{\pi_1^{l\,-1}}{\leftrightarrows}} \wp(\Sigma \cup \{\varepsilon\})$. Les expressions régulières sur Σ sont les objets R générés par la grammaire algébrique

$$\begin{array}{rcl} R ::= & \Sigma & \\ \mid & R.R & \\ \mid & R^* & \\ \mid & R \mid R & \end{array}$$

et leur sémantique est définie par induction structurelle :

$$\begin{array}{lcl} [\![x]\!] & = & \{x\} \\ [\![R_1.R_2]\!] & = & [\![R_1]\!].[\![R_2]\!] \\ [\![R^*]\!] & = & \mathrm{lfp} X \mapsto \{\varepsilon\} \cup [\![R]\!].X \\ [\![R_1 \mid R_2]\!] & = & [\![R_1]\!] \cup [\![R_2]\!]. \end{array}$$

Nous montrons formellement sous Coq que

$$\begin{array}{lcl} [\![x]\!]^\sharp & = & \{x\} \\ [\![R_1.R_2]\!]^\sharp & = & \left[\text{si } [\![R_2]\!]^\sharp \neq \emptyset \text{ alors } [\![R_1]\!]^\sharp \text{ sinon } \emptyset\right] \cup \left[\text{si } \varepsilon \in [\![R_1]\!]^\sharp \text{ alors } [\![R_2]\!]^\sharp \text{ sinon } \emptyset\right] \\ [\![R^*]\!]^\sharp & = & [\![R]\!]^\sharp \cup \left[\text{si } [\![R]\!]^\sharp \neq \emptyset \text{ alors } \{\varepsilon\} \text{ sinon } \emptyset\right] \\ [\![R_1 \mid R_2]\!]^\sharp & = & [\![R_1]\!]^\sharp \cup [\![R_2]\!]^\sharp. \end{array}$$

est une interprétation abstraite exacte ($\alpha([\![R]\!]) = [\![R]\!]^\sharp$).

Malgré sa simplicité pour un humain, la preuve de correction de cette abstraction a été très laborieuse, en raison notamment d'un grand nombre d'analyse de cas sur des têtes de listes. Notons que le prouveur a été néanmoins utile : ayant omis par mégarde le test $[\![R_2]\!]^\sharp \neq \emptyset$ dans une première formulation de la version abstraite, nous nous en sommes aperçu en traitant un sous-cas de la preuve de correction.

Nous avons exécuté le code extrait (appendice A.1).

2.4.2 Langage impératif simple

Nous considérons un langage impératif dont les variables sont entières, les constantes les entiers relatifs, les opérateurs arithmétiques l'addition, la soustraction, la multiplication entières, les opérateurs de comparaison entiers, les opérateurs booléens et, ou, non et les constructions de contrôle de flot le test et la boucle tant-que.

La sémantique dénotationnelle d'un programme écrit dans un tel langage peut s'exprimer comme le plus petit point fixe sur le domaine $\wp(L \times \mathbb{Z}_\Omega^n)$, où L est l'ensemble (fini) des points de programme, d'un système d'équations $L \to E$ obtenu par transformation syntaxique du programme. $\mathbb{Z}_\Omega$ est l'ensemble des entiers relatifs étendu par la constante Ω signifiant « donnée non initialisée ».

E est un ensemble de termes défini algébriquement :

$$\begin{array}{rll} E ::= & e_\perp & \\ | & e_0 & \\ | & \langle l \rangle & l \in L \\ | & x := A\langle l \rangle & x \in V,\ a \in A,\ l \in L \\ | & !B\langle l \rangle & b \in A,\ l \in L \\ | & E_1 \mid E_2 & \end{array}$$

avec V l'ensemble (fini) des variables, A l'ensemble des expressions arithmétiques

$$\begin{array}{rll} A ::= & z & z \in \mathbb{Z}_\Omega \\ | & v & v \in V \\ | & \mathsf{random} & \\ | & A_1 + A_2 & \\ | & A_1 - A_2 & \\ | & A_1 \times A_2 & \end{array}$$

et B l'ensemble des expressions booléennes

$$\begin{array}{rll} B ::= & \mathsf{True} & \\ | & \mathsf{False} & \\ | & x_1 < x_2 & x_1, x_2 \in V \\ | & x_1 = x_2 & x_1, x_2 \in V \\ | & B_1 \wedge B_2 & \\ | & B_1 \vee B_2. & \end{array}$$

Nous considérons le treillis $S = \wp(V \to \mathbb{Z}_\Omega)$ des environnements de stockage. La sémantique d'un programme $P \in L \to E$ est alors

$$[\![P]\!] = \mathrm{lfp}^{\sqsubseteq_{(L \to S)}} \lambda i : (L \to S) \lambda l : L\ [\![P.l]\!]_E.i$$

où $[\![e]\!]_E : ((L \to S) \to S)$ est définie par

$$\begin{array}{lcl}
[\![e_\perp]\!]_E.i & = & \perp_S \\
[\![e_0]\!]_E.i & = & \{\lambda v\, \Omega\} \\
[\![\langle l \rangle]\!]_E.i & = & i.l \\
[\![x := a\langle l \rangle]\!]_E.i & = & (i.l)\{x \leftarrow [\![a]\!]_A.(i.l)\} \\
[\![!b\langle l \rangle]\!]_E.i & = & [\![b]\!]_B \sqcap_S (i.l) \\
[\![e_1 \mid e_2]\!]_E.i & = & [\![e_1]\!]_E.i \sqcup_S [\![e_2]\!]_E.i
\end{array}$$

avec $[\![b]\!]_B : S$ définie par

$$\begin{array}{lcl}
[\![\mathsf{True}]\!]_B & = & \top_{(V \to \mathbb{Z}_\Omega)} \\
[\![\mathsf{False}]\!]_B & = & \perp_{(V \to \mathbb{Z}_\Omega)} \\
[\![x_1 < x_2]\!]_B & = & \{s \in (V \to \mathbb{Z}_\Omega) \mid s.x_1 = \Omega \vee s.x_2 = \Omega \vee (s.x_1 \in \mathbb{Z} \wedge s.x_2 \in \mathbb{Z} \wedge s.x_1 < s.x_2)\} \\
[\![x_1 = x_2]\!]_B & = & \{s \in (V \to \mathbb{Z}_\Omega) \mid s.x_1 = \Omega \vee s.x_2 = \Omega \vee (s.x_1 \in \mathbb{Z} \wedge s.x_2 \in \mathbb{Z} \wedge s.x_1 = s.x_2)\} \\
[\![b_1 \wedge b_2]\!]_B & = & [\![b_1]\!]_B \sqcap_S [\![b_2]\!]_B \\
[\![b_1 \vee b_2]\!]_B & = & [\![b_1]\!]_B \sqcup_S [\![b_2]\!]_B
\end{array}$$

et $[\![a]\!]_A : (S \to \wp(\mathbb{Z}_\Omega))$ définie par

$$\begin{array}{lcl}
[\![z]\!]_A.s & = & \{z\} \\
[\![v]\!]_A.s & = & s.v \\
[\![\mathsf{random}]\!]_A.s & = & \mathbb{Z} \\
[\![a_1 + a_2]\!]_A.s & = & [\![a_1]\!]_A.s + [\![a_2]\!]_A.s \\
[\![a_1 - a_2]\!]_A.s & = & [\![a_1]\!]_A.s - [\![a_2]\!]_A.s \\
[\![a_1 \times a_2]\!]_A.s & = & [\![a_1]\!]_A.s \times [\![a_2]\!]_A.s.
\end{array}$$

Nous opérons une première abstraction, l'analyse non relationnelle : on « oublie » les relations entre variables par

$$S = \wp(V \to \mathbb{Z}_\Omega) \underset{\alpha_V}{\overset{\gamma_V}{\leftrightarrows}} (V \to \wp(\mathbb{Z}_\Omega)) = S^\sharp$$

où $\alpha_V(P).v = \pi^V_v(P)$ et $\gamma_V(p) = \prod_v f.v$ (voir 2.1.3, 1).

Étant donnée une abstraction numérique etendue en Ω $\wp(\mathbb{Z}_\Omega) \underset{\alpha_{\mathbb{Z}_\Omega}}{\overset{\gamma_{\mathbb{Z}_\Omega}}{\leftrightarrows}} T_{\mathbb{Z}_\Omega}$, nous opérons une deuxième abstraction point à point (voir 2.1.3, 2).

