Algorithme certifié de Stålmarck
Stage de Maîtrise

Pierre Letouzey (letouzey@ens.fr)

2 octobre 1998

1 Présentation du stage

1.1 Sujet

Citation de l'URL http ://www.inria.fr/croap/stages98/stalmarck.html :

> Le groupe CROAP s'intéresse au développement de programmes de calcul symbolique certifiés et efficaces. Pour cela on utilise l'assistant de preuve Coq [1] pour développer en parallèle l'algorithme et sa preuve de correction. Les algorithmes définis dans Coq peuvent ensuite être efficacement compilés en OCaml.
>
> Dans ce stage on s'attaquera à un des algorithmes les plus efficaces du calcul booléen : l'algorithme de Stålmarck. Cet algorithme s'intéresse à la satisfiabilité des formules booléennes. Une formules booléenne est formée d'un ensemble de connecteurs logiques (ou, et, implique, etc) et d'un ensemble de variables. Une formule booléenne est dite satisfiable si on peut trouver des valeurs aux variables qui rendent cette formule vraie. L'objectif de ce stage est de définir en Coq l'algorithme de Stålmarck, d'en donner une preuve de correction, et d'évaluer les performances de l'algorithme ainsi obtenu. Une présentation de l'algorithme peut se trouver dans [4].

1.2 Responsable

Laurent Théry (Laurent.Thery@sophia.inria.fr)

1.3 Déroulement

Ce stage a eu lieu à l'INRIA Sophia-Antipolis, du 10 juin 1998 au 10 août. Il s'est déroulé dans de très bonnes conditions :

- L'équipe CROAP s'est montrée très acceuillante et très aidante (chaleureux remerciements à tous...)
- Mise à disposition d'un bureau spatieux (et climatisé...)
- Au niveau informatique : station Alpha 200Mhz, avec accès à un serveur Alpha 533Mhz et 256Mo de RAM. Équipement très confortable, mais pas toujours superflu, comme on le verra après.

2 Principe de l'algorithme de Stålmarck

2.1 Source

La description de l'algorithme, ainsi que tout le travail du stage repose sur l'article de John Harrison [4]. Il n'y a pas eu de recherche bibliographique plus poussée, faute de temps, mais aussi faute de données. En effet l'algorithme de Stålmarck a été breveté en ce qui concerne l'usage commercial, et il semble qu'il y ait peu de publications dessus. Une recherche rapide sur le Web a mené au site de la société fondée par Stålmarck, assez vague concernant l'algorithme lui même, et à l'article [4].

2.2 Principe général

Comme expliqué dans le sujet du stage, l'algorithme résoud le problème de la satisfiabilité d'une formule booléenne. C'est un problème non trivial : Cook [5] a démontré sa NP-complétude. En fait, la version présenté ici de l'algorithme de Stålmarck résoud le problème dual : déterminer si une formule booléenne est une tautologie.

Évidemment, construire la table de vérité de la formule résoud la question. Mais dans des problèmes industriels où le nombre de variables dépasse souvent le millier, cette méthode naïve et exponentielle est complètement inutilisable.

L'algorithme de Stålmarck évite autant que faire se peut d'explorer les différentes affectations possibles des variables booléennes, car c'est cette exploration qui introduit un coût exponentiel. Il essaye plutôt de déduire des informations grâce à des règles du style :

$$\text{De } p \wedge q = \top \text{, déduire } p = \top \text{ et } q = \top$$

Ce qu'on peut condenser en :

$$\frac{p \wedge q}{p \quad q}$$

Si la formule s'y prête bien, on peut parfois conclure directement grâce à de telles règles ! Par contre cela ne marche pas toujours aussi bien. L'algorithme passe alors à une deuxième phase, où il explore petit à petit l'ensemble des valeurs possibles des variables, mais toujours en retirant le maximum d'information des règles.

2.3 La phase préliminaire : normalisation

La donnée initiale se compose d'une formule booléenne, que l'on représente sous forme d'un arbre

- de noeuds $\neg$ (unaire), $\wedge$, $\vee$,$\Rightarrow$,$\Leftrightarrow$ (binaires)
- de feuilles $\top$,$\bot$ et les variables

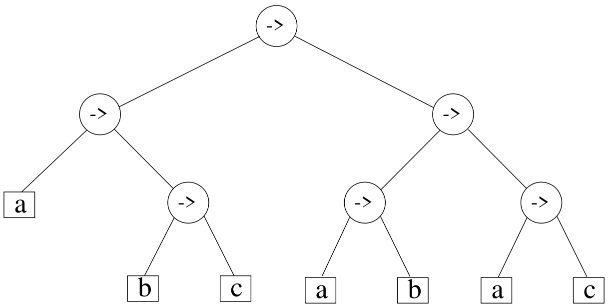

FIG. 1: Arbre de la tautologie $(a \Rightarrow b \Rightarrow c) \Rightarrow (a \Rightarrow b) \Rightarrow (a \Rightarrow c)$

La première étape de l'algorithme consiste à normaliser la formule :

1. élimination des $\top$ et $\bot$
2. suppression des noeuds de type $\vee$, grâce aux lois de De Morgan
3. idem avec $\Rightarrow$ en utilisant $p \Rightarrow q = \neg(p \wedge \neg q)$

Cela ce fait en un seul passage dans l'arbre, de bas en haut, en appliquant récursivement ces règles de réécriture (cf Annexe A pour la liste complète de ces règles).
On se ramène ainsi à un arbre composé de variables et de noeuds $\neg$, $\wedge$ et $\Leftrightarrow$.

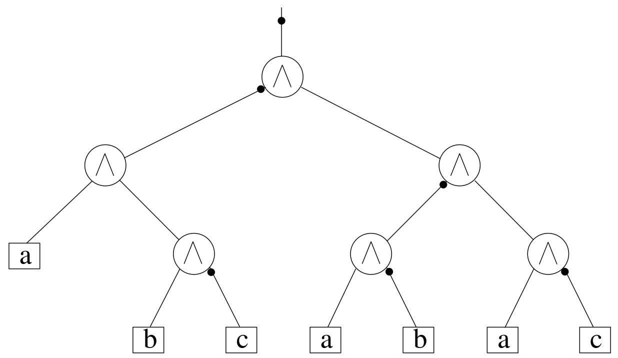

FIG. 2: Arbre précédent normalisé (les points symbolisent les ¬)

2.4 Le découpage en triplets

La suite de l'algorithme ne manipule pas l'arbre entier, mais plutôt ses fragments. On découpe l'arbre en une succession d'élements de type 2 feuilles et 1 noeud binaire, et on utilise de nouvelles variables pour baptiser ces "triplets".

La valeur globale de la formule de départ correspond à la valeur de la variable au sommet - à une négation près -.

On parle de triplets, car chaque élement peut s'ecrire $p \Leftrightarrow q \otimes r$ où $\otimes$ est soit $\wedge$, soit $\Leftrightarrow$, p est une variable, et q et r des litéraux (c.à.d. des variables, ou leur négations).

Dans notre exemple, les six triplets s'écrivent :

$$v_1 \Leftrightarrow b \wedge \neg c \quad (1)$$
$$v_2 \Leftrightarrow a \wedge v_1 \quad (2)$$
$$v_3 \Leftrightarrow a \wedge \neg b \quad (3)$$
$$v_4 \Leftrightarrow a \wedge \neg c \quad (4)$$
$$v_5 \Leftrightarrow \neg v_3 \wedge v_4 \quad (5)$$
$$v_6 \Leftrightarrow \neg v_2 \wedge v_5 \quad (6)$$

Et la valeur de la formule correspond à celle de $\neg v_6$.

2.5 La 0-saturation

Pour prouver la tautologie, on commence par supposer que la formule a la valeur $\perp$, ce qui nous donne une équation de départ, puis on essaie

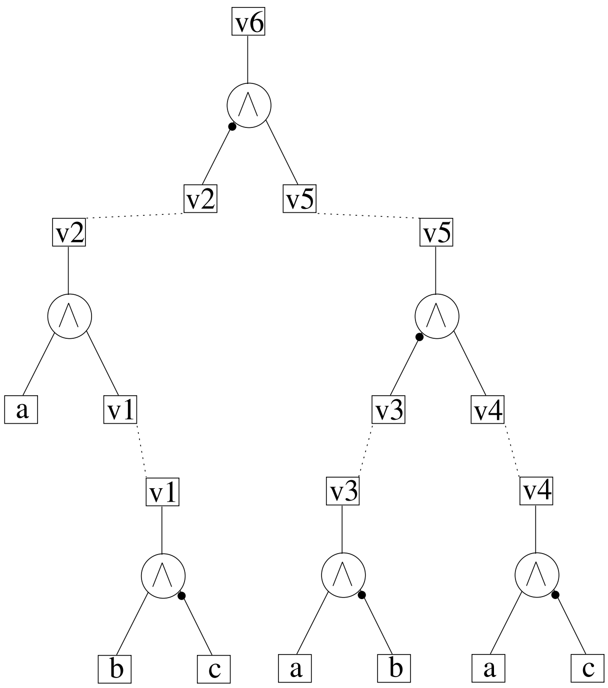

FIG. 3: Arbre précédent découpé en triplets

de déduire de nouvelles équations. L'apparition d'une contradiction du style $p = \neg p$ nous permettra de conclure positivement. Ici on part de l'équation $\neg v_6 = \bot$, soit $v_6 = \top$. La 0-saturation consiste alors à déduire le maximum d'informations possibles c.à.d de nouvelles équations, à partir

- des équations déjà trouvées
- de règles simples sur les triplets, du style :
  Si on a l'équation $p = \neg q$, et le triplet $p \Leftrightarrow q \wedge r$
  En déduire les équations $q = \top$ et $r = \bot$

Voir Annexe B la liste complète des règles.