A partir d'une abstraction numérique $\wp(\mathbb{Z}) \underset{\alpha_\mathbb{Z}}{\overset{\gamma_\mathbb{Z}}{\leftrightarrows}} T_\mathbb{Z}$ on construit une abstraction numérique étendue en Ω en faisant l'abstraction produit

$$\wp(\mathbb{Z}_\Omega) \simeq \wp(\mathbb{Z}) \times \mathbb{B} \underset{\lambda\langle x,y\rangle.\langle \alpha_\mathbb{Z}(x), y\rangle}{\overset{\lambda\langle x^\sharp, y^\sharp\rangle.\langle \gamma_\mathbb{Z}(x^\sharp), y^\sharp\rangle}{\leftrightarrows}} T \times \mathbb{B}.$$

Partie 3

Perspectives

3.1 Nécessité de tactiques adaptées

La preuve formelle de théorème nécessite un certain degré d'automatisation pour être efficace. Des tactiques plus évoluées semblent être nécessaires pour développer efficacement des preuves mécaniques d'interpréteurs abstraits.

Nous n'avons malheureusement pas pu beaucoup expérimenter en raison du manque de documentation sur l'architecture interne de Coq v6.1 et de l'arrivée tardive de Coq v6.2.

3.1.1 Tactiques d'équivalence de formalisme

Une part importante des preuves faites en pratique dans un système comme Coq passe dans le travail sur des formulations inutilement complexes équivalentes à des formes plus habituelles pour nos yeux. Par exemple, si l'on a $f : X \to Y$ et $g : Y \to Z$ et $A \subset X$, on a

$$g \circ f(A) = \{z \in Z \mid \exists x \in X \; z = g(f(x))\}$$

mais les calculs menés par le système peuvent aboutir à

$$\{z \in Z \mid \exists y \in Y \; z = g(y) \wedge \exists x \in X \; y = f(x)\},$$

ce qui est bien entendu équivalent mais demande de faire une petite preuve annexe de simplification (on fera par exemple usage de l'axiome d'extensionnalité des parties, qui est que

$$\forall z \in X P(z) \Rightarrow P'(z) \wedge \forall z \in X P'(z) \Rightarrow P(z) \Longrightarrow \{z \in Z \mid P(z)\} = \{z \in Z \mid P'(z)\}.$$

Sans même parler d'égalités d'ensembles, des formulations équivalentes de propositions peuvent entraîner des lourdeurs ; ainsi,

$$\forall x \in X \; P(x) \wedge Q(x)$$

est trivialement équivalent à

$$\forall x \in X\ P(x) \wedge \forall x \in X\ Q(x),$$

mais dans certaines circonstances la deuxième formulation est moins pratique. Dans le même ordre d'idées[1],

$$\forall x \in X\ \exists y \in Y\ P(x,y)$$

est équivalent à

$$\exists f : X \to Y\ \forall x \in X\ P(x, f(x)).$$

Il semble donc souhaitable d'avoir, pour une meilleure efficacité, un système qui « voit » les équivalences de ces différentes formes. On notera que la tactique `Intuition` « normalise » les hypothèses du contexte courant vers de telles formes. On pourrait envisager l'extension de ces possibilités.

3.1.2 Ordres

Une des principales difficultés rencontrées lors de ce stage a été la lourdeur des preuves sur les ordres. Ce qui nous paraît trivial est en effet difficile à faire pour le prouveur. De même qu'une tactique de décision de l'arithmétique de Presburger existe [BBC+98, ch 17, pp 249–252],il serait souhaitable de développer une tactique spécialisée en théorie des treillis.

Il faut cependant noter qu'une telle tactique ne pourrait être trop universelle (voir 2.2.5). Il faudra donc préciser un domaine de validité ou se contenter d'une procédure de semi-décision tronquée en temps, par exemple.

3.1.3 Métathéorie

Dans les mathématiques courantes, il nous arrive de faire des raisonnements métathéoriques, comme « si cette preuve était correcte, elle s'appliquerait sur cette autre structure où la propriété à démontrer est fausse, donc cette preuve est incorrecte ». Sur les cas traités, nous sommes tentés de justifier par exemple qu'une sémantique est ω-continue en disant « c'est une combinaison d'opérateurs qui sont tous ω-continus, donc elle est ω-continue ». Si cela est en partie possible à l'intérieur du système (prouver un théorème « la composée de deux fonctions ω-continues est ω-continue »), une telle approche ne résout pas le problème en pratique. Les raisons sont de deux ordres :

1. même si l'expression écrite est égale à une composée d'opérateurs, elle n'est pas forcément sous cette forme ;

[1] Il s'agit en fait du même exemple que précedemment, mais avec un type $\exists$ qui est un produit dépendant, alors que $\wedge$ est un produit non dépendant.

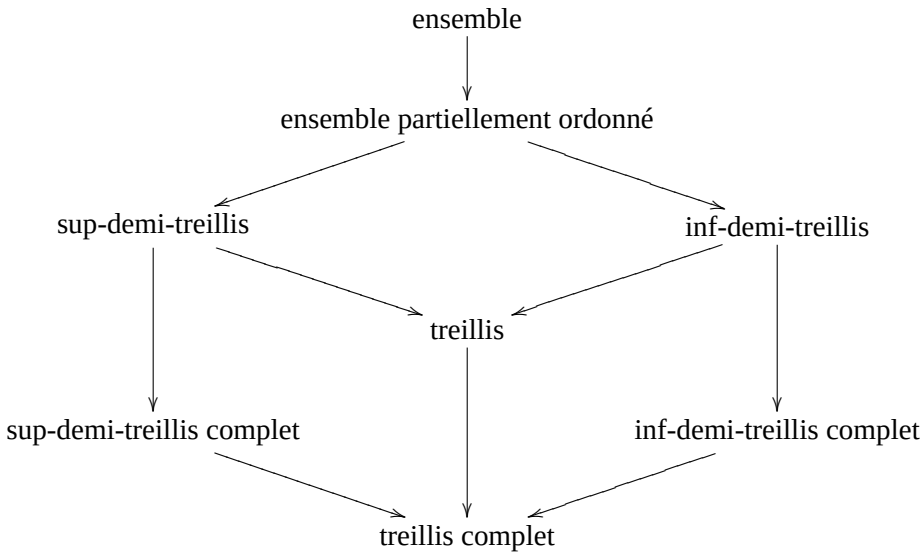


FIG. 3.1: Le treillis des structures utilisées (on n'a pas figuré toutes les combinaisons possibles). On peut encore rajouter des structures plus calculatoires, comme des treillis complets munis d'une fonction effective de comparaison.

2. on peut avoir à considérer des structures sur lesquelles les démonstrations ne sont pas internalisables, par exemple les définitions inductives ; on ne peut pas écrire de théorème Coq disant « pour tout type inductif X, pour toute définition par cas sur X où tous les cas sont ω-continus en l'argument, alors la fonction est ω-continue », ce théorème est uniquement métathéorique.

Il semble donc souhaitable d'avoir des tactiques permettant de régler ces cas.

3.2 Difficultés dans l'élaboration des théories

Nous manipulons des structures mathématiques de plus en plus étoffées (fig. 3.1). Ces structures font l'objet de théorèmes géneriques, pour lesquels il est pratique de nommber la structure complète sous un seul nom ; ainsi, en mathématiques, nous disons « soit $(X, \sqsubset, \sqcup)$ un sup-demi-treillis », voire « soit X un sup-demi-treillis » à la place de « soit X un ensemble muni d'une relation $\sqsubset$ telle que... ». Cela est d'autant plus flagrant dans les transformations des structures ; ainsi, on peut définir une sorte de fonction qui à un ensemble X et un treillis T associe le treillis T^X.

Cette approche structurelle permet de simplifier considérablement l'expression des théorèmes, particulièrement en donnant des hypothèses courtes et lisibles. Elle semble donc indispensable à des développements mathématiques conséquents comme ceux que nous avons commencé de faire.

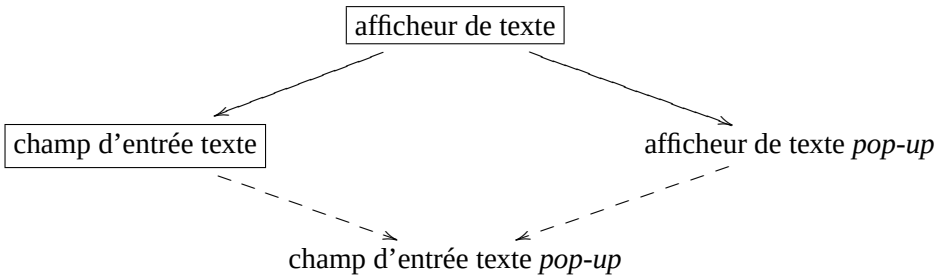


FIG. 3.2: Exemple des difficultés de l'héritage sur un problème concret de génie logiciel : les interfaces graphiques. Si on a une bibliothèque fournissant les types d'objets encadrés, et qu'on l'étend en étendant l'afficheur de texte par une possibilité de *pop-up*, on peut difficilement déclarer le champ d'entrée texte *pop-up* comme héritant de l'afficheur de texte *pop-up* et du champ d'entrée texte. Ce genre de problème se retrouve dans le développement formel de théories mathématiques.