La 0-saturation suffit ainsi à résoudre l'exemple :

- Grâce à $v_6 = \top$ et au triplet (6) on obtient $v_2 = \bot$ et $v_5 = \top$
- Grâce à $v_5 = \top$ et au triplet (5) on obtient $v_3 = \bot$ et $v_4 = \top$
- Grâce à $v_4 = \top$ et au triplet (4) on obtient $a = \top$ et $c = \bot$
- Grâce à $a = \top$, $v_2 = \bot$ et au triplet (2) on obtient $v_1 = \bot$
- Grâce à $c = \bot$, $v_1 = \bot$ et au triplet (1) on obtient $b = \bot$

- Grâce à $a = \top$, $v_3 = \bot$ et au triplet (3) on obtient $b = \top$
- D'où la contradiction $\top = b = \bot$

L'étape de 0-saturation a trois issues possibles :

- Une contradiction : il n'y a pas de valuation des variables qui permette de rendre la formule fausse, c'est donc une tautologie.
- Pas de contradiction, et toutes les variables affectées à $\top$ ou $\bot$: ces valeurs des variables rendent la formule fausse, autrement dit on a trouvé un contre-exemple, et la formule n'est pas une tautologie.
- Pas de contradiction, et des variables encore non determinées : la 0-saturation ne gagne pas toujours... il faut passer à l'étape suivante.

2.6 La n-saturation, $n > 0$

Dans le cas où la 0-saturation ne permet pas de conclure, on va commencer à affecter les différentes valeurs possibles aux variables, et regarder les résultats. Mais plutôt que d'affecter toutes les variables à la fois comme dans le cas de la table de vérité, on le fait progressivement.

Prenons une variable a. Soit Σ l'ensemble des équations obtenues par 0-saturation à partir de l'équation de départ. On applique de nouveau la 0-saturation, mais à partir de l'ensemble $\Sigma \cup \{a = \top\}$ pour obtenir un nouvel ensemble $\Sigma_\top$. Et de même on obtient $\Sigma_\bot$ à partir de $\Sigma \cup \{a = \bot\}$. Ce qui nous intéresse alors, c'est $\Sigma' = \Sigma_\top \cap \Sigma_\bot$ qui est l'ensembles des équations valables pour toute valeur de a. En effet l'équation de départ et l'hypothèse supplémentaire $a = \top$ impliquent les équations de $\Sigma_\top$ donc celles de Σ'. De même l'equation de départ et l'hypothèse supplémentaire $a = \bot$ impliquent les équations de $\Sigma_\bot$ donc celles de Σ'. Donc on sait que l'équation de départ implique Σ'. Au passage $\Sigma \subseteq \Sigma'$ puisque $\Sigma \subseteq \Sigma_\top$ et $\Sigma \subseteq \Sigma_\bot$: on n'a pas perdu d'information.

La 1-saturation consiste à effectuer successivement l'opération décrite ci-dessus pour toutes les valeurs possibles de a, en faisant à chaque fois grossir l'ensembles Σ des équations déduites. Après la 1-saturation, nouvelle discussion : une contradiction ? un contre-exemple ? ni l'un ni l'autre ?

Si on n'a toujours pas tiré de conclusion après la 1-saturation, On essaie la 2-saturation, qui consiste à affecter deux variables a et b en même temps. On 0-sature et on prend l'intersection à partir des quatres ensembles $\Sigma \cup \{a = \top, b = \top\}$, $\Sigma \cup \{a = \top, b = \bot\}$, $\Sigma \cup \{a = \bot, b = \top\}$ et $\Sigma \cup \{a = \bot, b = \bot\}$. Et encore une fois on répète sur toutes les paires de variables.

Au pire, on continue à augmenter ainsi le nombre de variables affectées simultanément jusqu'à la n-saturation avec n valant le nombre de variables dans la formule initiale : cette n-saturation comprend en fait le calcul de la table de verité, car les règles du calcul booléen (par exemple $\top \wedge \top = \top$) se déduise des règles de la 0-saturation. Cette n-saturation permet de conclure dans tous les cas :

– Une contradiction : tautologie.
– Pas de contradiction : pas de tautologie.

2.7 Intérêt de l'algorithme

L'intérêt de l'algorithme de Stålmarck réside dans sa vitesse d'exécution. Stålmarck a montré que si une n-saturation permet de résoudre un problème A (on dit que A est n-simple), alors le temps de calcul de l'algorithme est en $\mathcal{O}(|A|^{2n+1})$ où $|A|$ est taille de A (son nombre de connecteurs). Bien évidemment, si on doit aller jusqu'à n valant le nombre de variable de A, c'est nettement plus coûteux que la table de vérité. Et déjà rien qu'une 2-saturation peut être prohibitive... Par contre dans la pratique beaucoup de problèmes sont 1-simples, et un certain nombre sont même 0-simples. Il est même difficile de concevoir un exemple de tautologie qui ne soit pas 0-sat !

D'après l'article de John Harrison, les logiciels de la société fondée par Stålmarck tentent la 1-saturation, et en cas d'échec, cherchent directement un contre-exemple. D'où la règle expérimentale : 1-simple=tautologie ... et 2-hard (i.e non 1-simple) = too hard !

Pourquoi ces 0- et 1-saturations sont-elles aussi efficaces ?

– Les règles que l'on appliquent sont à double sens : si on sait des choses sur le bas (typiquement sur les variables), ces informations peuvent monter (sens "calcul"). Inversement si l'on a des informations sur le haut (typiquement grâce à l'équation de départ), il va y avoir propagation vers le bas.
– Lors du découpage de la formule en triplets, de nouvelles variables ont été introduites, et la 1-saturation porte aussi sur ces variables. Cela signifie que l'on va aussi faire des suppositions sur des sous-formules de la formule initiale, et non uniquement sur ces variables.

Si l'on compare avec la méthode par table de vérité, on met de l'information dans tout l'arbre et cette information monte et descend, au lieu d'en mettre uniquement en bas et de la faire monter. Et on essaie d'éviter un coup exponentiel en introduisant petit à petit les nouvelles hypothèses sur les variables. Il n'y a pas d'utilisation d'un arbre binaire des cas, comme pour les méthodes de table de vérités, ou de BDD [6]. On ajoute plutôt provisoirement de nouvelles hypothèses avant de revenir à la seule hypothèse "sommet=$\perp$" par intersection des équations communes.

3 Coq et CtCoq, petite visite

Ce paragraphe est un aperçu des documentations disponibles sur les deux systèmes ([1] [2] [3]).

3.1 Qu'est-ce que Coq ?

Réponse du Tutorial [1] :

> Coq est un système permettant le développement et la vérification de preuves formelles dans une logique d'ordre supérieure incluant un riche langage de définitions de fonctions.

On peut en effet écrire des fonctions, des énoncés de théorèmes, et prouver pas à pas ces théorèmes. On peut également exporter les fonctions que l'on a défini vers Ocaml, la dernière variante du langage fonctionnel Caml.

3.2 Le premier théorème

Lançons l'interpréteur de commandes `coqtop` , et énonçons un théorème :

```
speedy-~/coq $ coqtop
Welcome to Coq V6.2 (May 1998)

Coq < Theorem ma_tauto : (A,B,C:Prop)(A->(B->C))->(A->B)->(A->C).
1 subgoal

  ============================
   (A,B,C:Prop)(A->B->C)->(A->B)->A->C
```

Quelque commentaires : ce que l'on frappe est entre le < et le point, le reste correspond à la réponse de Coq. Concernant l'énoncé du lemme, le `(A,B,C:Prop)` signifie pour tout A, B, C de type `Prop` (type des propositions booléennes), et le reste devrait être clair... Pour ce qui est de la réponse de Coq, il nous dit qu'il y a un but à prouver, que c'est `(A,B,C:Prop)(A->B->C)->(A->B)->A->C` et qu'il n'y a pas d'hypothèses pour l'instant (rien au dessus de la ligne).

Comment allons nous prouver ce lemme ? En utilisant les règles, ou "Tactiques" de Coq. En effet Coq utilise le mode de raisonnement appelé "Natural

Deduction" : A chaque opérateur (p.ex. $\forall$, $\Rightarrow$), correspond des règles d'introduction et d'élimination. Ainsi on a l'introduction de $\Rightarrow$:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

Ce qui se lit : Si on a A parmi les hypothèses, et qu'on en déduit B, alors les hypothèses moins A permettent de déduire $A \Rightarrow B$. Et c'est exactement ce qu'on fait ici en utilisant la tactique `Intros` : on introduit les $\forall$ et les $\Rightarrow$:

```
ma_tauto < Intros.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A->B->C
  H0 : A->B
  H1 : A
  ============================
   C
```

Le but est maintenant C, à prouver à l'aide des variables A,B,C et des hypothèses H,H0 et H1. On applique alors l'hypothèse H, en utilisant la règle d'application de $\Rightarrow$:

$$\frac{\Gamma \vdash A \quad \Gamma \vdash A \Rightarrow B}{\Gamma \vdash B}$$

```
ma_tauto < Apply H.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  H : A->B->C
  H0 : A->B
  H1 : A
  ============================
   A

subgoal 2 is:
 B
```

Pour appliquer H, Coq demande qu'on prouve A, puis B. Pour A c'est facile : `Exact H1`. Pour B on fait `Apply H0`, Coq nous demamde de nouveau A, et de nouveau `Exact H1`. Accélérons un peu : après ces trois commandes, on a fini la preuve, et `Qed` la grave dans le marbre.

```
ma_tauto < Exact H1.
Subtree proved!

ma_tauto < Qed.
Intros.
Apply H.
Exact H1.

Apply H0.
Exact H1.

ma_tauto is defined
```

3.3 Qu'est-ce que CtCoq ?

CtCoq est une sur-couche graphique pour Coq, muni de multiples fonctionnalités permettant de rendre un peu plus intuitif et convivial l'usage de Coq. En vrac :

- "Pretty-Printing" des formules.
- L'édition peut être soit
  - structurée : si on clique sur `Theorem` dans le menu de la fig 4, CtCoq affiche `Theorem IDENT : FORMULA .` et il ne reste plus qu'à remplacer le champ `IDENT` par un nom de théorème, et à cliquer dans le menu qui s'est présenté pour compléter l'énoncé.
  - à la main : après avoir sélectionner une zone, `Esc-t` permet de l'éditer. On peut ainsi écrire tout un théorème, comme sous Coq. A la fin de l'édition (`Esc`), CtCoq "parse" l'expression entrée et la "pretty-print".
- On peut avoir plusieurs fenêtres avec plusieurs preuves menées simultanement, ce qui permet par exemple de prouver un lemme au milieu du théorème principal.
- Copier-Coller : entre les fenêtres ou non, preuves entières ou fragments, ...

– Navigation facile dans la preuve, retour en arrière, visualisation des différents buts : on peut ainsi commencer par prouver le dernier but.
– Manipulation des expressions à la souris (“Proof by Pointing”).
– Recherches parmi les théorèmes déjà écrits.
– Et bien plus encore...

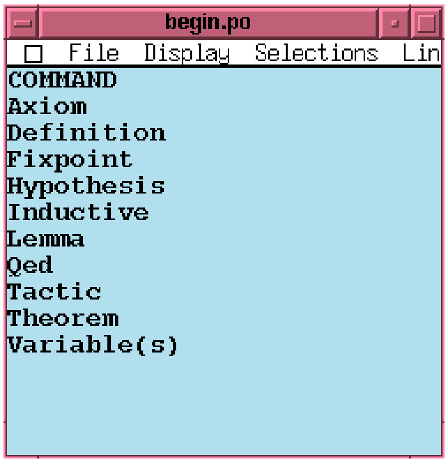

Fig. 4: Un exemple de menu de CtCoq

3.4 Un exemple

Décrivons un peu la fenêtre principale de CtCoq (cf fig 5), au cours d'une preuve :

– En haut se touve la partie édition : on y rentre les commandes. Une partie marron est un partie déjà exécutée, le jaune pâle ne l'a pas encore été, et le violet est en train de l'être. Sur l'image par exemple, seul l'énoncé du théorème a été exécuté.
– Au milieu, on trouve le but courant, une ligne, puis les hypothèses.
– En bas se trouve la partie “Search”. Y sont affichés les résultats des différentes recherches possibles. Ici on a demandé le type de `Prop` : c'est `Type`.
– Au passage, un certain nombre de boutons et d'informations sont disponibles sur la barre du milieu : `Do it` pour exécuter la commande sélectionnée, `Discard` pour revenir d'un cran en arrière, `Abort` (non

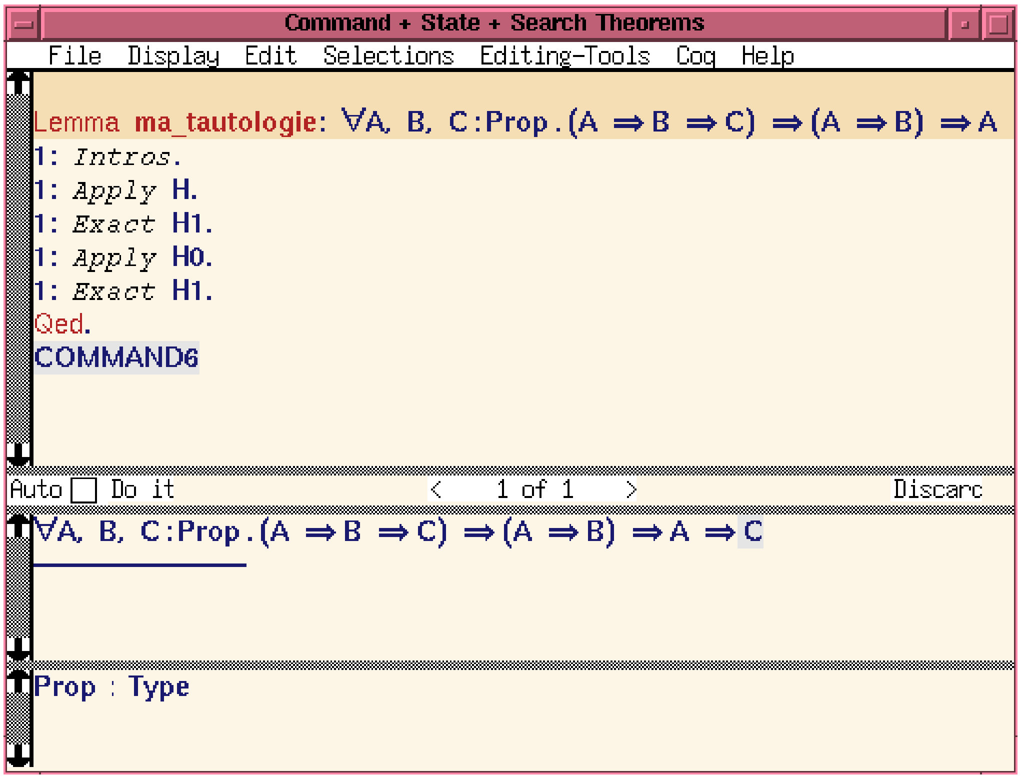

FIG. 5: Un théorème bien connu...

visible) pour arrêter la preuve, et `< 1 of 1 >` indique quel est le but courant, combien il y en a en tout, et permet de changer de but courant.

3.5 Pour ou contre CtCoq ?

Bref, CtCoq est un logiciel riche, très riche. Au point qu'un apprentissage certain est nécessaire pour l'utiliser efficacement. Et la présence à proximité de pourvoyeurs de "tuyaux" s'est révélée très utile, voire indispensable ... Il m'arrive encore fréquement de confondre fonctionnalités agaçantes et bugs.

Malgré tout, cette interface a été très utile et très pratique : je n'envisage pas de refaire le même travail (plus de 7000 lignes de fichiers sources) en mode texte avec Coq seul, ou même sous emacs..

4 Implémentation et Preuve

4.1 La Démarche

Elle a été très “linéaire” :

1. Ecrire une version de l'algorithme en Ocaml, pour se familiariser avec l'algorithme, en essayant d'être le plus fonctionnel possible, sachant qu'en Coq il faut se passer de références, d'exceptions et de tableaux.
2. Pour chaque partie de l'algorithme, traduire la fonction Caml en Coq, et prouver un théorème exprimant que cette fonction remplit ses spécifications.
3. Extraire le programme Coq global vers Ocaml.

Cette avance par petites étapes a plusieurs raisons. Déjà il a bien fallu apprendre Coq : d'où l'intérêt de traiter complètement la partie normalisation, plus facile, avant d'attaquer le reste. Ensuite le choix d'une structure de données et la façon de définir les fonctions influent terriblement sur les preuves. Il semble difficile de tout définir, puis de tout prouver. Ainsi le théorème de correction de la normalisation, qui faisait plus de 200 lignes dans sa version originale, s'est transformé en 5 lemmes de 10 lignes après modification de la fonction de normalisation.

Il y a donc deux buts lorsqu'on écrit une définition : faciliter les preuves qui vont suivre et rendre efficace le code Caml extrait. Et ces deux objectifs sont parfois conciliables, parfois non...