Une première approche de ce problème, que nous avons utilisée, est de définir des structures imbriquées avec des coercions implicites entre elles [BBC+98, p. 229]. Ainsi, une structure *B* étendant une structure *A* contient un champ correspondant à la structure *A* ; on peut aussi voir cela comme un héritage de *A* par *B*. Malgré quelques difficultés dûes aux complications que ces coercions induisent dans les mécanismes d'unification du prouveur, cette solution est relativement satisfaisante pour un héritage linéaire. En revanche, on a plus de mal à faire des structures qui soient la « borne supérieure » de deux autres : ainsi un treillis complet est la combinaison d'un inf-demi-treillis et d'un sup-demi-treillis. Il est malheureusement impossible de retracer cela à l'aide du mécanisme d'enregistrements à héritage de Coq. Cela est à rapprocher des difficultés que l'on trouve par exemple en C++ lorsque l'on a une structure que l'on a enrichi de deux façons différentes et dont l'on voudrait récupérer une version « plus enrichie que les deux autres » (fig. 3.2).

Le graphe 3.1 nous fait penser à un treillis de sous-typage. Des systèmes de modules avec sous-typage sont disponibles dans des langages comme Standard ML ou Objective Caml [Ler95], et J. Courant propose un système de modules pour les systèmes de types purs, adaptable à Coq [Cou98]. Un tel système pourrait, selon lui, faciliter l'élaboration de théories complexes :

> [...] on voudrait pouvoir disposer de bibliothèques de résultats et de preuves, dans lesquelles on aimerait puiser pour spécifier un problème donné ou effectuer une démonstration, de même qu'en mathématique on fera référence à la définition / au lemme / au corollaire / au théorème *X*, du chapitre *Y*, de l'ouvrage *Z*.

Une telle approche semble ainsi toute indiquée pour faciliter la réalisation de projets certifiés lourds, de même que la programmation modulaire facilite la program-

mation de gros projets « conventionnels ».

3.3 Génération automatique de version abstraite

Le travail d'écriture des opérateurs abstraits est fastidieux ; idéalement, on aimerait pouvoir rentrer la sémantique concrète et l'abstraction et avoir un processus automatique qui produise la sémantique abstraite, voire même une preuve de la correction de celle-ci.

Bensalem *et al.* [BLO98] proposent un système produisant automatiquement des abstractions de systèmes d'états finis. Le cadre théorique est le suivant : étant donné un ensemble d'états concrets Σ, une relation de transition $\xrightarrow{s} \subset \Sigma \times \Sigma$, un ensemble d'états abstraits $\Sigma^\sharp$, et une relation d'abstraction $A \subset \Sigma \times \Sigma^\sharp$ on veut fabriquer une relation $\xrightarrow{s^\sharp}$ de transition abstraite telle que

$$\forall x, y \in \Sigma\ \forall x^\sharp \in \Sigma^\sharp\ x \xrightarrow{s} y \wedge A(x, x^\sharp)\ \Rightarrow\ \exists y^\sharp\ x^\sharp \xrightarrow{s^\sharp} y^\sharp \wedge A(y, y^\sharp)^2.$$

Alors, avec $\sigma_0 \in \Sigma$ et $\sigma_0^\sharp \in \Sigma^\sharp$ tels que $A(\sigma_0, \sigma_0^\sharp)$, notant $\mathcal{A}(s) = \{x \in \Sigma \mid \sigma_0 \xrightarrow{s}{}^* x\}$, $\mathcal{A}(s^\sharp) = \{x \in \Sigma^\sharp \mid \sigma_0^\sharp \xrightarrow{s^\sharp}{}^* x^\sharp\}$ et $A^{-1}(X^\sharp) = \{x \in \Sigma \mid \exists x^\sharp \in X\ A(x, x^\sharp)\}$, on a

$$\mathcal{A}(s) \subset A^{-1}(\mathcal{A}(s^\sharp)),$$

ce qui donne une condition de sécurité.

La méthode proposée part d'un graphe abstrait complet (abstraction sûre, mais inintéressante) et ôte des arêtes $(x^\sharp, y^\sharp)$ qui peuvent l'être en conservant la sûreté de l'abstraction, c'est-à-dire que :

$$\forall x, y \in \Sigma\ A(x, x^\sharp) \wedge x \xrightarrow{s} y \Rightarrow \neg A(y, y^\sharp).$$

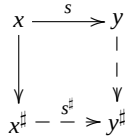
La vérité de cette formule peut être établie de deux façons :

- des moyens syntaxiques relatifs à la classe de système considérés, qui reviennent à l'application de certaines abstractions exactes d'opérateurs primitifs comme ceux d'assignation à l'environnement ;
- succès d'un prouveur de théorèmes (PVS) sur cette formule.

Cette méthode n'est évidemment pas complète, le problème général étant indécidable (voir 2.2.1). On espère cependant arriver à approximer raisonnablement bien la fonction afin de pouvoir prouver les propriétés voulues.

Dans notre cas, nous aurions pu nous servir d'une méthode analogue pour prouver la correction de certains opérateurs élémentaires. Prenons une abstraction

[2] On note souvent cette condition



$\wp(A) \underset{f}{\overset{f^{-1}}{\leftrightarrows}} \wp(B)$ où B est un ensemble fini et une fonction $g : A \to A$. g détermine canoniquement un opérateur image directe $g : \wp(A) \to \wp(A)$. Nous voulons abstraire cet opérateur. Pour chaque élément Y de $\wp(B)$, pour chaque élément b de B, nous essayons de prouver que $f^{-1}(Y) \cap g^{-1}(b) = \emptyset$. Si le prouveur y arrive, nous posons $g^\sharp(Y).b = \bot$, sinon $g^\sharp(Y).b = \top$. Il est clair que cette méthode fournit une abstraction sûre. De plus, une preuve de la correction de cette abstraction peut être construite à partir des preuves des formules $f^{-1}(Y) \cap g^{-1}(b) = \emptyset$.

Avec un prouveur pouvant prouver des formules d'arithmétique élémentaire sans se laisser déconcerter par les quantificateurs existentiels et une tactique adaptée, il est ainsi possible d'extraire les opérateurs d'addition, soustraction, multiplication sur le treillis des signes, ainsi que les constantes.

3.4 Conclusion

Nous avons implementé deux exemples d'interpréteurs abstraits, en avons formellement prouvé entièrement l'un deux et avons prouvé la plus grande partie de la correction du deuxième, cette deuxième preuve, fort fastidieuse, n'ayant pu être achevée faute de temps. Ce faisant, nous avons essayé de dégager les obstacles à la fois théoriques (nous donnons notamment des résultats d'indécidabilité) et pratiques, et essayé de donner des pistes pour la résolution des problèmes.

Annexe A

Code exporté

A.1 Expressions régulières

```
type 'x regexpr =
    Void
  | App of ('x regexpr) * ('x regexpr)
  | Star of ('x regexpr)
  | Letter of 'x
  | Disj of ('x regexpr) * ('x regexpr)
;;

let is_not_empty xs =
  if (xs (None)) then true else
  try
    for i=0 to 255
    do
      if (xs (Some (Char.chr i))) then raise Exit
    done; false
  with Exit->true
;;

let equal j k = j=k
;;

let sem_regexpr =
  let rec sem' = function
    Void -> (fun w -> match w with
                        None -> true
                      | Some a -> false)
  | App (r1, r2) -> (fun w ->
      match w with
        None ->
          (match sem' r1 None with
             true -> sem' r2 None
           | false -> false)
      | Some h ->
          (match sem' r1 (Some h) with
             true ->
```

```
              (match is_not_empty (sem' r2) with
                 true -> true
                | false ->
                    (match sem' r1 None with
                       true -> sem' r2 (Some h)
                     | false -> false))
          | false ->
              (match sem' r1 None with
                 true -> sem' r2 (Some h)
               | false -> false)))
  | Star r0 -> (fun x ->
      match x with
        None -> true
      | Some a -> sem' r0 x)
  | Letter a -> (fun w ->
      match w with
        None -> false
      | Some h -> equal a h)
  | Disj (r1, r2) -> (fun w ->
      match sem' r1 w with
        true -> true
      | false -> sem' r2 w)
  in sem'
;;
```

Annexe B

Portions du développement en Coq

B.1 Expressions régulières

```
Require PolyList.
Require proj1_list.
Require lattice_tools.
Require galois.
Require bool_tools.
Require Bool.
Require abstractions.
Require functions.
Require Relations.
Require lattices.
Require operators.
Require fixpoints.
Require general.
Require chains.
Require Ensembles.

Section regexp_sec.
Variable X: Set.
Variable equal: X->X->bool.
Variable equal_ok: (x,y: X) (bool_to_prop (x=y) (equal x y)).

(* Fixpoint napp [n: nat]: (list X)->(list X) :=
  Cases n of
    O => [x: (list X)] (nil X)
  | (S n) => [x: (list X)] (app x (napp n x))
  end.

Lemma napp_eq_nil: (n: nat) (x: (list X)) ((napp n x) = (nil X)) ->
   ((n = O) \/ (x = (nil X))).
Proof.
  Intro n.
  Case n.
  Left .
  Trivial.

  Simpl.
  Right .
  Cut x=(nil X)/\(napp n0 x)=(nil X).
  Intuition.

  Apply app_eq_nil.
  Assumption.
Save. *)

Inductive regexp: Set :=
    Void: regexp
  | App: regexp -> regexp -> regexp
  | Star: regexp -> regexp
  | Letter: X -> regexp
  | Disj: regexp -> regexp -> regexp(*
  | Plus: regexp -> regexp*).
```

```
Local A:=(parts_lattice (list X)).

Fixpoint sem [r: regexp]: (lat_T A) :=
  Cases r of
    Void => [w: (list X)]
      Cases w of
        nil =>true
      | _ => false
      end

  | (App r1 r2) => (app_concrete X (sem r1) (sem r2))

  | (Star r) => (lat_lfp A (Compose (add_empty_concrete X)
                                    (app_concrete X (sem r))))

  | (Letter a) => [w: (list X)]
      Cases w of
        (cons h _) => (equal a h)
      | _ => false
      end

  | (Disj r1 r2) => [w: (list X)] (orb (sem r1 w) (sem r2 w))
  end.

Local B:=(parts_lattice (option X)).

Variable is_not_empty: (lat_T B)->bool.
Hypothesis is_not_empty_ok: (pe: (lat_T B))
  (bool_to_prop (EX x: (option X) | (pe x)=true)
  (is_not_empty pe)).

Hypothesis find_back_proj1_list: (x:(option X)) (ys: (lat_T A))
    (bool_least_upper_bound
      [t0:bool](EX x1:(list X) | (proj1_list X x1)=x & (ys x1)=t0))
     =true
-> (EX x1:(list X) | (proj1_list X x1)=x & (ys x1)=true).

Fixpoint sem' [r: regexp]: (lat_T B) :=
  Cases r of
    Void => [w: (option X)]
      Cases w of
        none => true
      | _ => false
      end

  | (App r1 r2) => (app_abstract X is_not_empty (sem' r1) (sem' r2))

  | (Star r) => (add_empty_abstract X (sem' r))

  | (Letter a) =>  [w: (option X)]
      Cases w of
        (some h) => (equal a h)
      | _ => false
      end

  | (Disj r1 r2) => [w: (option X)] (orb (sem' r1 w) (sem' r2 w))
  end.

Definition abstr:= (parts_abstraction (list X) (option X) (proj1_list X)).

Lemma bottom_is_empty:
  (is_not_empty [_:(option X)]false)=false.
Proof.
  Cut ([pe:(lat_T B)]
        (bool_to_prop (EX x:(option X) | (pe x)=true) (is_not_empty pe))
        [_:(option X)]false).
  Case (is_not_empty [_:(option X)]false).
  Simpl.
  Intro.
  Case H.
  Auto.

  Auto.

  Exact (is_not_empty_ok [_:(option X)]false).
Save.

Lemma star_sem_simplifies:
  (xs: (lat_T B))
  (lat_lfp B (Compose (add_empty_abstract X)
                          (app_abstract X is_not_empty xs)))=
  (add_empty_abstract X xs).
Proof.
```

```
Intro.
Simpl.
Apply extensionality.
Intro ox.
Unfold lat_lfp compose.
Unfold lfp.
Simpl.
Unfold power_least_upper_bound omega_chain_image bottom.
Simpl.
Unfold power_least_upper_bound.
Replace [x:(option X)]
         (bool_least_upper_bound
           [y0:bool]
            (EX f0:(option X)->bool | (Empty_set (option X)->bool f0)
               & (f0 x)=y0)) with [x:(option X)]false.
Case ox.
Clear ox.
Simpl.
Apply least_upper_bound_unique
 with T:=bool R:=bool_order
      pe:=[y:bool]
           (EX f:(option X)->bool |
                (EX n:nat |
                   (omega_iterate (option X)->bool
                     [x:(option X)->bool]
                      (add_empty_abstract X
                        (app_abstract X is_not_empty xs x))
                     [_:(option X)]false n)=f) & (f (none X))=y).
Exact bool_order_ok.

Split.
Apply bool_least_upper_bound_ok.

Split.
Unfold is_upper_bound.
Intro t.
Case t.
(Simpl; Trivial).

(Simpl; Trivial).

Intro.
Unfold is_upper_bound.
Intro H.
Apply H.
Split
 with ([n:nat]
        (omega_iterate (option X)->bool
          [x:(option X)->bool]
           (add_empty_abstract X (app_abstract X is_not_empty xs x))
          [_:(option X)]false n) (S O)).
Split with (S O).
Trivial.

Simpl.
Trivial.

Clear ox.
Intro x.
Simpl.
Apply least_upper_bound_unique
 with T:=bool R:=bool_order
      pe:=[y:bool]
           (EX f:(option X)->bool |
                (EX n:nat |
                   (omega_iterate (option X)->bool
                     [x0:(option X)->bool]
                      (add_empty_abstract X
                        (app_abstract X is_not_empty xs x0))
                     [_:(option X)]false n)=f) & (f (some X x))=y).
Exact bool_order_ok.

Split.
Apply bool_least_upper_bound_ok.

Split.
Unfold is_upper_bound.
Intros t Z.
Case Z.
Clear Z.
Intros xs Z.
Case Z.
Clear Z.
Intro n.
```

```
Case n.
Simpl.
Clear n.
Intro R.
Rewrite <- R.
Clear R.
Intro R.
Rewrite <- R.
Simpl.
Trivial.

Clear n.
Intro n.
Case n.
Simpl.
Intro R.
Rewrite <- R.
Clear R.
Simpl.
Intro R.
Rewrite <- R.
Clear R t.
Replace (is_not_empty [_:(option X)]false) with false.
Rewrite andb_b_false.
Simpl.
Rewrite andb_b_false.
Simpl.
Trivial.

Cut ([pe:(lat_T B)]
      (bool_to_prop (EX x:(option X) | (pe x)=true) (is_not_empty pe))
      [_:(option X)]false).
Case (is_not_empty [_:(option X)]false).
Simpl.
Intro K.
Case K.
Auto.

Auto.

Exact (is_not_empty_ok [_:(option X)]false).

Clear n.
Intro n.
Intro R.
Rewrite <- R.
Clear R xs0.
Intro R.
Rewrite <- R.
Clear R t.
Replace (omega_iterate (option X)->bool
          [x0:(option X)->bool]
           (add_empty_abstract X (app_abstract X is_not_empty xs x0))
          [_:(option X)]false (S (S n))) with (add_empty_abstract X xs).
Simpl.
Case (xs (some X x)).
(Simpl; Trivial).

(Simpl; Trivial).

Generalize n.
Clear n.
Induction n.
Simpl.
Apply extensionality.
Intro ox.
Case ox.
Clear ox.
Simpl.
Trivial.

Clear ox.
Intro x.
Simpl.
Replace (is_not_empty
          (add_empty_abstract X
            (app_abstract X is_not_empty xs [_:(option X)]false)))
 with true.
Replace (is_not_empty [_:(option X)]false) with false.
Rewrite andb_b_false.
Rewrite andb_b_true.
Simpl.
Rewrite andb_b_false.
Rewrite andb_b_false.
```

```
Rewrite orb_b_false.
Trivial.

Symmetry.
Exact bottom_is_empty.

Cut ([pe:(lat_T B)]
      (bool_to_prop (EX x:(option X) | (pe x)=true) (is_not_empty pe))
      (add_empty_abstract X
        (app_abstract X is_not_empty xs [_:(option X)]false))).
Case (is_not_empty
       (add_empty_abstract X
         (app_abstract X is_not_empty xs [_:(option X)]false))).
Auto.

Simpl.
Unfold not.
Intro K.
Apply False_ind.
Apply K.
Split with (none X).
Simpl.
Trivial.

Exact (is_not_empty_ok
        (add_empty_abstract X
          (app_abstract X is_not_empty xs [_:(option X)]false))).

Clear n.
Intro n.
Simpl.
Intro R.
Rewrite <- R.
Clear R.
Apply extensionality.
Intro ox.
Case ox.
Clear ox.
Simpl.
Trivial.

Clear ox.
Intro x.
Simpl.
Replace (is_not_empty (add_empty_abstract X xs)) with true.
Case (xs (some X x0)).
Simpl.
Trivial.

Simpl.
Rewrite andb_b_false.
Trivial.

Cut ([pe:(lat_T B)]
      (bool_to_prop (EX x:(option X) | (pe x)=true) (is_not_empty pe))
      (add_empty_abstract X xs)).
Case (is_not_empty (add_empty_abstract X xs)).
Auto.

Simpl.
Unfold not.
Intro K.
Apply False_ind.
Apply K.
Split with (none X).
Simpl.
Trivial.

Exact (is_not_empty_ok (add_empty_abstract X xs)).

Unfold is_upper_bound.
Intros t H.
Apply H.
Split
 with (omega_iterate (option X)->bool
        [x0:(option X)->bool]
         (add_empty_abstract X (app_abstract X is_not_empty xs x0))
        [_:(option X)]false (S (S O))).
Split with (S (S O)).
Trivial.

Simpl.
Replace (is_not_empty
          (add_empty_abstract X
```

```
            (app_abstract X is_not_empty xs [_:(option X)]false)))
   with true.
  Rewrite bottom_is_empty.
  Rewrite andb_b_false.
  Rewrite andb_b_false.
  Simpl.
  Rewrite andb_b_false.
  Rewrite andb_b_true.
  Rewrite orb_b_false.
  Trivial.

  Cut ([pe:(lat_T B)]
       (bool_to_prop (EX x:(option X) | (pe x)=true) (is_not_empty pe))
       (add_empty_abstract X
         (app_abstract X is_not_empty xs [_:(option X)]false))).
  Case (is_not_empty
        (add_empty_abstract X
          (app_abstract X is_not_empty xs [_:(option X)]false))).
  Auto.

  Simpl.
  Unfold not.
  Intro K.
  Apply False_ind.
  Apply K.
  Split with (none X).
  Simpl.
  Trivial.

  Exact (is_not_empty_ok
          (add_empty_abstract X
            (app_abstract X is_not_empty xs [_:(option X)]false))).

  Apply extensionality.
  Clear ox.
  Intro ox.
  Symmetry.
  Apply bool_constant_ffalse_lub
   with X:=(option X)->bool
        P:=[f0:(option X)->bool](Empty_set (option X)->bool f0)
        f:=[f0:(option X)->bool](f0 ox).
  Intros x Hx.
  Case Hx.
Save.

Theorem abstr_ok: (r: regexp) (Alpha abstr (sem r)) = (sem' r).
Proof.
  Induction r.
  Simpl.
  Apply extensionality.
  Intro ox.
  Case ox.
  Clear ox.
  Apply bool_exists_x_true
   with X:=(list X) P:=[x:(list X)](proj1_list X x)=(none X)
        f:=[x:(list X)]Cases x of
                         nil => true
                       | (cons _ _) => false
                       end x:=(nil X).
  Simpl; Trivial.

  Trivial.

  Clear ox.
  Intro x.
  Apply bool_constant_ffalse_lub
   with X:=(list X) P:=[x0:(list X)](proj1_list X x0)=(some X x)
        f:=[x0:(list X)]Cases x0 of