4.2 Quelle implémentation choisir ?

Dans son article, John Harrison décrit l'algorithme, mais également donne beaucoup d'indications sur son implémentation : il utilise une structure à base de pointeurs pour stocker les équations déjà obtenues.

Chaque variable v est confondue avec un entier positif $i > 1$, et par convention $-i$ représente $\neg v$. $\top$ et $\bot$ sont ausi considérés comme étant des variables, de représentants 1 et -1. Les équations obtenues sont donc vue comme des équivalences sur les entiers ($3 \sim -1$ pour $v_3 = \bot$ par exemple).

Au lieu de stocker toutes les équations, ce qui serait très redondant, on s'intéresse aux classes d'équivalence :

– la plus petite variable de la classe d'equivalence pointe vers la liste des autres éléments de la classe.

– les variables plus grandes pointent vers le minimum de leur classe.

L'ordre auquel il est fait allusion est l'ordre en valeur absolue sur les entiers ($1 \prec 3 \prec -5$). Y a-t-il unicité du minimum ? Apparemment non : n et -n peuvent être minimaux dans la même classe d'équivalence. En fait si l'on est dans ce cas, c'est qu' on a déduit l'équation $n = -n$ ce qui signifie $v_n = \neg v_n$: on s'arrête car on a trouvé une contradiction.

Avec la représentation ci-dessus, il est par exemple aisé de retrouver toute la classe à partir d'un de ses membres n :

Si n pointe vers une liste l (i.e. c'est un minimum de classe),
  Retourner `(cons n l)`
Sinon n pointe vers un entier $n0$ qui pointe vers une liste l
  Retourner `(cons n0 l0)`

Il est également facile de savoir si deux variables sont équivalentes : il suffit de comparer les minima de leurs classes respectives...

Cette implémentation a visiblement été pensée dans l'optique d'un langage tel que le C. Le langage de programmation associé à Coq, lui, est 100% fonctionnel, au point que les entiers sont des entiers de Peano : 5 s'écrit $(S(S(S(S(S\ O)))))$. Et avec des listes à la place des tableaux, chaque opération de lecture/écriture prend un temps linéaire...

Mais j'ai quand même suivi autant que possible cette structure de données “tableaux remplis d' entiers et de listes d'entiers”. On peut ainsi évaluer le prix à payer pour certifier une version relativement proche de ce qu'on écrirait dans un seul souci d'efficacité.

Le code extrait de Coq vers Ocaml est donc très lent, mais il est envisageable par la suite de remplacer entiers Peano par entiers Caml et listes par “array”, sans modifier la correction du programme.

4.3 La phase de normalisation

Comme je l'ai dis plus haut, c'est la partie la plus simple : il s'agit juste de faire une réecriture dans l'arbre et de prouver que la valeur de la formule a été conservée. Malgré tout, on rencontre déjà un certain nombre de traits propres à Coq.

4.3.1 Les fonctions totales

Définissons le type expr des expressions booléennes :

```
Mutual Inductive expr : Set :=
   T : expr
  | F : expr
  | Var : nat →expr
  | Not : expr →expr
  | And : expr → expr →expr
  | Or : expr → expr →expr
  | Impl : expr → expr →expr
  | Eq : expr → expr →expr .
```

On a alors envie d'écrire une fonction `normalise` de type expr → expr. Mais cette fonction n'est pas "surjective" : elle ne retourne jamais d'arbre ayant des noeuds Or ou Impl. On a alors envie de ne définir les fonctions suivantes que pour les noeuds Not, And, et Eq. En Caml, cela ne pose aucun problème, il suffit de faire des filtrages non exhaustifs. Mais en Coq les fonctions doivent être totales. On a alors le choix :

- rendre les fonctions totales en retournant n'importe quoi dans le cas des noeuds Ou et Impl, et alors il faut prouver qu'on n'arrive jamais dans un de ces cas Ou ou Impl,
- écrire une fonction `normalise` de type expr → expr2, où expr2 est défini en dessous.

```
Mutual Inductive expr2 : Set :=
   T2 : expr2
  | F2 : expr2
  | Var2 : nat →expr2
  | Not2 : expr2 →expr2
  | And2 : expr2 → expr2 →expr2
  | Eq2 : expr2 → expr2 →expr2 .
```

4.3.2 L'art de la preuve

...est un art difficile. Revenons sur l'affirmation "types de données et définitions influent énormement sur les preuves".

La fonction de normalisation dépend de l'opérateur au sommet de l'arbre e. S'il s'agit d'un opérateur binaire $e = p \otimes q$, on applique la normalisation récursivement sur p et q pour obtenir `(norm p)` et `(norm q)`, puis on rend un résultat fonction des opérateurs aux sommets de `(norm p)` et `(norm q)`. Par exemple si $e = p \wedge q$ et que `(norm p)=T2`, on retourne `(norm q)`.

Le théorème clé de cette partie s'énonce alors comme suit : pour toute valuation f des variables (de type nat → bool) et tout expression booléenne e, la valeur de e pour la valuation f est conservée par normalisation.

Theorem normalisation : (f :nat →bool) (e :expr)(eval f e)=(eval2 f (norm e)).

La preuve se fait classiquement par induction sur la structure de e. On prouve alors le théorème pour chaque cas de la définition. Or des cas, il y en a plus de 150 ! En effet on fait un `Cases (norm p) (norm q)`, qui génère 36 cas ((norm p) et (norm q) peuvent prendre chacun 6 valeurs), et ce pour chaque cas de e.

La définition, elle, ne fait pas 150 lignes, mais juste 45, grâce à des jokers (_ de Caml) dans les `Cases`. Par contre la preuve explore bien chaque cas : cela donne une preuve imbuvable de plus de 200 lignes.

Heureusement on peut faire plus simple :

1. distinguer les cas And, Or, ... n'est pas toujours utile, et il est profitable d'unifier ces cas :

   Mutual Inductive node1 : Set :=
     And1 : node1
    | Or1 : node1
    | Impl1 : node1
    | Eq1 : node1 .

   Mutual Inductive expr1 : Set :=
     T1 : expr1
    | F1 : expr1
    | V1 : nat →expr1
    | N1 : expr1 →expr1
    | Node1 : node1 → expr1 → expr1 →expr1 .

   Mutual Inductive node2 : Set :=
     And2 : node2
    | Eq2 : node2 .

   Mutual Inductive expr2 : Set :=
     T2 : expr2
    | F2 : expr2
    | V2 : nat →expr2
    | N2 : expr2 →expr2
    | Node2 : node2 → expr2 → expr2 →expr2 .

2. On gagne également à subdiviser la définition de `norm` grâce à des sous-fonctions `norm_not`, `norm_and`,...

```
Fixpoint norm[e :expr1] : expr2 :=
  Cases e of
     T1 ⇒ T2
    | F1 ⇒ F2
    | (V1 n) ⇒ (V2 n)
    | (N1 p) ⇒ (norm_not (norm p))
    | (Node1 n1 p q) ⇒ Cases n1 of
                          And1 ⇒ (norm_and (norm p) (norm q))
                         | Or1 ⇒ (norm_or (norm p) (norm q))
                         | Impl1 ⇒ (norm_impl (norm p) (norm q))
                         | Eq1 ⇒ (norm_eq (norm p) (norm q))
                       end
  end.
```

De cette façon la preuve de validité de norm devient :

```
Theorem th_norm : (f :nat →bool) (e :expr1)(eval1 f e)=(eval2 f (norm e)).
Induction e ; Clear e ; Simpl ; Auto.
Intros p H ; Rewrite th_norm_not ; Rewrite H ; Auto.
Intros n p Hp q Hq ; Case n.
Rewrite th_norm_and ; Rewrite Hp ; Rewrite Hq ; Auto.
Rewrite th_norm_or ; Rewrite Hp ; Rewrite Hq ; Auto.
Rewrite th_norm_impl ; Rewrite Hp ; Rewrite Hq ; Auto.
Rewrite th_norm_eq ; Rewrite Hp ; Rewrite Hq ; Auto.
Qed.
```

où `th_norm_not`, `th_norm_and` ... sont des lemmes de validité associés aux 5 sous-fonctions précédentes, chacun long d'une dizaine de lignes.

Moralité : ne pas avoir peur d'écrire des sous-fonctions et des sous-lemmes.