                          nil => true
                        | (cons _ _) => false
                        end.
  Intro l.
  Case l.
  Simpl.
  Intro.
  Discriminate.

  Clear l.
  Auto.

  Clear r.
  Intros r1 Hr1 r2 Hr2.
  Simpl.
  Rewrite <- Hr1.
```

```
Rewrite <- Hr2.
Replace (app_abstract X is_not_empty
          (gal_alpha A (power_lattice (option X) bool_lattice) abstr
            (sem r1))
          (gal_alpha A (power_lattice (option X) bool_lattice) abstr
            (sem r2)))
 with (Alpha abstr (app_concrete X (sem r1) (sem r2))).
Simpl.
Trivial.

Symmetry.
Exact (app_abstr_works X is_not_empty is_not_empty_ok
        find_back_proj1_list (sem r1) (sem r2)).

Clear r.
Intros r R.
Simpl.
Rewrite <- R.

Rewrite <- star_sem_simplifies.

Cut (lat_lfp B (compose (option X)->bool (option X)->bool (option X)->bool
        (add_empty_abstract X)
(app_abstract X is_not_empty
        (gal_alpha A (power_lattice (option X) bool_lattice) abstr
          (sem r)))))=
    (Alpha abstr (lat_lfp A (Compose
        (add_empty_concrete X) (app_concrete X (sem r))))).
Intro K.
Change ([y:(option X)]
   (bool_least_upper_bound
     [t:bool]
      (EX x:(list X) | (proj1_list X x)=y
         & (lat_lfp A
              (compose (list X)->bool (list X)->bool (list X)->bool
                (add_empty_concrete X) (app_concrete X (sem r))) x)=t)))
   = (lat_lfp B
      (compose (option X)->bool (option X)->bool (option X)->bool
        (add_empty_abstract X)
        (app_abstract X is_not_empty
          (gal_alpha A (power_lattice (option X) bool_lattice) abstr
            (sem r))))).
Rewrite K.
Clear K.
Apply extensionality with A:=(option X) B:=bool.
Intro ox.
Apply least_upper_bound_unique
 with T:=bool R:=bool_order
      pe:=[t:bool]
    (EX x:(list X) | (proj1_list X x)=ox
             & (lat_lfp A
                  (compose (list X)->bool (list X)->bool (list
  X)->bool
                (add_empty_concrete X)
 (app_concrete X (sem r))) x)=t).
Exact bool_order_ok.

Split.
Apply bool_least_upper_bound_ok.

Simpl.
Apply bool_least_upper_bound_ok.

Symmetry.
Apply fixpoint_transfert2.

Apply compose_omega_upper_continuous with
  T1:=(lat_T A) R1:=(lat_R A)
  least_upper_bound1:=(lat_least_upper_bound_cm A)
  T2:=(lat_T A) R2:=(lat_R A)
  least_upper_bound2:=(lat_least_upper_bound_cm A)
  T3:=(lat_T A) R3:=(lat_R A)
  least_upper_bound3:=(lat_least_upper_bound_cm A)
  f23:=(add_empty_concrete X)
  f12:=(app_concrete X (sem r)).
Apply lat_R_order.
Apply lat_R_order.
Apply lub_morphism_is_omega_upper_continuous.
Unfold A.
Apply app_concrete_right_lub_morphism.
Unfold A.
Apply add_empty_concrete_omega_upper_continuous.

Rewrite R.
```

```
Apply compose_omega_upper_continuous with
  T1:=(lat_T B) R1:=(lat_R B)
  least_upper_bound1:=(lat_least_upper_bound_cm B)
  T2:=(lat_T B) R2:=(lat_R B)
  least_upper_bound2:=(lat_least_upper_bound_cm B)
  T3:=(lat_T B) R3:=(lat_R B)
  least_upper_bound3:=(lat_least_upper_bound_cm B)
  f23:=(add_empty_abstract X)
  f12:=(app_abstract X is_not_empty (sem' r)).
Apply lat_R_order.
Apply lat_R_order.
Apply lub_morphism_is_omega_upper_continuous.
Unfold B.
Apply app_abstract_right_lub_morphism.
Exact is_not_empty_ok.
Unfold B.
Apply add_empty_abstract_omega_upper_continuous.

Unfold A B.
Intro x.
Unfold compose.
Symmetry.
Rewrite (add_empty_abstr_works X).
Apply (equal_apply (lat_T B) (lat_T B) (add_empty_abstract X)).
Exact (app_abstr_works X is_not_empty is_not_empty_ok
        find_back_proj1_list (sem r) x).

Simpl.
Intro x.
Apply extensionality.
Intro ox.
Case ox.
Clear ox.
Apply bool_constant_ffalse_lub
 with X:=(list X) P:=[x0:(list X)](proj1_list X x0)=(none X)
      f:=[x0:(list X)]
          Cases x0 of
            nil => false
          | (cons h _) => (equal x h)
          end.
Intro l.
Case l.
Auto.

Clear l.
Intros head tail.
Simpl.
Intro.
Discriminate.

Clear ox.
Intro x.
Apply least_upper_bound_unique
 with T:=bool R:=bool_order
      pe:=[t:bool]
           (EX x1:(list X) | (proj1_list X x1)=(some X x0)
              & (Cases x1 of
                   nil => false
                 | (cons h _) => (equal x h)
                 end)=t).
Exact bool_order_ok.

Split.
Apply bool_least_upper_bound_ok.

Split.
Unfold is_upper_bound.
Intros t Z.
Case Z.
Clear Z.
Intro l.
Case l.
Clear l.
Simpl.
Intro.
Discriminate.

Simpl.
Intros head tail R.
Injection R.
Clear R.
Intro R.
Rewrite R.
Clear R head.
```

```
Intro R.
Rewrite R.
Case t.
(Simpl; Trivial).

(Simpl; Trivial).

Intros t H.
Unfold is_upper_bound in H.
Apply H.
Split with (cons x0 (nil X)).
Simpl.
Trivial.

Trivial.

Clear r.
Intros r1 Hr1 r2 Hr2.
Simpl.
Apply extensionality.
Intro ox.
Apply least_upper_bound_unique
 with T:=bool R:=bool_order
      pe:=[t:bool]
           (EX x:(list X) | (proj1_list X x)=ox
              & (orb (sem r1 x) (sem r2 x))=t).
Exact bool_order_ok.

Split.
Apply bool_least_upper_bound_ok.

Split.
Unfold is_upper_bound.
Rewrite <- Hr1.
Rewrite <- Hr2.
Simpl.
Intros t Z.
Case Z.
Clear Z.
Case t.
Intros l Hl C.
Cut (sem r1 l)=true\/(sem r2 l)=true.
Clear C.
Intro C.
Case C.
Clear C.
Intro C.
Apply bool_orb_order3.
Change (bool_order true
         (bool_least_upper_bound
           [t:bool](EX x:(list X) | (proj1_list X x)=ox & (sem r1 x)=t))).
Replace (bool_least_upper_bound
          [t:bool](EX x:(list X) | (proj1_list X x)=ox & (sem r1 x)=t))
 with true.
(Simpl; Trivial).

Symmetry.
Apply bool_exists_x_true
 with X:=(list X) P:=[x:(list X)](proj1_list X x)=ox
      f:=[x:(list X)](sem r1 x) x:=l.
Assumption.

Assumption.

Clear C.
Intro C.
Apply bool_orb_order4.
Change (bool_order true
         (bool_least_upper_bound
           [t:bool](EX x:(list X) | (proj1_list X x)=ox & (sem r2 x)=t))).
Replace (bool_least_upper_bound
          [t:bool](EX x:(list X) | (proj1_list X x)=ox & (sem r2 x)=t))
 with true.
Simpl.
Trivial.

Symmetry.
Apply bool_exists_x_true
 with X:=(list X) P:=[x:(list X)](proj1_list X x)=ox
      f:=[x:(list X)](sem r2 x) x:=l.
Assumption.

Assumption.
```

```
Cut (orb (sem r1 l) (sem r2 l))=true.
Case (sem r1 l).
Auto.

Auto.

Assumption.

Simpl.
Auto.

Unfold is_upper_bound.
Intros t H.
Apply H.
Clear H.
Rewrite <- Hr1.
Rewrite <- Hr2.
Case ox.
Simpl.
Split with (nil X).
Simpl.
Trivial.

Change (orb (sem r1 (nil X)) (sem r2 (nil X)))
       =(orb
          (bool_least_upper_bound
            [t:bool]
             (EX x:(list X) | (proj1_list X x)=(none X) & (sem r1 x)=t))
          (bool_least_upper_bound
            [t:bool]
             (EX x:(list X) | (proj1_list X x)=(none X) & (sem r2 x)=t))).
Replace (orb
          (bool_least_upper_bound
            [t:bool]
             (EX x:(list X) | (proj1_list X x)=(none X) & (sem r1 x)=t))
          (bool_least_upper_bound
            [t:bool]
             (EX x:(list X) | (proj1_list X x)=(none X) & (sem r2 x)=t)))
 with (bool_least_upper_bound
        [t:bool]
         (EX x:(list X) | (proj1_list X x)=(none X)
            & (orb (sem r1 x) (sem r2 x))=t)).
Symmetry.
Apply nil_proj with X:=X pe:=[x:(list X)](orb (sem r1 x) (sem r2 x)).

Symmetry.
Apply bool_orb_lub_lub
 with X:=(list X) P:=[x:(list X)](proj1_list X x)=(none X) f1:=(sem r1)
      f2:=(sem r2).

Simpl.
Clear ox.
Intro x.
Change (EX x0:(list X) | (proj1_list X x0)=(some X x)
          & (orb (sem r1 x0) (sem r2 x0))
             =(orb
                (bool_least_upper_bound
                  [t:bool]
                   (EX x1:(list X) | (proj1_list X x1)=(some X x)
                      & (sem r1 x1)=t))
                (bool_least_upper_bound
                  [t:bool]
                   (EX x1:(list X) | (proj1_list X x1)=(some X x)
                      & (sem r2 x1)=t)))).
Replace (orb
          (bool_least_upper_bound
            [t:bool]
             (EX x1:(list X) | (proj1_list X x1)=(some X x)
                & (sem r1 x1)=t))
          (bool_least_upper_bound
            [t:bool]
             (EX x1:(list X) | (proj1_list X x1)=(some X x)
                & (sem r2 x1)=t)))
 with (bool_least_upper_bound
        [t:bool]
         (EX x1:(list X) | (proj1_list X x1)=(some X x)
            & (orb (sem r1 x1) (sem r2 x1))=t)).
Cut (bool_least_upper_bound
      [t:bool]
       (EX x1:(list X) | (proj1_list X x1)=(some X x)
          & (orb (sem r1 x1) (sem r2 x1))=t))=true
    \/(bool_least_upper_bound
        [t:bool]
         (EX x1:(list X) | (proj1_list X x1)=(some X x)
```

```
            & (orb (sem r1 x1) (sem r2 x1))=t))=false.
  Intro C.
  Case C.
  Clear C.
  Intro C.
  Rewrite C.
  Apply find_back_proj1_list
   with x:=(some X x) ys:=[x1:(list X)](orb (sem r1 x1) (sem r2 x1)).
  Assumption.

  Clear C.
  Intro C.
  Rewrite C.
  Split with (cons x (nil X)).
  Simpl.
  Trivial.

  Cut (bool_order
        (orb (sem r1 (cons x (nil X))) (sem r2 (cons x (nil X)))) false).
  Case (orb (sem r1 (cons x (nil X))) (sem r2 (cons x (nil X)))).
  Simpl.
  Intro K.
  Apply False_ind.
  Assumption.

  Auto.

  Rewrite <- C.
  Apply (Proj1
          (bool_least_upper_bound_ok
            [t:bool]
             (EX x1:(list X) | (proj1_list X x1)=(some X x)
                & (orb (sem r1 x1) (sem r2 x1))=t))).
  Split with (cons x (nil X)).
  Simpl.
  Trivial.

  Trivial.

  Case (bool_least_upper_bound
         [t:bool]
          (EX x1:(list X) | (proj1_list X x1)=(some X x)
             & (orb (sem r1 x1) (sem r2 x1))=t)).
  Auto.

  Auto.

  Symmetry.
  Apply bool_orb_lub_lub
   with X:=(list X) P:=[x1:(list X)](proj1_list X x1)=(some X x)
        f1:=(sem r1) f2:=(sem r2).
Save.

End regexp_sec.
```

B.2 Treillis

```
Require Relations.
Require Ensembles.
Require lattices.
Require functions.
Require general.
Require fixpoints.

Record lattice : Type := {
  lat_T : Set;
  lat_R : (relation lat_T);
  lat_R_order : (order lat_T lat_R);
  lat_least_upper_bound : (Ensemble lat_T)->lat_T;
  lat_least_upper_bound_ok : (pe: (Ensemble lat_T))
    (is_least_upper_bound lat_T lat_R
      (lat_least_upper_bound pe) pe);
  lat_greatest_lower_bound : (Ensemble lat_T)->lat_T;
  lat_greatest_lower_bound_ok : (pe: (Ensemble lat_T))
    (is_greatest_lower_bound lat_T lat_R
      (lat_greatest_lower_bound pe) pe)
```

```
}.

Coercion lat_T: lattice >-> SORTCLASS.
Hint lat_R_order.
Hint lat_least_upper_bound_ok.
Hint lat_greatest_lower_bound_ok.

Definition lat_least_upper_bound_cm:= [T: lattice]
  [pe: (Ensemble (lat_T T))] (exist (lat_T T)
    ([x: (lat_T T)] (is_least_upper_bound (lat_T T) (lat_R T) x pe))
    (lat_least_upper_bound T pe) (lat_least_upper_bound_ok T pe)).

Definition lat_bottom := [T: lattice]
  (lat_least_upper_bound T (Empty_set (lat_T T))).

Definition lat_top := [T: lattice]
  (lat_greatest_lower_bound T (Empty_set (lat_T T))).

Lemma lat_bottom_ok : (T: lattice)
  (x: (lat_T T)) (lat_R T (lat_bottom T) x).
Proof.
    Intros.
  Unfold lat_bottom.
  Cut (is_least_upper_bound (lat_T T) (lat_R T)
        (lat_least_upper_bound T (Empty_set (lat_T T)))
        (Empty_set (lat_T T))).
  Intro zoinx.
  Case zoinx.
  Intros zoinx1 zoinx2.
  Apply zoinx2.
  Unfold is_upper_bound.
  Intros y H.
  Case H.

  Apply lat_least_upper_bound_ok.
Save.

Lemma lat_top_ok : (T: lattice)
  (x: (lat_T T)) (lat_R T x (lat_top T)).
Proof.
    Intros.
  Unfold lat_bottom.
  Cut (is_greatest_lower_bound (lat_T T) (lat_R T)
        (lat_greatest_lower_bound T (Empty_set (lat_T T)))
        (Empty_set (lat_T T))).
  Intro zoinx.
  Case zoinx.
  Intros zoinx1 zoinx2.
  Apply zoinx2.
  Unfold is_lower_bound.
  Intros y H.
  Case H.

  Apply lat_greatest_lower_bound_ok.
Save.

Hint lat_bottom_ok.
Hint lat_top_ok.

Definition lat_lfp := [T: lattice]
  (lfp (lat_T T) (lat_R T) (lat_least_upper_bound_cm T)).

Section lattice_builders.
Definition bool_order: bool->bool->Prop :=
  [x: bool]
  Cases x of
    false => [y: bool] True
  | true => [y: bool]
      Cases y of
        false => False
      | true => True
      end
  end.

Lemma bool_order_ok: (order bool bool_order).
Proof.
  Apply Build_order.
  Unfold reflexive.
  Intro x.
  Unfold bool_order.
  Case x.
  Trivial.

  Trivial.
```

```
  Unfold transitive.
  Unfold bool_order.
  Intro x.
  Case x.
  Intro y.
  Case y.
  Intro z.
  Case z.
  Auto.

  Auto.

  Intro z.
  Case z.
  Auto.

  Auto.

  Intro y.
  Case y.
  Intro z.
  Case z.
  Auto.

  Auto.

  Auto.

  Unfold antisymmetric.
  Intro x.
  Unfold bool_order.
  Case x.
  Intro y.
  Case y.
  Auto.

  Intro H.
  Apply False_ind.
  Assumption.

  Intro y.
  Case y.
  Intros.
  Apply False_ind.
  Assumption.

  Auto.
Save.

Axiom bool_least_upper_bound: (Ensemble bool)->bool.

Axiom bool_least_upper_bound_ok: (X: (Ensemble bool))
  (is_least_upper_bound bool bool_order (bool_least_upper_bound X) X).

Axiom bool_greatest_lower_bound: (Ensemble bool)->bool.

Axiom bool_greatest_lower_bound_ok: (X: (Ensemble bool))
  (is_greatest_lower_bound bool bool_order (bool_greatest_lower_bound X) X).

Definition bool_lattice : lattice := (Build_lattice
  bool bool_order bool_order_ok
  bool_least_upper_bound
  bool_least_upper_bound_ok
  bool_greatest_lower_bound
  bool_greatest_lower_bound_ok).

Definition power_order: (X, Y: Set) (relation Y)->(relation (X->Y))
 := [X,Y: Set] [RY: (relation Y)] [f,g: X->Y]
    (x: X) (RY (f x) (g x)).

Lemma power_order_is_an_order: (X, Y: Set) (RY: (relation Y))
  (RY_order: (order Y RY)) (order (X->Y) (power_order X Y RY)).
Proof.
  Intros X Y RY RY_H.
  Case RY_H.
  Intros.
  Apply Build_order.
  Unfold reflexive in ord_refl.
  Unfold reflexive.
  Unfold power_order.
  Auto.

  Unfold transitive power_order.
```

```
  Unfold transitive in ord_trans.
  Intros.
  Apply ord_trans with y:=(y x0).
  Auto.

  Auto.

  Unfold antisymmetric.
  Unfold antisymmetric in ord_antisym.
  Unfold power_order.
  Intros.
  Apply extensionality.
  Intro.
  Apply ord_antisym.
  Auto.

  Auto.
Save.

Definition power_least_upper_bound:= [X, Y: Set]
  [ lub: (Ensemble Y)->Y ]
  [ pe: (Ensemble (X->Y)) ] [x: X]
  (lub ([y: Y] (EX f: X->Y | (pe f) & (f x)=y))).

Definition power_greatest_lower_bound:= [X, Y: Set]
  [ glb: (Ensemble Y)->Y ]
  [ pe: (Ensemble (X->Y)) ] [x: X]
  (glb ([y: Y] (EX f: X->Y | (pe f) & (f x)=y))).

Lemma power_least_upper_bound_ok : (X, Y: Set)
  (R: (relation Y))
  ( lub: (Ensemble Y)->Y )
  ( lub_ok: (pe: (Ensemble Y)) (is_least_upper_bound Y R (lub pe) pe) )
  ( pe: (Ensemble (X->Y)) )
  (is_least_upper_bound (X->Y) (power_order X Y R)
     (power_least_upper_bound X Y lub pe) pe).
Proof.
  Intros.
  Unfold is_least_upper_bound.
  Split.
  Unfold is_upper_bound power_order power_least_upper_bound.
  Intros.
  Cut (is_least_upper_bound Y R (lub [y0:Y](EX f:? | (pe f) & (f x)=y0))
        [y0:Y](EX f:? | (pe f) & (f x)=y0)).
  Intro zoinx.
  Unfold is_least_upper_bound in zoinx.
  Case zoinx.
  Intros lub_ok1 lub_ok2.
  Clear lub_ok.
  Unfold is_upper_bound in lub_ok1.
  Apply lub_ok1.
  Split with y.
  Assumption.

  Trivial.

  Apply lub_ok.

  Unfold power_order is_upper_bound power_least_upper_bound.
  Intros f Hf x.
  Apply (Proj2 (lub_ok [y:Y](EX f0:X->Y | (pe f0) & (f0 x)=y))).
  Unfold is_upper_bound.
  Intros y Hy.
  Case Hy.
  Clear Hy.
  Intros y H1 H2.
  Rewrite <- H2.
  Apply Hf.
  Assumption.
Save.

Lemma power_greatest_lower_bound_ok : (X, Y: Set)
  (R: (relation Y))
  ( glb: (Ensemble Y)->Y )
  ( glb_ok: (pe: (Ensemble Y)) (is_greatest_lower_bound Y R (glb pe) pe) )
  ( pe: (Ensemble (X->Y)) )
  (is_greatest_lower_bound (X->Y) (power_order X Y R)
     (power_greatest_lower_bound X Y glb pe) pe).
Proof.
  Intros.
  Unfold is_greatest_lower_bound.
  Split.
  Unfold is_lower_bound power_order power_greatest_lower_bound.
  Intros.
```

```
  Cut (is_greatest_lower_bound Y R (glb [y0:Y](EX f:? | (pe f) & (f x)=y0))
        [y0:Y](EX f:? | (pe f) & (f x)=y0)).
  Intro zoinx.
  Unfold is_greatest_lower_bound in zoinx.
  Case zoinx.
  Intros glb_ok1 glb_ok2.