4.3.3 Le théorème de validité de la normalisation

L'expression e est une tautologie ssi son normalisé l'est aussi.

```
Definition tauto := [e :expr] (f :nat →bool)(eval f e)=true.

Definition tauto2 := [e :expr2] (f :nat →bool)(eval2 f e)=true.

Theorem tauto_tauto2 : (e :expr)(iff (tauto e) (tauto2 (norm (conv e)))).
```

4.4 La phase de découpage en triplets

4.4.1 Définition des triplets

On rentre dans le vif de l'algorithme, avec l'utilisation de triplets. Rappelons qu'il y a deux types de triplets : ceux d'opérateur $\wedge$ et ceux d'opérateur $\Leftrightarrow$. Réutilisons alors le type node2 précédant, qui prend justement ces deux valeurs :

```
Mutual Inductive triplet : Set :=
   Eq : node2 → int → int → int →triplet .
```

Le type triplet a donc Eq pour unique constructeur, qui prend en argument un node2 et trois entiers. Mais quels entiers, au fait ? Il ne peut pas s'agir des entiers de Peano de Coq (appelé nat), car il nous faut des entiers signés (toujours la convention $-i$ pour $\neg i$). Il y a bien une librairie d'entiers relatifs (ZARITH) en Coq, mais elle est assez complexe et il m'a semblé plus simple d'écrire un type int sur mesure :

```
Mutual Inductive int : Set :=
   Un : int
  | Moins_Un : int
  | P : nat →int
  | M : nat →int .
```

Un et Moins_Un sont les codages de $\top$ et $\perp$, (P n) et (M n) ceux de v_n et de $\neg v_n$ (P pour positif). Mon type int est donc en fait le type des littéraux. L'inconvénient de définir son propre type, c'est qu'il s'est avéré indispensable d'écrire toute une librairie de fonctions et de lemmes concernant ces int, comme par exemple les fonction abs ou moins, l'ordre en valeur absolue et tous ses axiomes, etc. Total : 512 lignes (cf figure 6).

4.4.2 L'algorithme de découpage

Il est récursif, sans malice. Il génère une liste de triplets, ainsi que l'entier codant le sommet de la formule. Un entier, initialisé au nombre de variable de l'expression de départ, permet de savoir quelle nouvelle variable générer lorsqu'on en a besoin. Noter au passage la syntaxe affreuse des listes et des paires...

```
Fixpoint make_triplets_rec[e :expr2] :
  (list triplet) → nat →(list triplet) * (int * nat) :=
  [l :(list triplet)] [n :nat]
    Cases e of
       T2 ⇒ (l, (Un, n))
      | F2 ⇒ (l, (Moins_Un, n))
      | (V2 i) ⇒ (l, ((P i), n))
      | (N2 p) ⇒ Cases (make_triplets_rec p l n) of
                   (pair l1 (pair s1 n1)) ⇒ (l1, ((moins s1), n1))
                 end
      | (Node2 node p q) ⇒ Cases (make_triplets_rec p l n) of
                              (pair l1 (pair s1 n1)) ⇒
                                Cases (make_triplets_rec q l1 n1) of
                                  (pair l2 (pair s2 n2)) ⇒
                                    ((cons (Eq node (P (S n2)) s1 s2) l2),
                                     ((P (S n2)), (S n2)))
                                end
                           end
    end.

Definition make_triplets : expr2 →(list triplet) * (int * nat) :=
  [e :expr2](make_triplets_rec e (nil triplet) (max_var e)).
```

4.4.3 L'invariant à prouver

Comment parler de tautologie lorsqu'il ne nous reste sur les bras qu'une liste de triplets, et une variable “sommet d'arbre” ?

```
Definition tauto_triplets :=
  [l :(list triplet)] [s :int] (f :nat →bool) (realise_triplets f l) →
  (eval_int f s)=true.

Theorem tauto2_tauto_triplets :
 (e :expr2)
 (iff
   (tauto2 e)
   (tauto_triplets (Fst (make_triplets e)) (Fst (Snd (make_triplets e))))).
```

Le couple (triplets, sommet) représente une tautologie si pour toute valuation f tel que f réalise chaque triplet, l'evaluation du sommet est $\top$. Et réaliser un triplet $p \Leftrightarrow (q \otimes r)$ signifie tout simplement que les valeurs de p, q et r déterminées par f rendent vrai la formule précédente.

Déjà, prouver le théorème précédent n'est plus complètement évident. Ce qu'on voudrait c'est un théorème disant que la valeur du sommet correspond à l'évaluation de la formule initiale. Mais ce n'est pas si simple à écrire, car on a ajouté des variables suppplémentaires. Une valuation f peut très bien être telle que `(eval f e)`$=\top$ mais `(eval_int f sommet)`$=\bot$: il suffit de prendre pour f des valeurs sur les nouvelles variables qui font que f ne réalise pas les triplets.

Une démarche, un peu laborieuse mais juste, consiste à faire comme suit :

Theorem Th_existence :
(f :nat →bool)
(e :expr2) (l :(list triplet)) (n :nat) (s :int) (make_triplets e)=(l, (s, n)) →
(Ex [g :nat →bool]
(egaux_avant (max_var e) f g) ∧
((realise_triplets g l) ∧ (eval_int g s)=(eval2 f e))).

Signification : partant d'une valuation f des seules variables de e, il existe (Ex) une valuation g de toutes les variables qui prolonge f sur les variables de e (`egaux_avant` ...), qui réalise les triplets et qui vérifie `(eval_int f sommet)` = `(eval f e)`. Pour prouver ce théorème, il suffit de calculer la valeur de l'expression e sous la valuation f pour trouver au fur et à mesure quelles valeurs donner aux nouvelles variables.

Ce dernier théorème nous donne déjà le sens $\Leftarrow$ du théorème sur les tautologies . Prenons en effet e tel que la (triplets,sommet) associée verifie `tauto_triplets` et une valuation f. La valuation g donnée par le théorème réalise les triplets. Donc `(eval_int g sommet)`$=\top$. Et donc `(eval f e)`$=\top$.

Puis ensuite il faut s'assurer de l' "unicité" de ce g, au moins sur les variables :

Theorem Th_unicite :
(f, g :nat →bool)
(e :expr2)
(l :(list triplet))
(n :nat)
(s :int)
(egaux_avant (max_var e) f g) →
(realise_triplets f l) →
(realise_triplets g l) → (make_triplets e)=(l, (s, n)) →
(eval_int f s)=(eval_int g s).

La combinaison des deux théorèmes précédents nous donne une version du Th_existence où ∃ est devenu ∀ :

Theorem Th_pour_tout :
(f, g :nat →bool)(e :expr2)
(l :(list triplet))(n :nat)(s :int)
(egaux_avant (max_var e) f g) →
(realise_triplets g l) → (make_triplets e)=(l, (s, n)) →
(eval_int g s)=(eval2 f e).

En particulier, lorsqu'on fait f=g dans ce théorème on obtient :

Theorem Th_recip :
(f :nat →bool)
(e :expr2)
(l :(list triplet))
(n :nat) (s :int) (realise_triplets f l) → (make_triplets e)=(l, (s, n)) →
(eval2 f e)=(eval_int f s).

Ce qui nous donne l'autre sens : prenons e une tautologie et f une valuation vérifiant les triplets. On a `(eval f e)`$=\top$ car e est une tautologie, et le théorème Th_recip donne `(eval_int f sommet)`$=\top$.

4.4.4 Raisonnement par l'absurde et décidabilité

Mais la définition de `tauto_triplets` ne correspond pas réellement à ce qu'on va utiliser par la suite. Rappelons le principe de la suite de l'algorithme : on suppose que la valeur du sommet est $\perp$, et on veut en déduire une contradiction. Il nous faut donc le théorème :

Theorem tauto_absurde :
(l :(list triplet))(s :int)
(iff
(tauto_triplets l s)
((f :(nat→bool))(realise_triplets f l) → (eval_int f s)=false → False)).

Cela peut se voir comme un bête raisonnement par l'absurde : Si je suppose que `(eval_int f sommet)`$=\perp$, j'obtiens une contradiction (False).

Mais attention ! Coq est bâti sur une logique intuitioniste (dite encore constructive) : le raisonnement par l'aburde $(\neg A \Rightarrow \perp) \Leftrightarrow A$ ou encore le tiers-exclu $A \vee \neg A$) ne font pas partie des choses vraies en général. Il faut donc montrer au coup par coup que l'on a le droit d'appliquer ce genre de règles.

Ici il n'y a aucun problème : le type bool n'a que deux constructeurs true et false, et tout élément de ce type est soit égal à true, soit à false .

Par contre il est bien plus dur de prouver un théorème de "décidabilité" comme celui sur `tauto2` :

Theorem tauto2_dec : (e :expr2){(tauto2 e)}+{¬ (tauto2 e)}.

qui exprime que e est une tautologie ou n'en est pas une (sic !). Le problème viens du fait que par souci de simplicité, je prends les valuations parmi nat→bool, qui est un ensemble infini, et non parmi [0..n]→bool, n étant le nombre de variable de e. Il est donc impossible de dire "essayer toutes les valuations, et ...". Il faut donc montrer qu'on peut ramener cet ensemble infini à un nombre fini de valuations significatives.

4.5 La phase de 0-saturation

C'est la partie la plus dur, la partie suivante de n-saturation s'inspirant grandement des techniques utilisées ici.