  Clear glb_ok.
  Unfold is_lower_bound in glb_ok1.
  Apply glb_ok1.
  Split with y.
  Assumption.

  Trivial.

  Apply glb_ok.

  Unfold power_order is_lower_bound power_greatest_lower_bound.
  Intros f Hf x.
  Apply (Proj2 (glb_ok [y:Y](EX f0:X->Y | (pe f0) & (f0 x)=y))).
  Unfold is_lower_bound.
  Intros y Hy.
  Case Hy.
  Clear Hy.
  Intros y H1 H2.
  Rewrite <- H2.
  Apply Hf.
  Assumption.
Save.

Definition power_lattice : Set->lattice->lattice := [X: Set] [Y: lattice]
  [tY = (lat_T Y)]
  [rY = (lat_R Y)]
  [lubY = (lat_least_upper_bound Y)]
  [glbY = (lat_greatest_lower_bound Y)]
  [po = (power_order X tY rY)]
  (Build_lattice
    (X->tY) po
    (power_order_is_an_order X tY rY (lat_R_order Y))
    (power_least_upper_bound X tY lubY)
    (power_least_upper_bound_ok X tY rY lubY
       (lat_least_upper_bound_ok Y))
    (power_greatest_lower_bound X tY glbY)
    (power_greatest_lower_bound_ok X tY rY glbY
       (lat_greatest_lower_bound_ok Y)) ).

Definition parts_lattice : Set->lattice := [X: Set]
  (power_lattice X bool_lattice).

  Section product_lattice_sec.
    Variable A, B: lattice.
    Local ZZ:=(lat_T A)*(lat_T B).

    Definition product_order:= [x, y: ZZ]
      Cases x of (pair x1 x2) =>
        Cases y of (pair y1 y2) => (lat_R A x1 y1) /\ (lat_R B x2 y2)
        end
      end.

    Lemma product_order_ok: (order ZZ product_order).
    Proof.
      Apply Build_order.
      Unfold reflexive product_order.
      Intro x; Case x; Intros x1 x2; Clear x.
      Split.
      Cut (reflexive (lat_T A) (lat_R A)).
      Unfold reflexive.
      Auto.

      Apply ord_refl.
      Apply lat_R_order.

      Cut (reflexive (lat_T B) (lat_R B)).
      Unfold reflexive.
      Intuition.

      Apply ord_refl.
      Apply lat_R_order.

      Unfold transitive.
      Intro x; Case x; Intros x1 x2; Clear x.
      Intro y; Case y; Intros y1 y2; Clear y.
      Intro z; Case z; Intros z1 z2; Clear z.
      Unfold product_order.
      Cut (transitive (lat_T A) (lat_R A)).
```

```
  Cut (transitive (lat_T B) (lat_R B)).
  Unfold transitive.
  Intros.
  Case H1; Case H2; Clear H1; Clear H2.
  EAuto.

  Apply ord_trans.
  Apply lat_R_order.

  Apply ord_trans.
  Apply lat_R_order.

  Unfold antisymmetric.
  Intro x; Case x; Intros x1 x2; Clear x.
  Intro y; Case y; Intros y1 y2; Clear y.
  Unfold product_order.
  Intros.
  Case H; Case H0; Clear H; Clear H0.
  Cut (antisymmetric (lat_T A) (lat_R A)).
  Cut (antisymmetric (lat_T B) (lat_R B)).
  Unfold antisymmetric.
  Intros.
  Cut x1=y1.
  Cut x2=y2.
  Intros M N.
  Rewrite M; Rewrite N.
  Reflexivity.

  Auto.

  Auto.

  Apply ord_antisym.
  Apply lat_R_order.

  Apply ord_antisym.
  Apply lat_R_order.
Save.

Definition product_lub := [pe: (Ensemble ZZ)]
(pair (lat_T A) (lat_T B)
  (lat_least_upper_bound A ([a: (lat_T A)] (EX z: ZZ | (pe z)& (Fst z)=a)))
  (lat_least_upper_bound B ([b: (lat_T B)] (EX z: ZZ | (pe z)& (Snd z)=b)))
).

Lemma product_lub_ok: (pe: (Ensemble ZZ))
  (is_least_upper_bound ZZ product_order (product_lub pe) pe).
Proof.
  Intro pe.
  Split.
  Unfold is_upper_bound.
  Intro y; Case y; Intros y1 y2; Clear y.
  Intro Hy.
  Split.
  Apply (Proj1
          (lat_least_upper_bound_ok A
            [a:(lat_T A)](EX z | (pe z) & (Fst z)=a))).
  Split with (y1,y2).
  Assumption.

  Simpl.
  Reflexivity.

  Apply (Proj1
          (lat_least_upper_bound_ok B
            [b:(lat_T B)](EX z | (pe z) & (Snd z)=b))).
  Split with (y1,y2).
  Assumption.

  Simpl; Reflexivity.

  Intro y; Case y; Intros y1 y2; Clear y.
  Unfold is_upper_bound.
  Intro Hy.
  Split.
  Apply (Proj2
          (lat_least_upper_bound_ok A
            [a:(lat_T A)](EX z | (pe z) & (Fst z)=a))).
  Unfold is_upper_bound.
  Intros x1 exx.
  Case exx.
  Intro k; Case k; Intros k1 k2; Clear k.
  Intros Hk Rk.
  Clear exx.
```

```
  EApply proj1 (*#GENTERM (WITHARGS (BINDINGS )) *).
  Unfold product_order in Hy.
  Simpl in Rk.
  Rewrite <- Rk.
  Apply (Hy (k1,k2)).
  Assumption.

  Apply (Proj2
         (lat_least_upper_bound_ok B
           [b:(lat_T B)](EX z | (pe z) & (Snd z)=b))).
  Unfold is_upper_bound.
  Intros x2 exx.
  Case exx.
  Intro k; Clear exx; Case k; Clear k; Intros k1 k2.
  Simpl.
  Intros Hk Rk.
  Rewrite <- Rk.
  Unfold product_order in Hy.
  EApply proj2 (*#GENTERM (WITHARGS (BINDINGS )) *).
  Apply (Hy (k1,k2)).
  Assumption.
Save.

Definition product_glb := [pe: (Ensemble ZZ)]
(pair (lat_T A) (lat_T B)
  (lat_greatest_lower_bound A ([a:(lat_T A)] (EX z: ZZ |(pe z)&(Fst z)=a)))
  (lat_greatest_lower_bound B ([b:(lat_T B)] (EX z: ZZ |(pe z)&(Snd z)=b)))
).

Lemma product_glb_ok: (pe: (Ensemble ZZ))
  (is_greatest_lower_bound ZZ product_order (product_glb pe) pe).
Proof.
  Intro pe.
  Split.
  Unfold is_lower_bound.
  Intro y; Case y; Intros y1 y2; Clear y.
  Intro Hy.
  Split.
  Apply (Proj1
         (lat_greatest_lower_bound_ok A
           [a:(lat_T A)](EX z | (pe z) & (Fst z)=a))).
  Split with (y1,y2).
  Assumption.

  Simpl.
  Reflexivity.

  Apply (Proj1
         (lat_greatest_lower_bound_ok B
           [b:(lat_T B)](EX z | (pe z) & (Snd z)=b))).
  Split with (y1,y2).
  Assumption.

  Simpl; Reflexivity.

  Intro y; Case y; Intros y1 y2; Clear y.
  Unfold is_lower_bound.
  Intro Hy.
  Split.
  Apply (Proj2
         (lat_greatest_lower_bound_ok A
           [a:(lat_T A)](EX z | (pe z) & (Fst z)=a))).
  Unfold is_lower_bound.
  Intros x1 exx.
  Case exx.
  Intro k; Case k; Intros k1 k2; Clear k.
  Intros Hk Rk.
  Clear exx.
  EApply proj1 (*#GENTERM (WITHARGS (BINDINGS )) *).
  Unfold product_order in Hy.
  Simpl in Rk.
  Rewrite <- Rk.
  Apply (Hy (k1,k2)).
  Assumption.

  Apply (Proj2
         (lat_greatest_lower_bound_ok B
           [b:(lat_T B)](EX z | (pe z) & (Snd z)=b))).
  Unfold is_lower_bound.
  Intros x2 exx.
  Case exx.
  Intro k; Clear exx; Case k; Clear k; Intros k1 k2.
  Simpl.
  Intros Hk Rk.
```

```
      Rewrite <- Rk.
      Unfold product_order in Hy.
      EApply proj2 (*#GENTERM (WITHARGS (BINDINGS )) *).
      Apply (Hy (k1,k2)).
      Assumption.
    Save.

     Definition product_lattice := (Build_lattice ZZ
       product_order product_order_ok
       product_lub product_lub_ok
       product_glb product_glb_ok).
   End product_lattice_sec.
End lattice_builders.

Hint bool_order_ok.
Hint bool_least_upper_bound_ok.
```