4.5.1 Les tableaux

Il s'agit maintenant de définir une structure de donnée pour stocker un ensemble d'équations. On a vu que John Harrison utilise un tableau dont les case sont soit des entiers, soit des listes d'entiers. Ici pas de tableaux, donc place aux listes...

```
Mutual Inductive assignement : Set :=
    V : int →assignement
  | L : (list int) →assignement .

Mutual Inductive etat : Set :=
    Contrad : etat
  | Cex : etat
  | Ras : etat .

Mutual Inductive tableau : Set :=
    Tab : etat → (list assignement) →tableau .
```

L'état associé à un tableau permet de marquer si les équations stockées dans ce tableau sont contradictoires, si elle permettent d'obtenir un contre-exemple (toutes les variables affectées à $\top$ ou $\bot$, et pas de contradiction), ou encore ni l'un ni l'autre (R.A.S).

4.5.2 Des preuves en arguments...

C'est là que les ennuis commencent : quand on a un tableau, typiquement on veut pouvoir lire/écrire dedans. Par exemple définir une fonction `valeur` de type `tableau` → `int` → `assignement` qui retourne le n-ième élément du tableau a. Mais les fonctions Coq sont toujours totales : que retourner si `n>(length a)` ? Les solutions ne manquent pas :

- Convenir d'une valeur, par exemple (L nil), à retourner dans ce cas. Outre le fait que cette solution n'est pas très élégante, il y a ambiguïté : si la fonction retourne (L nil), la lecture a-t-elle réussi ? On doit donc prouver un certain nombre de chose par la suite.
- Créer une valeur exceptionnelle dans le type assignement, grâce à un nouveau constructeur Erreur_Lecture en plus de V et L. Mais la signification du type assignement est modifiée...
- Encapsuler : créer un type résultat_lecture, et faire une fonction de lecture qui retourne (Résultat_Normal x) ou Erreur_Lecture. C'est une solution "pure", mais lourde : après chaque lecture, un `Case` s'impose...
- S'assurer en amont que le paramètre n est bien tel que `(n < (lenght a))`, et fournir la preuve de ce fait en argument à la fonction de lecture. Et oui ! En Coq une preuve est un élément comme un autre, son type est en fait son énoncé, et sa valeur le contenu de la preuve. Notre fonction de lecture aura donc pour type `(a:tableau)(n:int)(n < (lenght a))`→`int`, ce qui signifie "pour tout tableau a, pour tout entier n, prendre une preuve que `(n < (lenght a))` et rendre un entier"

Autant le dire tout de suite, c'est la dernière méthode que j'ai choisie, car c'est la solution philosophiquement idéale, puisqu'on **sait** que le programme va faire des lectures correctes. Ainsi lors de l'extraction vers Ocaml, le paramètre de preuve disparait tout simplement, et il ne reste plus qu'une fonction de type tableau → int → assignement. Et sur le plan pédagogique, cette méthode, spécifique à Coq, était très intriguante et tentante. Par contre c'est un choix définitif : si l'on écrit une fonction ayant des preuves en arguments, à chaque appel de cette fonction, il faudra lui fournir des preuves, et donc soit les créer, soit les hériter en arguments, etc. Après la première fois, la majorité des fonctions qui suivent prennent aussi des preuves en arguments. Et c'est parfois très pénible et très lourd.

Il est par exemple très dur de manipuler une preuve à l'intérieur d'une fonction "normale", c'est à dire comme on en a vu jusqu'à maintenant. On est alors obligé d'utiliser une autre méthode de définition, la "définition par preuve" : on se contente de donner le type d'une fonction, et Coq nous demande, de la même façon qu'on prouve un théorème, de donner un représentant de ce type, qui prendra le nom de la fonction voulue. Exemple : définissons notre fonction de lecture.

```
Definition valeur : (a :tableau) (i :int) (in_tab i a) →assignement.
Intros a i ; Case a ; Intros.
Cut (lt (position i) (length l)).
Intros N ; Exact (if_neg_opp i (nieme l (position i) N)).
Inversion H ; Auto.
Defined.
```

C'est évidement beaucoup moins intuitif qu'une définition à la Caml, mais c'est plus puissant : il est possible de tout définir ainsi, et pour les fonctions ayant des preuves en paramètres, c'est souvent la seule solution.

4.5.3 Tableaux valides ou invalides

Les tableaux, tels qu'on les a définis, ne remplissent pas toutes les spécifications qu'on veut leur imposer (cf 4.2). Rien n'empêche une case de renvoyer à une autre case, qui renvoie à une troisième, alors qu'en tant que minimum d'une classe d'équivalence, la deuxième case devrait contenir une liste d'éléments. Une propriété essentielle des tableaux va donc être la validité :

```
Mutual Inductive valide_int[a :tableau ; i :int] : Prop :=
    valide_int_V :
      (j :int)
      (l :(list int))
      (hl :(sorted l))
      (h :(in_tab i a))
      (h' :(in_tab j a))
      (valeur a i h)=(V j) →
      (valeur a j h')=(L l) → (In i l) → (lt_va j i) →(valide_int a i)
   | valide_int_L :
      (l :(list int))
      (hl :(sorted l))
      (h :(in_tab i a))
      (valeur a i h)=(L l) →
      (in_tab_rec l a) → (tous_egaux l a i) → (inf_list i l) →
      (valide_int a i) .

Definition valide :=
  [a :tableau]((i :int) (in_tab i a) →(valide_int a i)) ∧ (lt O (taille a)).
```

Et la grande majorité des fonctions et des théorèmes sur les tableaux vont avoir un type commençant par (a :tableau)(ha :(valide a))..., ce qui revient à dire "pour tout tableau valide ...". Il aurait même été possible de définir un type tableau_valide, à un seul constructeur Tab_et_preuve prenant un tableau et la preuve de sa validité.

Un des gros morceaux de cette partie a été de prouver que ma définition de l'ajout d'une équation dans le tableau conserve la validité. Ajouter une équation $x = y$, c'est en fait unifier les classes de x et de y. Cela se fait en manipulant les pointeurs : soit x_0 et y_0 les minima des classes de x et de y ; si $x_0 \prec y_0$ alors x_0 devient le minimum de la nouvelle classe, et désormais tous les élements de l'ancienne classe de y pointent vers lui. C'est (relativement) simple à dire, ça l'est moins quand il faut définir la fonction en question, et ça ne l'est plus du tout quand il faut prouver que la validité est conservée... plus de 900 lignes pour cela !

4.5.4 Induction structurelle, induction bien fondée

Une fois qu'on a les briques élementaires que sont lecture/écriture, ajout d'une équation, et test si une équation a déjà été déduite, on peut passer à la 0-saturation proprement dite. Elle se décompose en trois sous-fonctions :

1. Une fonction qui prend un triplet, et regarde si ce triplet plus les équations "contenues" dans le tableau a, permettent de déduire d'autres équations selon les règles décrites auparavant.
2. Une autre fonction qui itère la première sur la liste des triplets.
3. Une troisième qui itère la deuxième jusqu'à arriver dans un état stationnaire où plus aucune équation ne peut plus être trouvée.

Pas grand chose à dire de la première fonction, c'est une longue suite de "if ... then ... else if ...". Les problèmes avec cette fonction ont plutôt été d'ordre matériel : avec la version 6.2 de Coq, il était indispensable d'utiliser le plus gros serveur de l'équipe, soit un Alpha 533Mhz avec 256 Mo de RAM. En effet ces douze niveaux de "if ...then ...else if ..." demandaient plus de 200 Mo de RAM pour être digérés par Coq... Heureusement, la nouvelle version (6.2.1), arrivée en cours de stage, a reglé le problème[1] : 20 Mo sont maintenant suffisants.

Pour la deuxième, rien à signaler, c'est une itération standard sur une liste.

Par contre, la fonction qui itère "jusqu'à plus soif" pose un problème théorique : va-t-elle s'arrêter ? Encore une fois, en Caml on **sait** que la fonction

[1]Encore que... lors de la définition d'une autre fonction, Coq 6.2.1 est monté à 115 Mo de RAM... Cette fonction s'est révélée fort heureusement inutile et je l'ai abandonnée.

va s'arrêter, et si on se trompe ... `Interrupt Caml`. Ici il est assez génant de prouver des propriétés sur un programme qui peut boucler. Et donc Coq ne l'autorise pas : il va donc falloir prouver l'arrêt au moment de la définition.

Mais au fait, comment se fait-il que ce problème n'a pas été rencontré plus tôt ? Ce n'est pas la première fonction récursive que l'on écrit. Mais c'est la première qui ne soit pas récursive structurelle. Une récurrence est structurelle si elle porte sur la structure même d'un type de données. Ainsi pour les entiers de Peano, lorsqu'on prouve une propriété (P n) pour tout n, n vaut soit O et c'est l'initialisation, soit (S n') et c'est l'étape de récurrence. De même pour les listes (nil et cons).

Pour la 0-saturation, on n'est plus dans ce cas. Il faut alors utiliser "l'induction bien fondée" : exhiber un ordre sur un paramètre qui

1. décroisse à chaque appel de la fonction
2. soit bien fondé (pas de suites infinies décroissantes).

Inutile de dire que cela peut être très pénible, surtout la preuve de bon ordre...

Heureusement ici, il y a un paramètre assez simple qui décroit strictement lorsqu'on itère la fonction : il s'agit du nombre de classe d'équivalence. En effet si pendant un tour le nombre de classes d'equivalence stationne c'est qu'on n'en a unifié aucunes : il n'y a pas de nouvelles équations déductibles, et on s'arrête. A l'inverse s'il y a de nouvelles équations déductibles, le nombre de classes décroit. On a donc trouvé un ordre sur les tableaux : il suffit de comparer le nombre de case contenant des listes. Et cet ordre a le bon goût d'être la composée de la fonction "nb_de_case_à_liste" avec l'ordre standard sur les entiers naturels : il existe alors directement un théorème affirmant que cet ordre sur les tableaux est bien fondé.

4.5.5 La preuve de correction de la 0-saturation

Il y a en fait deux morceaux :

- Prouver que les équations déduites sont correctes, c'est à dire que pour toute valuation f réalisant les triplets et telle que `(eval_int f sommet)`$=\bot$, et pour toute équation x=y déduite, on a `(eval_int f x)=(eval_int f y)`.
  Cette partie assure que si l'on trouve une contradiction x=(-x), alors `(eval_int f x)=(eval_int f (-x))`, ce qui ce reduit à False. On peut alors utiliser le théorème tauto_absurde (cf partie 4.4.4) pour conclure.
- Prouver que si toutes les variables sont affectées à $\top$ ou $\bot$ à la fin de la 0-saturation, et que l'on a pas trouvé de contradiction, c'est que l'on

est en présence d'un contre-exemple.

Ces deux théorèmes n'ont pas pu être prouvés au cours du stage, mais dans le mois qui a suivi, sur mon PC.

4.6 La phase de p-saturation

Lors du stage, seules les définitions de cette partie ont pu être écrites, faute de temps. Mais même écrire les définitions nécessite de prouver des propriétés (toujours la faute aux preuves en arguments). Aussi ai-je utilisé un certain nombre de fois `Hypothesis`, qui permet de fournir des axiomes à Coq. Ces axiomes sont parfaitement similaires aux théorèmes, simplement ils n'ont pas été prouvés.

On peut donc par la suite revenir en arrière, remplacer `Hypothesis` par `Theorem` et fournir une démonstration, en étant sûr que le comportement de ce qui utilisait cet axiome ne va pas changer. Il faut simplement éviter de mettre en axiomes des choses fausses...

Grâce à cela, j'ai pu aller jusqu'au bout de la partie définition dans le temps imparti, et j'ai pu procéder à l'extraction vers Ocaml.

La encore, j'ai pu finir cette partie sur mon PC personnel doté de 32 Mo de mémoire vive, preuve que la nouvelle version de coq est bien plus avare en place mémoire.

4.6.1 La fonction d'intersection

La seule réelle nouveauté de la n-saturation sur la 0-saturation, c'est la fonction d'intersection de deux tableaux.

D'ailleurs ce n'est pas réellement une fonction d'intersection, mais plutôt d'enrichissement du tableau original a par les parties communes de a1 et a2, les tableaux déduits de a par disjonction sur un variable et saturation. Plutôt que de construire un nouveau tableau, il est en effet plus efficace de rajouter par dessus a. Pour chaque élement x du tableau a, on regarde les listes l1 et l2 des éléments de a1 et a2 qui lui sont égaux. Soit x_0 le plus petit élément commun à l1 et l2. Coq est très content : cet élément existe toujours, car x, égal à lui-même, est dans l1 et l2. Et x_0 = x dans a1 et dans a2, donc dans a : on rajoute donc l'équation x=x_0 dans a.

Mais il y a moyen de gagner en efficacité, grâce aux états (Contrad, Cex et Ras) d'un tableau.

Si a1 est dans l'état Cex (il contient un contre-exemple), alors c'est fini : un contre-exemple est un contre-exemple, peu importe les hypothèses qu'on a fait pour l'obtenir. En Caml on jèterait une exception qui terminerait le programme, ici on se contente de retourner a1 comme intersection de a1 et a2.

Si a1 est dans l'état Contrad (il est contradictoire), alors cela signifie qu'il contient toutes les équations, et l'on retourne a2 comme intersection de a1 et a2.

Il n'y a donc que dans le cas où a1 et a2 sont dans l'état Ras qu'il faut calculer l'intersection comme décrit auparavant.

4.6.2 La p-saturation proprement dite

Il ne reste plus qu'à faire une fonction `(combi n p)` qui génère les $\binom{n}{p}$ manières de prendre p variables parmi le total des n variables, génerer pour chaque manière toutes les façons d'affecter $\top$ ou $\perp$ aux p variables.

Pour chaque valuation de p variables ainsi obtenue, on rentre les nouvelles équations dans le tableau a et on 0-sature. Puis on prend l'intersection de tout ce petit monde, deux par deux.

Et on recommence, jusqu'à ce que le processus stationne. Et il s'arrête bien, exactement pour les mêmes raisons que la 0-saturation (ce n'est pas pour rien que 0-saturation est un cas particulier de p-saturation ...).

L'ennui c'est que chaque boucle (pour parler langage itératif), chaque recursion (langage fonctionnelle), se traduit par une fonction Coq dont il faut prouver les propriétés. Ainsi la p-saturation se compose de trois boucles principales :

Repeter
  Pour toutes les combinaisons de p variables parmi n **Faire**
    Pour chaque variable de la combinaison choisie **Faire**
      ...
Tant que apparition de nouvelles équations

Cela fait donc 3 fonctions à écrire, et pour chacune un théorème de terminaison, 3 ou 4 de conservation de quelque chose, etc. Au final le copier/coller fonctionne beaucoup, et la preuve globale enfle énormement..

5 Les résultats

La version pessimiste : un programme extrait qui marche mais qui est au moins 20 fois plus lent que la version Ocaml écrite au début du stage. Et 13000 lignes écrites pour ça, dont 7500 lors du stage, et 5500 après.

Nom de la fonction	Nb de lignes	Nb de définitions	Nb de théorèmes	Taille moyenne par th/def
Int	512	15	52	8
Make_triplets	785	12	31	18
Tab	1638	30	97	13
Affecte	1785	1	40	44
Do_equ	897	2	30	28
Do_triplet	1451	5	50	26
Do_triplets	385	3	18	18
Sat0	813	6	25	26
Intersection	951	4	33	26
Dilemma	823	4	30	24
Dilemmas	530	3	24	20
Satn	662	7	28	26
Stalmarck	500	5	22	19
Total	13065	134	566	19

FIG. 6: Un peu de statistique : récapitulatif par fonction

La version optimiste : la preuve existe, ce n'est déjà pas si mal : l'URL http ://www.inria.fr/croap/CFC/stalmarck donne l'accès à une version hypertexte de la preuve.

Theorem stalmarck_correcte : (e :expr)(iff (stalmarck_global e)=true (tauto e)).

Elle est lourde, sans doute maladroite, mais se "recompile" (i.e. se vérifie automatiquement) sans trop de mal (18 min sur une machine puissante). Quant au programme extrait, il fait 1075 lignes, et n'est pas un modèle de vitesse :

1. l'exemple $(a \Rightarrow b \Rightarrow c) \Rightarrow (a \Rightarrow b) \Rightarrow (a \Rightarrow c)$ de la figure 7 est quasi instantané (la 0-saturation marche)
2. par contre, $(a \wedge b) \vee (a \wedge \neg b) \vee (\neg a \wedge b) \vee (\neg a \wedge \neg b)$ nécessite déjà 10 secondes. Il est vrai que cet exemple nécessite une 1-saturation.

Je n'ai pas eu le temps de pousser plus avant les tests, mais il doit y avoir moyen d'accélérer sérieusement le programme extrait en remplaçant (automatiquement ?) les Peano par des int Caml, et surtout les listes par des tableaux.

```
        Objective Caml version 1.07

# #use "extraction.ml";;
type bool = | True | False
type nat = | O | S of nat
type ('a, 'b) prod = | Pair of 'a * 'b
...
type 'a list = | Nil | Cons of 'a * 'a list
...
type int = | Un | Moins_Un | P of nat | M of nat
...
type assignement = | V of int | L of int list
type tableau = | Tab of bool * assignement list
...
val nieme : 'a list -> nat -> 'a = <fun>
val valeur : tableau -> int -> assignement = <fun>
...
val modif : tableau -> int -> assignement -> tableau = <fun>
...
val do_equ : tableau -> int -> int -> (bool, tableau) prod = <fun>
type expr =
  | T
  | F
  | Var of nat
  | Not of expr
  | And of expr * expr
  | Or of expr * expr
  | Impl of expr * expr
  | Eq of expr * expr
...
type triplet = | Eq0 of node2 * int * int * int
val sat0 : tableau -> triplet list -> tableau = <fun>
...
val combi : nat -> nat -> int list list = <fun>
...
val sat_p : nat -> tableau -> triplet list -> tableau = <fun>
...
val stalmarck_global : expr -> bool = <fun>
#  stalmarck_global (Impl
   (Impl (Var (O), Impl (Var (S O), Var (S (S O)))),
    Impl (Impl (Var O, Var (S O)), Impl (Var O, Var (S (S O))))));;
- : bool = True
#
```

FIG. 7: Remplacez (Var (O)) par a, (Var (S (O))) par b, et (Var (S (S (O)))) par c. Cela ne vous rappelle rien ?