Bibliographie

[BBC+98] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christo phe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Vigouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual*. INRIA, Domaine de Voluceau - Rocquencourt, BP105, 78153 Le Chesnay Cedex, France, v6.2, draft version edition, Mai 1998.
`http://pauillac.inria.fr/coq/doc1-eng.html`

[BLO98] Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In A.J. Hu and M.Y. Vardi, editors, *Computer Aided Verification, CAV'98, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
`http://www.informatik.uni-kiel.de/~yl/cav98.ps.gz`

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *J. Logic Prog.*, 2-3(13) :103–179, 1992.

[CC98] Patrick Cousot and Radhia Cousot. Introduction to abstract interpretation. Notes de cours du DEA, extraites de [CC92], 1998.
`http://www.dmi.ens.fr/~cousot/cours/DEASP2/CoursDEA_SP2_1998_JLP_92.ps.gz`

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. Thèse d'état ès sciences mathématiques, Université scientifique et médicale de Grenoble, Grenoble, France, 21 mars 1978.

[Cou98] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs*. Thèse de Doctorat, Ecole Normale Supérieure de Lyon, 1998.
`ftp://ftp.lip.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD98-03.ps.Z`

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge Univer-

sity Press, 1989. Translated and with appendices by Paul Taylor, Yves Lafont.

[HS96] Klaus Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In *Formal Methods Europe FME '96*, number 1051 in Lecture Notes in Computer Science, pages 662–681, Oxford, UK, March 1996. Springer-Verlag.
`http://www.csl.sri.com/fme96.html`

[JD90] Jean-Pierre Jouannaud and Nachum Dershowitz. Rewriting systems. In Jan van Leuween, editor, *Handbook of Theoretical Computer Science, volume B*. Elsevier, The MIT Press, 1990.

[Lal90] René Lalement. *Logique, réduction, résolution*. Masson, 1990.

[Ler] Xavier Leroy. *The Objective Caml system, documentation and user's guide*. INRIA.
`http://pauillac.inria.fr/ocaml/htmlman`

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proc. 22nd symp. Principles of Programming Languages (POPL '95)*, pages 142–153. ACM Press, 1995.

[MY78] Michael Machtey and Paul Young. *An Introduction to the General Theory of Algorithms*. Theory of Information series. North-Holland, New-York, 1978.

[ORS97] Sam Owre, John Rushby, and N. Shankar. Integration in PVS : Tables, types, and model checking. In Ed Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems TACAS '97*, number 1217 in Lecture Notes in Computer Science, pages 366–383, Enschede, The Netherlands, April 1997. Springer-Verlag.
`http://www.csl.sri.com/tacas97.html`

[PM89] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse de doctorat, Université Paris 7, January 1989.
`http://www.lri.fr/~paulin/these.ps.gz`