6 Conclusion

La démarche suivie lors de ce stage est assez particulière. Il m'a suffit de 2 à 3 jours, au début du stage, pour implémenter l'algorithme en Ocaml, convertisseur chaines de caractères vers expressions boolénnes compris. Par contre écrire le même programme en Coq et le prouver a nécessité plus de 2 mois, le tout pour obtenir une version Ocaml qui marche beaucoup moins vite que la version initiale. Certifier un programme par cette méthode semble donc très coûteux en temps, et fatal à l'efficacité du dit programme.

C'est vrai, cette comparaison est faussée par le fait que je n'ai pas fait l'apprentissage de Caml dans les 2 ou 3 jours en question, alors que je suis parti de zéro concernant Coq. Lorsque je reprends de vieilles preuves, il n'est pas rare qu'elles fondent de moitié. Et je découvre encore de nouvelles tactiques : par exmple je me suis aperçu sur le tard qu'il était possible de simplifier non seulement le but courant (Simpl) mais aussi n'importe laquelle des hypothèses courantes (Simpl in H).

Par contre Coq reste globalement très lourd à manier. Cela s'améliore au fil des versions, et on n'est plus obligé de parler λ-calcul couramment pour l'utiliser. Et il semble (soyons optimiste) que le bug qui faisait que 200 Mo de mémoire vive était parfois un minimum soit réglé. Par contre l'absence d'outils comme les exceptions est vraiment très contraignant.

Le principe de Coq est par contre enthousiasmant : on peut renter les axiomes de n'importe quelle théorie, et prouver les théorèmes de cette théorie. On peut également réutiliser les librairies[2] de quelqu'un d'autre en un vrai jeu de Légo. Peut-être que quelqu'un utilisera ma fonction dans un autre programme où les tautologies sont utilisées, et se servira de la preuve de sa correction.

En même temps la critique est facile : à quoi ça sert ? Coq n'aide pas à trouver de nouveaux théorèmes, bien au contraire il fait avoir une idée très claire de la démarche pour écrire une preuve Coq. Ce n'est pas non plus de la démonstration automatique : le nombre de théorème que l'on peut prouver par Auto seul est quasiment nul. Est-ce bien de la démonstration assistée par ordinateur, ou plutôt de la démonstration sur ordinateur ? Auto aide bien dans les cas triviaux, évite parfois de vérifier les hypothèses d'un théorème qu'on applique. Mais à coté de cela combien de lemmes évidents doit-on énoncer avant que la tactique Auto ne gagne ? L'avantage essentiel de Coq

[2] A ce propos les librairies standards de Coq sont perfectibles. Par exemple il n'y a pas de librairies sur le maximum de deux entiers, alors qu'il y en a sur le minimum

c'est la sécurité : pour un théorème mathématique, il n'y a pas de partie soit-disant triviale qui puissent échapper à Coq, et pour un algorithme pas de comportement sortant des spécifications qui puisse arriver.

Mais encore une fois il y des limitations à ce côté sécurité. Du côté informatique, l'argument classique, c'est que la première Ariane 5 a explosé à cause d'un "overflow" non prévu. Si le programme avait été certifié , etc etc etc. Mais dans mon exemple certifié, le code extrait peut aussi échouer à cause d'un dépassement mémoire : il suffit de trouver la taille limite d'un objet Caml, et d'envoyer une expression plus grosse encore. La seule chose dont je soit sûr, c'est que si le programmme répond "tautologie" ou "non tautologie", il a raison. Et c'est déjà mieux que rien ...

A Liste des règles de simplification

On applique ces règles de bas en haut dans l'arbre. Et s'il y en a plusieurs qui peuvent s'appliquer au même endroit, on prend la première solution.

1. Élimination des $\top$ et $\bot$:

$$\neg\top \rightarrow \bot \tag{7}$$
$$\neg\bot \rightarrow \top \tag{8}$$
$$p \wedge \top \rightarrow p \tag{9}$$
$$\top \wedge p \rightarrow p \tag{10}$$
$$p \wedge \bot \rightarrow \bot \tag{11}$$
$$\bot \wedge p \rightarrow \bot \tag{12}$$
$$p \vee \top \rightarrow \top \tag{13}$$
$$\top \vee p \rightarrow \top \tag{14}$$
$$p \vee \bot \rightarrow p \tag{15}$$
$$\bot \vee p \rightarrow p \tag{16}$$
$$p \Rightarrow \top \rightarrow \top \tag{17}$$
$$\top \Rightarrow p \rightarrow p \tag{18}$$
$$p \Rightarrow \bot \rightarrow \neg p \tag{19}$$
$$\bot \Rightarrow p \rightarrow \top \tag{20}$$
$$p \Leftrightarrow \top \rightarrow p \tag{21}$$
$$\top \Leftrightarrow p \rightarrow p \tag{22}$$
$$p \Leftrightarrow \bot \rightarrow \neg p \tag{23}$$
$$\bot \Leftrightarrow p \rightarrow \neg p \tag{24}$$

2. Simplification des négations :

$$\neg\neg p \quad \rightarrow \quad p \tag{25}$$

3. Élimination des disjonctions :

$$\neg p \vee \neg q \quad \rightarrow \quad \neg(p \wedge q) \tag{26}$$
$$\neg p \vee q \quad \rightarrow \quad \neg(p \wedge \neg q) \tag{27}$$
$$p \vee \neg q \quad \rightarrow \quad \neg(\neg p \wedge q) \tag{28}$$
$$p \vee q \quad \rightarrow \quad \neg(\neg p \wedge \neg q) \tag{29}$$

4. Élimination des implications :

$$\neg p \Rightarrow \neg q \quad \rightarrow \quad \neg(\neg p \wedge q) \tag{30}$$
$$\neg p \Rightarrow q \quad \rightarrow \quad \neg(\neg p \wedge \neg q) \tag{31}$$
$$p \Rightarrow \neg q \quad \rightarrow \quad \neg(p \wedge q) \tag{32}$$
$$p \Rightarrow q \quad \rightarrow \quad \neg(p \wedge \neg q) \tag{33}$$

B Liste des règles de 0-saturation

1. Si l'on a le triplet $p \Leftrightarrow (q \wedge q)$:
   - De l'équation $p = \neg q$, déduire $q = \top$ et $r = \bot$.
   - De l'équation $p = \neg r$, déduire $q = \bot$ et $r = \top$.
   - De l'équation $q = r$, déduire $p = q$.
   - De l'équation $q = \neg r$, déduire $p = \bot$.
   - De l'équation $p = \top$, déduire $q = \top$ et $r = \top$.
   - De l'équation $q = \top$, déduire $p = r$.
   - De l'équation $q = \bot$, déduire $p = \bot$.
   - De l'équation $r = \top$, déduire $p = q$.
   - De l'équation $r = \bot$, déduire $p = \bot$.
2. Si l'on a le triplet $p \Leftrightarrow q \Leftrightarrow r$:
   - De l'équation $p = q$, déduire $r = \top$.
   - De l'équation $p = \neg q$, déduire $r = \bot$.
   - De l'équation $p = r$, déduire $q = \top$.
   - De l'équation $p = \neg r$, déduire $q = \bot$.

– De l'équation $q = r$, déduire $p = \top$.
– De l'équation $q = \neg r$, déduire $p = \bot$.
– De l'équation $p = \top$, déduire $q = r$.
– De l'équation $p = \bot$, déduire $q = \neg r$.
– De l'équation $q = \top$, déduire $p = r$.
– De l'équation $q = \bot$, déduire $p = \neg r$.
– De l'équation $r = \top$, déduire $p = q$.
– De l'équation $r = \bot$, déduire $p = \neg q$.

Références

[1] G. Huet, G. Kahn, C. Paulin-Mohring, The Coq Proof Assistant : A Tutorial : Version 6.1, Rapport Technique INRIA 0204, août 1997. (http ://www.inria.fr/RRRT/RT-0204.html)

[2] The Coq Reference Manual : Version 6.2 (http ://pauillac.inria.fr/coq/doc-fra.html)

[3] Janet Bertot, Yves Bertot, Yann Coscoy, Healfdene Goguen, and Francis Montagnac, User Guide to the CtCoq Proof Environment, February 1996.

[4] J. Harrison, "Stålmarck's algorithm as a HOL derived rule", TPHOLs'96. (http ://www.cl.cam.ac.uk/users/jrh/papers/stalmarck.html)

[5] S.A. Cook The complexity of theorem-proving procedures. In Proceedings of the 3rd ACM Symposium on the Theory of Computing,pages 151-158, 1971.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, C-35 :677-691, 1986.